Semantic-Preserving Transformations for Stream Program Orchestration on Multicore Architectures

A thesis submitted in fulfilment of the requirements for the
degree of Doctor of Philosophy in the School of Information Technologies at
The University of Sydney

Yousun Ko

2016

ABSTRACT

Because the demand for high performance with big data processing and distributed computing is increasing, the stream programming paradigm has been revisited for its abundance of parallelism in virtue of independent actors that communicate via data channels. The synchronous data-flow (SDF) programming model is frequently adopted with stream programming languages for its convenience to express stream programs as a set of nodes connected by data channels. Unlike general data-flow graphs, SDF requires the specification of the number of data items produced and consumed by a node already at compile-time. Static data-rates enable program transformations that greatly improve the performance of SDF programs on multicore architectures. The major application domain is for SDF programs are digital signal processing, audio, video, graphics kernels, networking, and security.

The major optimization objective with stream programs is data throughput. Stream program orchestration is a term that denotes compiler optimizations and run-time techniques that aim at performance improvements of stream programs on multicore architectures. A large body of research has already been devoted to stream program orchestration. Nevertheless, current compilers and run-time systems for stream programming languages are not able yet to harvest the raw computing power of contemporary parallel architectures. We identify data channels as the dominating roadblock for achieving high performance of SDF programs. Data channels between communicating nodes, i.e., between a producer and a consumer, employ FIFO-queue semantics. Funneling a data item (token) from a producer to a consumer through a FIFO queue incurs non-negligible overhead. The producer is required to perform an enqueue-operation, followed by a dequeue operation in the consumer. The enqueue and dequeue operations induce the run-time overhead of the underlying queue implementation. Queues on shared-memory multicores are implemented as buffers that are indexed via read- and write-pointers. Enqueueing and dequeuing tokens via such indirect address operations obscures the data-dependencies between producer and consumer. As a result, compiler optimizations are rendered ineffective. Although FIFO queues are a valuable abstraction mechanism to separate concerns (i.e., implementation details) between producer and consumer nodes, they represent an insurmountable abstraction barrier for current optimizing compilers.

This thesis makes the following three contributions that improve the performance of SDF programs: First, a new intermediate representation (IR) called LaminarIR is introduced. LaminarIR replaces FIFO queues with direct memory accesses to reduce the data communication overhead and explicates data dependencies between producer and consumer nodes. We provide transformations and their formal semantics to convert conventional, FIFO-queue based program representations to LaminarIR. Second, a compiler framework to perform sound and semantics-preserving program transformations from FIFO semantics to LaminarIR. We employ static program analysis to resolve token positions in FIFO queues and replace them by direct memory accesses. Third, a communication-cost-aware program orchestration method to establish a foundation of LaminarIR parallelization on multicore architectures. The LaminarIR framework, which consists of the aforementioned contributions together with the benchmarks that we used with the experimental evaluation, has been open-sourced to advocate further research on improving the performance of stream programming languages.

Keywords : Multicore Architecture, Stream Programming Languages, Synchronous Data Flow, Compiler Optimization, Performance Analysis, Static Program Analysis

국문요약

멀티코어 컴퓨터구조에서의 스트림 프로그램 편성을 위한 의미 보존적 프로그램 변환에 대한 연구

최근 효율적인 빅데이터 처리방식과 분산처리 시스템에 대한 요구가 증가함에 따라, 계산을 수행하는 액터와 액터간의 통신을 위한 데이터 채널을 독립적으로 정의하는 스트림 프로그래밍 패러다임의 풍부한 병렬성이 재조명 되고 있다. 동기성 데이터 흐름(synchronous data flow) 프로그래밍 모델은 스트림 프로그래밍 언어를 꼭짓점과 그 점을 잇는 변의 집합, 즉 그래프로 추상화하는 데에 용이하게 활용된다. 일반적인 데이터 흐름 그래프와는 달리, 동기성 데이터 흐름 그래프로 표현되는 프로그램은 프로그램을 구성하는 최소 단위인 액터(actor)가 소모하고 생성하는 데이터의 양을 소스코드 단에서 미리 정의하고, 이 특성으로 인해 동기성 데이터 흐름 프로그램은 추가적인 메모리 소모 없이 무한히 동작할 수 있다. 동기성 데이터 흐름 프로그램은 디지털 신호 처리, 오디오, 비디오, 그래픽 커널, 네트워크, 데이터 암호화 및 해독과 같은 분야에서 널리 활용되고 있다.

스트림 프로그램 최적화의 주된 목표는 단위시간 당 데이터 처리량의 증진이다. 스트림 프로그래밍 언어의 성능을 향상시키기 위한 다양한 컴파일러 기법과 최적화 방법이 선행연구되었지만, 현대의 병렬 컴퓨터 구조 본래의 연산 능력을 오롯이 활용하기에는 개발된 기법들의 최적화도가 충분하지 못했다. 본 논문은 고성능 동기성 데이터 흐름 프로그래밍 언어를 개발함에 있어서 가장 큰 장애물은 데이터 채널의 생산자와 소비자 모델을 구현하기 위해 활용된 선입선출(FIFO)식 의미론임을 보인다. 선입선출식 의미론은 데이터를 접근함에 있어 무시할 수 없는 부하를 일으킬 뿐만 아니라, 선입선출식 의미론으로 인해 간접화된 데이터 접근방식은 데이터간의 의존도를 모호하게 하여 결과적으로 컴파일러 최적화 기법들의 효용성을 저하시킨다.

본 논문은 동기성 데이터 흐름 프로그램의 성능을 향상시키기 위해 다음과 같은 세가지 새로운 최적화 방식을 제시한다: (1) 선입선출 의미론을 직접적인 데이터 접근방식으로 대체할 새로운 중간 표현형(intermediate representation)인 LaminarIR을

정의하고 선입선출 의미론을 사용하는 프로그램을 LaminarIR로 변환하는 제반 이론을 확립한다. (2) 정적 프로그램 분석 기법을 기반으로 선입선출 의미론을 LaminarIR로 변환하기 위한 온전(sound)하고 의미 보존적인 프로그램 변환 기법을 지원하는 컴파일러 프레임워크를 개발한다. (3) 통신부하를 고려한 프로그램 편성 기법을 설계하고 그 기법을 LaminarIR에 적용하여 LaminarIR이 멀티코어 컴퓨터구조에서 활용될 수 있도록 한다. 위의 기법들이 구현된 LaminarIR 프레임워크와 성능실험을 위해 활용된 벤치마크는 스트림 프로그래밍 언어의 성능을 향상시키기 위한 후속연구를 용이하게 하기 위해 오픈소스화되었다.

핵심되는 말 : 멀티코어 컴퓨터구조, 스트림 프로그래밍 언어, 동기성 데이터 흐름, 컴파일러 최적화, 성능 분석, 정적 프로그램 분석

Acknowledgments

Yousun(1) Yousun(1)

NAME

Yousun - the command-line interface to develop Yousun Ko into a holder of a doctoral degree

SYNTAX

Yousun [<u>options</u>]

DESCRIPTION

This interface enables the thesis author to interact with great people around her, who have contributed to expand her perceptions of research and society.

OPTIONS

--advisors <u>**advisor**</u>

Consult with advisor(s). The advisors are:

Professor Bernd Burgstaller:

A scholar who astonishes all the time by his depth of intelligence, enthusiasm, and patience. He also knows how to hit the rock with a shovel.

Professor Bernhard Scholz:

A scholar of great insights and talent, who bridges the gap between theory and practice. He knows how to brainstorm in the middle of stagnating ideas.

--committee-members

Consult with committee members Professor Sang-hyun Park, Professor Yo-Sub Han and Professor Kyoungwoo Lee, for insightful comments and guidances which enrich research perspectives.

--colleagues

Meet lab members from the ELC Lab., Minyoung, Wasuwee, Edy, Shinhyung, Hyoseok, Yoojin, Yong-hyun and Seongho, to develop reasoning ability and tackle research issues together. Share joys and sorrows of night shifts.

--friends

Meet dear friends, Soojin, Woojin, Jongchoel, Hoyoung and Jeongho, for considerate support and intense discussions. Meeting them is very effective, especially to relieve tension from pursuing research.

--family

Meet parents and Joohyoung who are always supportive and instructive, to seek questions not only in research but also in life. They provide a space to breathe, think, and eventually come up with inspiring ideas.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

The increasing demand for high performance with big data processing and distributed computing demands novel parallel programming paradigms. Sequential programming languages provide insufficient parallel hardware abstractions, which greatly hampers performance and portability of software on multi/many-core architectures. With sequential programming languages, it is left to the programmer to identify parallelism in programs, which is tedious and error-prone. In contrast, the stream programming paradigm provides an abundance of parallelism already in the source-code, through its programming abstraction of independent actors that communicate via FIFO data channels.

1.1 Stream Programming Paradigm

The stream programming paradigm facilitates application domains characterized by regular sequences of data. These characteristics occur with digital signal processing, audio, video, graphics kernels, networking, and data encryption and decryption algorithms. In the stream programming model, computations are expressed through a set of actors that are connected by FIFO data channels. An actor is said to be *fireable*, if there are sufficient *tokens* on its incoming data channels. An actor firing removes tokens from the incoming data channels, executes the operation of the actor, and places the computed tokens in the outgoing data channels [42]. Except the data-dependencies between a producer and a consumer, actors are independent of each other, e.g., actors are not allowed to communicate via shared variables. Because each actor contains its own program counter, actors can execute in parallel. The actor programming paradigm, thus, enforces a programming style where ample parallelism is already expressed in the source-code.

Examples of stream programming languages and environments that adopts the stream programming paradigm include Baker [19], Brook [17], Cg [55], CQL [7], Lime [8], StreamFlex [71], StreamIt [77], ΣC [32], OpenCL [72], and SPUR [85]. Systems based on the streaming model are acknowledged by various previous researches such as Borealis [4], Flextream [37], and DANBI [58].

1.2 Thesis Contributions

A major challenge with stream programs is to fully harvest the raw computing power of contemporary parallel architectures. This problem is paraphrased by the term "*The Parallel Programming Gap*", expressing the steady increase in the number of cores over time, and the slow adoption of new programming models to effectively utilize the computational power of parallel cores. Current compilers and programming language implementations for stream programming languages are in their infancy and cannot leverage the available parallelism to utilize the underlying hardware to the maximum. Until now, the traditional focus of stream programming language compilers has been to leverage the available parallelism during compilation, auto-tuning and run-time adaptation by exploiting underlying stream graph structure with an objective to maximize data throughput [15, 26, 37, 46]. However, the optimization opportunities with actual data transfers between communicating actors have been neglected from the considerations. Such optimization opportunities are particularly imminent in structured stream programs that re-distribute data via structural components, such as split-joins, pipelines and feedback loops.

In this thesis, we capitalize on the fact that the stream programming model is designed to process regular data streams, which means that communication overhead between actors is a major contributing factor to the performance of stream programs. In particular, even a small inefficiency in the data-item access model will accumulate quickly and impact overall performance. Thus, we propose three optimization techniques which tackle performance issues related to data communication to improve the performance of stream programs on multicore architectures.

In this work, we employ the semantics of *Synchronous Data Flow* (SDF) [11], which restricts the general Kahn's process [43]. In SDF, an actor consumes and produces a fixed number of tokens when it is fired. The fixed number of tokens consumed and produced in an actor firing permits the computation of a *static finite periodic schedule* [11], simplifying the processing model. Further background knowledge on SDF semantics are discussed in Chapter 2.

Contribution 1: LaminarIR—a Novel IR for Stream Programs

First, we present a new intermediate representation (IR) called *LaminarIR*, to replace FIFO (first-in, first-out) semantics with direct memory accesses and its underlying theory for program transformation. Stream programming languages employ FIFO 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 data-flow implicit. For a compiler, indirect token-access obscures data-dependencies, which renders standard optimizations ineffective and impacts stream program performance negatively. Thus 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 implemented a LaminarIR compilation framework, and 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.

The goal of the new IR 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 also propose a lowering transformation that converts a stream program to *LaminarIR*. 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 [77]). 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.

Contribution 2: Specification-driven Static Analysis for Actor Transformations

The local direct access transformation is the former transformation step of the two steps in the LaminarIR program transformation, which replaces queue operations with static local positions in the queue for each actor. To perform a one-to-one mapping of a queue statement and a queue position, a control-flow update of actor code may be inevitable, because a given statement can be mapped to multiple queue positions, e.g. queue operations in a loop. Therefore, a loop-bound analysis is essential to perform our local direct access transformation. To the courtesy of the static characteristic of SDF programs, the control-flow of all but five actor definitions in the StreamIt benchmark suite [76] are data independent, and partial constant propagation of static variables are sufficient to determine loop bounds statically, so as the queue positions denoted by the queue operations in the loops. Yet, data dependent control-flow is commonplace in real-world applications. Thus we propose a sound static program analysis technique that exploits actor specifications to resolve data dependent control-flow. By abstract interpretation [22, 81], an integer interval is employed to abstract the space of possible data rates for each statement of an actor code. A narrowing operator is provided to enhance the analysis quality and the narrowing condition is driven by the actor specification.

Based on the static analysis, standard loop unrolling and trace partitioning techniques [10, 56, 68] are applied on the AST of a program to flatten control-flow paths.

Thus more queue operations can be mapped with a static queue position as a result. Static analysis is performed iteratively until an internal inspector decides an input AST is fully transformed and no further transformations are possible.

Contribution 3: Communication-aware Orchestration

In the third contribution, we design a communication-aware orchestration algorithm which will be a foundation to expand the target hardware of LaminarIR to parallel architectures. Load-balancing actors among available processor cores is a major challenge. The mapping of stream graphs on processor cores (known as orchestration) has received a lot of attention already in the literature [18, 19, 26, 27, 31, 33, 44, 47, 75, 78, 79, 80, 84]. The traditional load-balancing problem that only considers computational overheads are known to be NP-hard. The LaminarIR model requires an a further advanced load-balancing technique than the traditional approach. The model weighs benefits from data communication cost elimination by LaminarIR and benefits from parallel execution despite of accompanying communication overhead. The gains obtained from parallel execution are easily overshadowed by communication overhead. The workload of a processor core comprises both the execution time of actors and the communication overhead of data channels. In [27], it was shown that the makespan can worsen by up to 346% if communication costs of data channels are not considered. A unified integer linear programming (ILP) formulation has been presented by Farhad et al. [27], which considers communication costs of data channels on cache-coherent multicore architectures. However, as the number of processor cores increases, the ILP program quickly becomes intractable and does not present a practical solution for the actor placement problem. We present an approximation algorithm for placing stream programs on multicores that takes communication costs of data channels into account. The algorithm balances the workload of processor cores such that the *makespan* becomes minimal. The theory of approximation algorithms [82] was used to design our actor placement. Approximation algorithms are an active field of research in optimization theory and theoretical computer science. Unlike load-balancing heuristics, our approximation algorithm runs in polynomial time with solutions within a factor of $\log_2 n$ of the optimal solution, where n is the number of actors in a stream program. Also, our approximation algorithms provide solutions whose value is within a factor of the optimal solution.

1.3 Thesis Organization

This thesis is organized as follows: in Chapter 2, we present background material. Chapter 3 introduces the LaminarIR compiler framework and provides an overview of its three main research contributions. Chapter 4 describes theoretical background of LaminarIR, including proofs of semantic preserving transformations from FIFO semantics to LaminarIR. Chapter 5 introduces static program analyses deployed with the local direct access transformation of LaminarIR. In Chapter 6, we present the approximation algorithm that exhibits a polynomial run-time, and prove its correctness and approximation bounds. We survey related work in Chapter 7 and draw our conclusions in Chapter 8.

Chapter 2

Background

This chapter presents the background of SDF programs including graph theory, parallel scheduling and its formulation, and code generation to maximize data throughput of SDF programs.

2.1 Synchronous Dataflow (SDF)

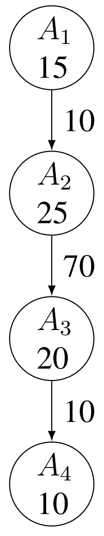
An SDF graph is a data-flow graph, which statically specifies the number of samples produced or consumed by each node on each invocation *a priori* [11]. An SDF graph is represented by a tuple (V, E) where V and E are finite sets of nodes and edges, respectively. An edge $e \in E$ is a tuple of two nodes (u, v), which indicates a data channel from node u to node v. We use n for the size of set V, and m for the size of set E.

Each node $v \in V$ has two properties, *preds* (v) and *succs* (v) which indicate the sets of predecessors of v and successors of v, respectively.

With graph $G = (V, E)$, we define *src* (G) as a set of any source node $v \in V$ which has no $u \in V$ that satisfies $(u, v) \in E$. Similarly, *sink* (G) is a set of any sink node $v \in V$ which has no $w \in V$ that satisfies $(v, w) \in E$. We assume that an SDF graph has a single source node and a single sink node. An SDF graph has three vectors **cns**, **prd** and **del** whose elements correspond to edges in the SDF graph [59]. The consumption rate *cns* $((u, v))$ is a positive integer that represents the number of tokens associated with edge (u, v) by each invocation of node v. The production rate *prd* $((u, v))$ is a positive

```
void->void pipeline Program() {
  add A1();
  add A2();
  add A3();
  add A4();
}
void->int filter A1() {
  int x;
  init {x=0;}
  work push 1 {push (x++);}
}
int->int filter A2() {
  work push 6 pop 1 {
      pop();
      // do some work
      push(); // 6 times
  }
}
int->int filter A3() {
  work push 1 pop 6 {
      pop();  // 6 times
      // do some work
      push();
  }
}
int->void filter A4() {
  work pop 1 {print(pop());}
}
```

(a)



(b)

Figure 2.1: (a) Actor source code   (b) Corresponding stream graph.

integer that indicates the number of tokens produced on edge (u, v) by each invocation of node u. By *del* $((u, v))$, we present a non-negative integer which indicates number of initial tokens on edge (u, v).

The Synchronous data-flow (SDF) programming model [11] is frequently adopted to the design of stream programming languages for its convenience to abstract stream programs into a set of nodes and channels in between. SDF defines data-flow graph that restricts the general Kahn's process [43] by statically specifying the number of data tokens produced or consumed by each actor on each invocation a priori. The fixed number of tokens consumed and produced in an actor firing permits the computation of a *static finite periodic schedule* making the processing model simple.

An example program of SDF semantics is depicted in Figure 2.1.

2.2 Scheduling of SDF programs

Significance of scheduling for the throughput of SDF graphs has been recognized from the early stage of the SDF paradigm. Murthy et al. [61] introduced Single Appearance Scheduling (SAS), where each actor only appears once in the schedule to avoid code-size explosion. Compared to a schedule where an actor may appear in multiple parts, SAS comes at the cost of larger buffer sizes. For a given SDF graph more than one SAS schedule may exist. Ritz et al. [65] introduced two variants of SAS, Single Appearance Single Activation Schedules (SASASs) and Single Appearance Minimum Activation Schedules (SAMASs). SASAS was designed for minimum program memory usage and SAMAS pursues minimum context switch overhead. As an opposite scheduling to SASAS, push scheduling executes downmost node which has sufficient number of available tokens to invoke even once. Push scheduling results in the minimal buffer size at the expense of code size. Karczmarek et al. [44] proposed a new scheduling algorithm, phased minimum latency scheduling, which is an alternation between SAS.

2.2.1 Steady-state Scheduling

A *periodic static* schedule [49] consists a finite sequence of node invocations; the periodic schedule is computed at compile time, invokes each node 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 schedule can be executed ad-infinitum without exhausting memory, and we refer to the state before and after the execution of a periodic schedule as the *steady-state*.

A periodic schedule has two properties denoted by two positive integer vectors. First vector is $\mathbf{r} \in \mathbb{N}^n$ called *repetition vector* [49] whose elements correspond to nodes in the SDF graph. Element $r(v)$ for all $v \in V$ is equal to the number of occurrences of node v in the periodic schedule. Because every node needs to be invoked at least once in the schedule, $r(v) \geq 1, \forall v \in V$. To run the nodes of an SDF graph in parallel, a sufficient number of initial tokens must exist on all edges $(u, v) \in E$. This number of pre-loaded tokens is denoted by vector $\mathbf{s} \in \mathbb{N}^m$, which is the second property of a periodic schedule.

Definition 1. *Given an SDF graph G for which a periodic schedule exists, we call this graph* parallel-executable *if the following condition holds for all edges* $(u, v) \in E$*,*

$$\mathit{s}\left((u,v)\right) \geq \mathit{r}\left(v\right) \times \mathit{cns}\left((u,v)\right).$$

To run a parallel periodic schedule ad-infinitum without exhausting memory, the following *balance equation* must hold for all edges $(u, v) \in E$ of the SDF graph.

Definition 2. *For each edge* (u, v) *of a parallel-executable graph, the balance equation is denoted by*

$$\mathit{r}\left(u\right) = \mathit{r}\left(v\right) \times \frac{\mathit{cns}\left((u,v)\right)}{\mathit{prd}\left((u,v)\right)}$$

for any edge $(u, v) \in E$*.*

2.2.2 Pre-steady-state Scheduling

The initial idea of a pre-steady-state schedule has been borrowed from software-pipelining. The prolog in software-pipelining denotes the sequence of instructions used to fill a software-pipelined loop [5]. Gordon et al. [31] introduced a *loop prologue*, which serves to advance each node to a different iteration of the stream graph.

StreamIt [77], which is one of the stream programming languages, allows a node to have a *prework* stage which is executed once before the periodic static schedule. It permits programmers to insert tokens on particular edges of the SDF graph. A node's production and consumption rates may differ from its steady state. In addition, nodes may use a *peek* operation where tokens are read in a non-consuming way. Nodes may peek more tokens than they consume, which must be considered in a pre-steady-state schedule. To account for prework stages and peek operations, an *initialization schedule* has been proposed in [30, 44]. Therefore, the pre-steady-state phase of StreamIt programs is a composition of an initialization schedule followed by a pre-steady-state schedule.

To run a node as part of a static periodic schedule, each incoming edge has to contain sufficient number of tokens. If this condition holds for all nodes, then all nodes can execute completely independent to each other. The easiest way to fill each edges with a sufficient number of tokens is to fill edges with zero data tokens [51]. With this approach, the first few results of the SDF program will have undefined values. But based on the ideal assumption that the SDF program runs infinitely, an initial small and constant number of undefined values would be tolerable for SDF program execution. However, filling edges with zero data tokens will not execute correctly if an actor contains input-data-dependent state variables. For such actors called *stateful actors*, pre-steady-state schedule prior to the steady-state schedule are necessary to advance the SDF graph into a state with a sufficient number of tokens for a periodic static schedule.

2.2.3 Steady-state Scheduling of SDF Graphs with Cycles

Existence of cycles in an SDF graph makes the computation of a schedule not trivial. If we define C as a set of at least two edges which forms a single cycle in an SDF graph G, Reiter showed in [64] that the SDF graph is free of deadlock if and only if every cycle C in the SDF graph satisfies

$$\sum_{(u,v)\in C} \mathit{del}\left((u,v)\right) \geq \min_{(u,v)\in C} \left(\mathit{cns}\left((u,v)\right) \times r\left(v\right)\right)^{*}. \tag{2.1}$$

Even though the SDF graph is deadlock free, it does not guarantee that a cycle can execute in parallel.

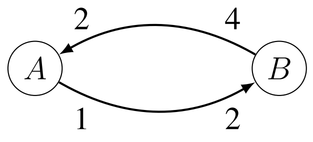


Figure 2.2: An SDF with a cycle.

Figure 2.2 shows an SDF graph with a cycle. Numbers written on each edge indicates production and consumption rate of the edge e.g., for the edge (A, B), $\mathit{cns}\left((A, B)\right) = 2$

*Reiter considers homogeneous SDF (HSDF) graphs in [64]: HSDF graphs are SDF graphs which consume and produce only a single token for each invocation of nodes. Thus, the original condition to be a deadlock free SDF graph written in [64] is $\sum_{(u,v)\in C} \mathit{del}\left((u,v)\right) > 0$

and *prd* $((A, B)) = 1$. If *del* $((A, B)) =$ *del* $((B, A)) = 0$, then there is no schedule to run the SDF graph as none of A or B has available tokens to execute, ending up in deadlock. If edge (A, B) satisfies Equation (2.1), for example *del* $((A, B)) = 2$, then valid periodic schedules exist such as BAA. However, these schedules are only valid when executed on a single processor, and no valid parallel periodic schedule exists unless all edges are filled with sufficient number of tokens to run the parallel periodic schedule. Thus, every cycle C in an SDF graph must satisfy the following condition if and only if the SDF graph is deadlock free for parallel execution.

$$\textit{del}\,((u, v)) \geq (\textit{cns}\,((u, v)) \times \textit{r}\,(v)), \forall (u, v) \in C. \tag{2.2}$$

With the example in the Figure 2.2, the condition to be deadlock free for parallel execution is *del* $(B, A) = 4$ and *del* $((A, B)) = 2$.

In the thesis, we consider two ways to ensure that the SDF graph is deadlock free for parallel execution. First is to urge users to program SDF graphs which at least satisfies the Equation (2.1) in compile time, and introduces an algorithm to transform the SDF graph into acyclic SDF graph by condensing nodes in a cycle in a new merged node. This transformation does not change any semantic of an SDF program, but only forces to execute all nodes of a cycle on a processor. Basic algorithms to transform a cycle into a new merged node is introduced by Bhattacharya et al. [11]. The other approach is to guide users to provide sufficient delays to satisfy Equation (2.2) in compile-time. These two approaches are realistic as SDF programs are static already in compile-time.

2.3 SDF Code Optimization

Although buffering techniques and scheduling are inter-dependent, there is a large body of work focusing mainly on buffering techniques. The common idea is to put adjacent nodes to different iterations to run them independently during each subsequent iteration of the SDF graphs. The buffer space must be provided to store at least one steady state of tokens between each pair of adjacent nodes [31, 60]. Murthy et al. [60] introduced a buffering technique that retains separated buffers for each input and output buffers of an edge but which shares the memory space as much as possible. This approach could further minimize buffer sizes compared to the results of the previous work of the time [66, 74]. But this work did not take the performance of the SDF programs into account, and it required the programmer to supply annotations or static program analysis techniques which can comprehend semantics of each nodes' computations. Furthermore, their buffer design is coarse-grained which considers number of tokens produced and consumed in a steady-state iterations but does not consider individual tokens.

Regarding techniques to manage buffers, a straight-forward and the most popular approach to implement communication of two actors is to use First-In-First-Out (FIFO) queues. But queues are costly, so Lee et al. [51] introduced *static buffering* which reads its input directly from specified memory locations and writes its outputs directly to specified memory locations within buffer. This method was applied to *Gabriel* [13], a design environment for digital signal processing, but the idea was introduced briefly only to be applied on the buffers connect two nodes in a same processor.

Chapter 3

Overview: LaminarIR Compiler Framework

This chapter introduces each step of LaminarIR compiler framework and describes its software engineering aspects. LaminarIR compiler framework is a source to source compiler framework which accepts a structured stream program e.g., StreamIt, and generates C code that employs FIFO queues or direct token accesses as depicted in Figure 3.1. The LaminarIR compiler frontend parses structured stream programs and lowers them to LaminarIR by applying local and global direct access transformations. A *static analysis* pass is employed on the Abstract Syntax Tree (AST) in the frontend phase to enhance replacement of queue operations with local named tokens. An *orchestration* pass provides actor allocation information on processor(s) which is incorporated into LaminarIR after the global direct access transformation. The code generator subsequently generates C code from LaminarIR. To enable rapid prototyping, LaminarIR compiler framework employs object-oriented design including the *strategy pattern* [28] for parser, scheduler and code generator. For example, as shown in Figure 3.1, the LaminarIR compiler framework provides a code generator interface for two different implementations of the code generator, i.e., FIFO queues and Direct Access. The intention of implementing own FIFO queue backend is to isolate effects of LaminarIR as the independent variable in our experimental evaluation from the input language specific optimizations. For example, the StreamIt compiler applies its own graph transformation methods on input programs, e.g., combining multiple actors into a single aggregate actor (fusion) or by instantiating

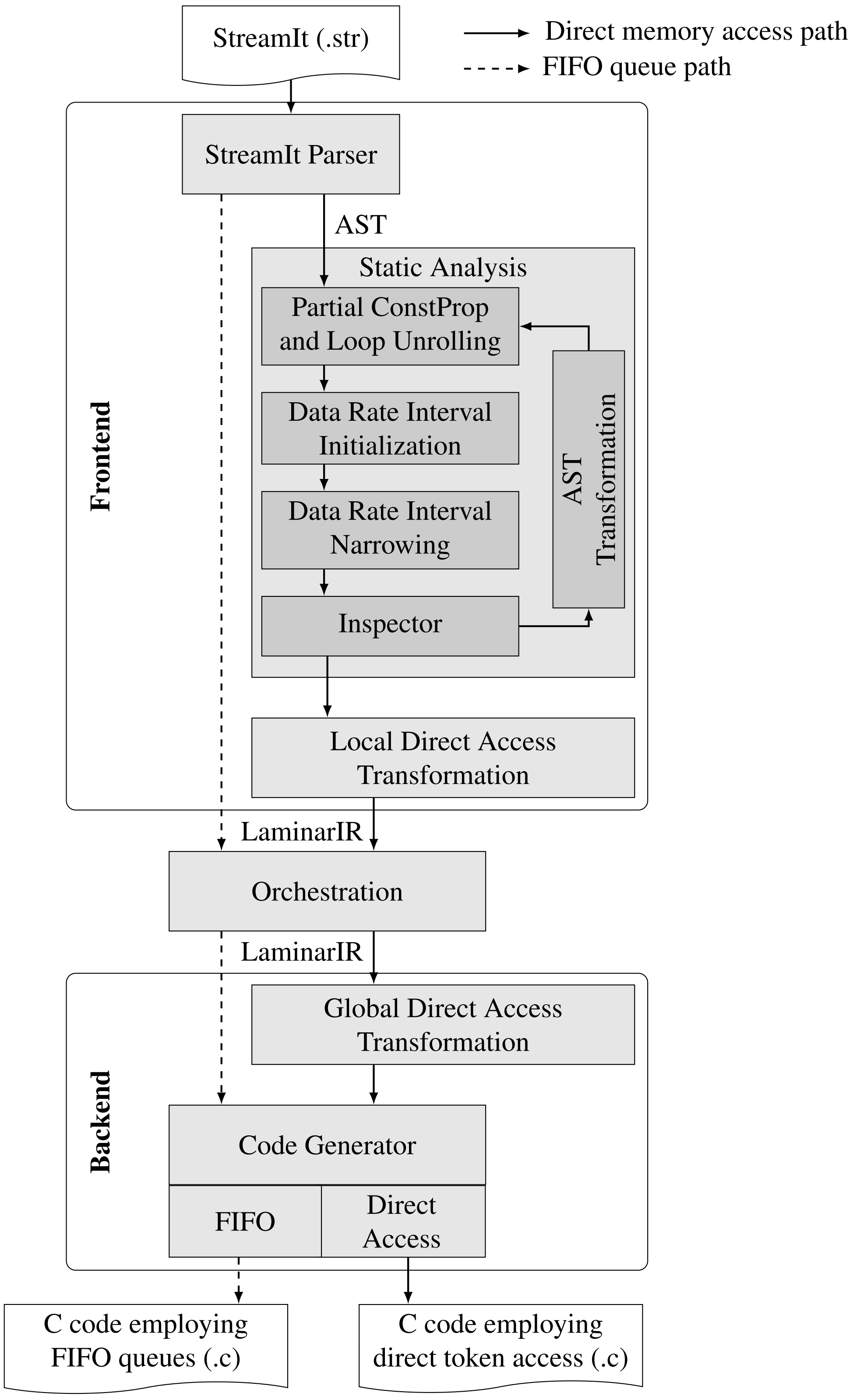

Direct memory access path
FIFO queue path
StreamIt (.str)
StreamIt Parser
AST
Static Analysis
Partial ConstProp and Loop Unrolling
Data Rate Interval Initialization
Data Rate Interval Narrowing
Inspector
AST Transformation
Frontend
Local Direct Access Transformation
LaminarIR
Orchestration
LaminarIR
Global Direct Access Transformation
Backend
Code Generator
FIFO
Direct Access
C code employing FIFO queues (.c)
C code employing direct token access (.c)

Figure 3.1: LaminarIR compiler framework

an actor multiple times (fission) and wrapping the instances by a round-robin split-join combination to achieve data parallelism [30]. Similarly, scheduler interface can be implemented by various actor allocation algorithms, and parser interface can support any structured stream programming languages, independent to the functionalities of other passes.

The LaminarIR framework and the benchmarks that we used with the experimental evaluation are available online [1].

3.1 Frontend

The frontend of the LaminarIR compiler framework is composed of three major parts: (1) A parser for the input language to generate ASTs, (2) a local direct access transformation that replaces FIFO queues with local named tokens from ASTs of a stream program and generates LaminarIR code, and (3) a static program analyzer that performs AST transformations to foster replacement of FIFO queues with local named tokens in the local direct access transformation.

Parser The parser of the frontend comes with the input programming language that the LaminarIR compiler framework supports. For the StreamIt parser, the ANTLR parser generator is adopted from the StreamIt compiler to accept StreamIt programs as input programs. In addition to AST generation, the parser generates stream graph information, such as actor instances and topology of actor instances. For the sake of programmability, structured stream programs facilitate constant stream parameters to scale each component. For example, the StreamIt parser generates a java bytecode that evaluates constant stream parameters and finally results an end stream graph of a StreamIt program. Topology of actor instances in the stream graph is an essential information to lower the structured stream programs into LaminarIR. The frontend may be expanded to support other stream languages. For a new language, a parser is required that generates an AST and a stream graph from the input code.

A FIFO queue path, a comparison group to LaminarIR, bypasses the static program analyzer pass and the local direct access transformation pass to generate LaminarIR code with FIFO queues directly from parser.

Static program analyzer The generated AST is passed on to the static program analyzer. The static analyzer is not technically mandatory to perform the local direct access transformation path generating LaminarIR code, and can be bypassed by letting the inspector exit the static analyzer pass without visiting the AST transformation pass. However, the quality of local direct access transformation is heavily dependent on the static analysis pass. The local direct access transformation replaces a queue operation with a direct access token when a queue operation is statically mapped to a queue position, otherwise the local direct access transformation path will keep queue operations in form of local array accesses. Multiple mapping of queue positions in a queue operation happens when the queue operation is invoked in loops, and the static analyzer employs static specification of actors to resolve the loop bounds of loops that contain queue operations.

When loop bounds are data independent, partial constant propagation of constant stream parameters derived from the parser is sufficient to resolve loop bounds in most of cases. On the other hand, data dependent loop bounds which are more frequently used in real-world applications, cannot be resolved during compile time in general. However, in our static analysis, we resolve the data dependent loop bounds by using abstract interpretation [22, 81] to denote the range of possible loop bounds, and narrow the range of possible loop bounds by the static data rates given by the actor specification. Standard loop unrolling and trace partitioning [10, 56, 68] are performed on AST to improve the analysis quality and the static analysis is revisited until the inspector decides to stop. Because loop unrolling combined with trace partitioning may explode code size, the inspector exams feasibility of the AST transformation in advance. Inspector also checks soundness of the analysis. Chapter 5 covers the static analysis pass in detail.

Local direct access transformation The local direct access transformation path replaces queue operations with local named tokens inside to actors. The local named tokens from producer and consumer are mapped to a global symbol later through the global direct access transformation path. Because most of complex mappings of queue operations to queue positions are resolved via the means of static program analysis, the replacement of queue operations to local direct access tokens are performed rather mechanically. Yet, two phase direct access transformations have its virtue on simplifying theoretical problems especially in case of transforming cyclic graphs or multi-phase scheduling. The design of LaminarIR including syntax is presented in Chapter 4.

3.2 Orchestration

Orchestration is an essential pass to expand the target hardware of LaminarIR to parallel architectures. Orchestration computes the mapping of stream graphs on multicores to achieve load-balancing among available processors. The traditional load-balancing problem that only considers computation overhead is known to be NP-hard, and yet it is hard to achieve the estimated load-balancedness by the traditional approach because the influence of data communication cost to the makespan is non-neglectable. In [27], it was shown that the makespan can worsen by up to 356% if communication cost of data channels are not considered. Even more, a further advanced load-balancing technique is essential to orchestrate LaminarIR because FIFO queues are required again for inter-processor communication where intra-processor communications are transformed into direct token access. Thus the orchestration should be able to weigh benefits from data communication cost elimination by LaminarIR and benefits from parallel execution despite of accompanying communication overhead.

As a foundation of the parallel LaminarIR, we introduce a new orchestration algorithm which considers communication costs of FIFO queues. The novelty of our orchestration algorithm is that we use an approximation algorithm to consider communication costs in a practical time and within a reasonable quality bound. Farhad et al. [27] showed a unified ILP formulation that considers communication costs of data channels, however, the ILP program quickly becomes intractable as the number of processor cores increases. The approximation algorithm for communication-cost-aware orchestration is presented in Chapter 6.

3.3 Backend

The backend of the LaminarIR compiler framework consists of two major passes, the global direct access transformation pass that maps the producer- and consumer-side of local named tokens, and the code generator that generates C target source code from LaminarIR. The underlying theory and the proof of our semantic-preserving transformation are presented in Chapter 4.

3.4 Run-time Support for Performance Evaluation

The LaminarIR compiler framework provides a run-time support system for precise and accurate performance measurement. For the processors that support CPU frequency scaling, scaling governors are employed to lock the clock frequency to the value stated in the hardware specification. In addition, affiliation of processes are finely controlled by the lightweight performance tool (LIKWID, [2]). For ARM architectures which LIKWID dost not support, we use `sched_setaffinity` function from the GNU C Library to dedicate a specific core to a thread. To observe effect of our approaches on microarchitectural CPU events that are highly CPU-specific [40], the LaminarIR compiler framework provides a run-time support system that collects notable hardware performance counters by aid of PAPI [16]. LaminarIR compiler framework uses LLVM compiler infrastructure for an in-depth analysis of the profitability of each optimization on the evaluated formats.

Chapter 4

LaminarIR

4.1 FIFO Queue Overhead with Stream Programs

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 [15, 26, 37, 46]. 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

```
void->void pipeline Prog{
 add A();
 add float->float splitjoin
 {
  split duplicate;
  add B();
  add C();
  join roundrobin;
 }
 add D();
}
void->float filter A(){
 work push 1
  {push(my.frand());}
}
float->float filter B(){
 work push 1 pop 2{
  push(pop()+pop()/2);}
}
float->float filter C() {
 work push 1 pop 2{
  push(sqrt(pop()*pop()));}
}
float->void filter D(){
 work pop 2{
  println(pop());
  println(pop());}
}
```

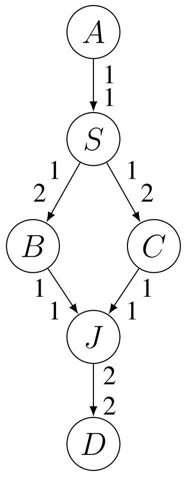
(a)

```
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);}
 }
}
```

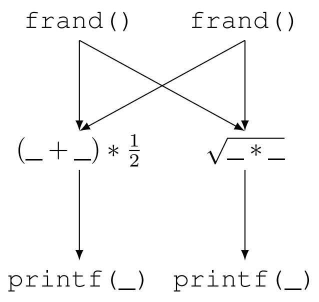
(b)

```
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);
 }
}
```

(c)

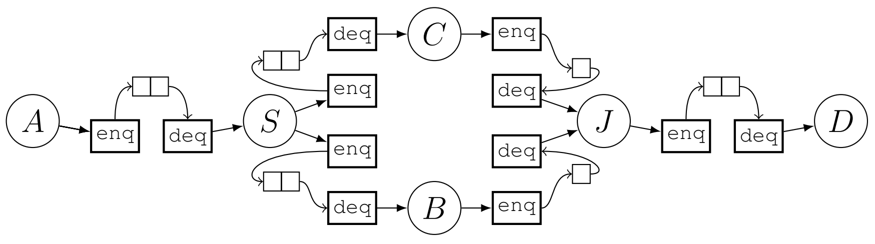


(d)

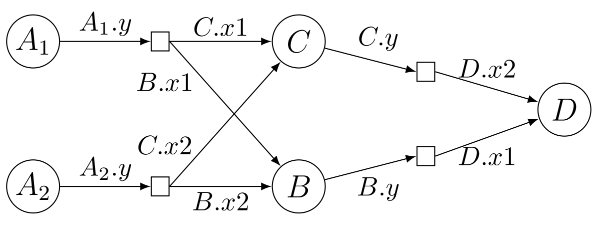


(e)

Figure 4.1: (a) An example SDF source code, (b) LaminarIR, (c) generated C code, (d) streamgraph, and (e) dataflow-graph for schedule $2A, 2S, B, C, J, D$,



(a)



(b)

Figure 4.2: Implementations of (a) FIFO queues vs. (b) LaminarIR

from one invocation to the next, limit the available parallelism as reported in the literature [26, 46]. 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 [77]). 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.

In this chapter, we discuss the following contributions of this thesis:

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

The remainder of this chapter is organized as follows. In Section 4.2 we present a motivating example for our framework. In Section 4.3 we describe the LaminarIR, the local direct access transformation with StreamIt as a case-study, and the LaminarIR code generator. Section 4.4 describes our global direct access transformation. Section 4.5 contains the experimental evaluation.

4.2 Motivating Example

Let us consider the example in Fig. 4.1a. 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 4.1d; 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 4.2a. 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 4.1a. 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 4.1e. 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 4.1b, 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 4.2b. Boxes (□) 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 data-flow graph in Figure 4.1e 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 4.5 applies this technique to entire StreamIt applications. Our proposed technique is applicable to static-rate sub-graphs of hybrid approaches like the one from [70]. 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 work, 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.

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

Figure 4.3: Abstract Syntax of LaminarIR. For brevity, constructs not relevant for buffer management have been omitted.

4.3 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.

LaminarIR's abstract syntax is summarized in Figure 4.3. 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 4.1b constitute the edge section of our motivating example's stream graph. An *edge* declaration consists of the token-type and

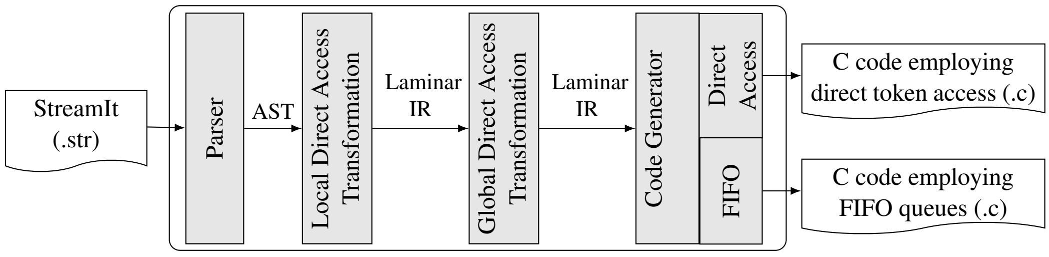

StreamIt (.str)
Parser
AST
Local Direct Access Transformation
Laminar IR
Global Direct Access Transformation
Laminar IR
Code Generator
Direct Access
FIFO
C code employing direct token access (.c)
C code employing FIFO queues (.c)

Figure 4.4: LaminarIR framework

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 4.1b 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 chapter 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 4.4, 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 data-flow between actor invocations for the compiler is fully exposed.

4.3.1 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, 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 [76].

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 4.5.

Our StreamIt-to-C source to source transformation framework is depicted in Figure 4.4. 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 [30]. 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 4.5 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].

4.4 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.

4.4.1 Background and Notation

An SDF program is a data-flow graph which statically specifies the number of data tokens produced or consumed by each actor on each invocation *a priori* [11]. 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 [49]; 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 $\mathit{head}_k : (t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_l) \mapsto (t_1, \ldots, t_k)$ extracts the first k elements from the list and function $\mathit{tail}_k : (t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_l) \mapsto (t_{k+1}, \ldots, t_l)$ 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., $\mathit{head}_{(k_1,\ldots,k_m)}(l_1, \ldots, l_m) = (\mathit{head}_{k_1}(l_1), \ldots, (\mathit{head}_{k_m}(l_m))$ to represent the state of an SDF program.

Boot Schedule The boot schedule of an SDF graph $G = (V, E)$ denotes a sequence of node invocations of any node $v \in V$ to arrange a sufficient number of tokens for all edges $e \in E$ according to Definition 1. We define a non-negative integer vector $\mathbf{i} \in \mathbb{N}_0^n$ called the *init vector* whose elements correspond to nodes in the SDF graph. Element $i\,(v)$ for all $v \in V$ indicates the number of occurrences of node v in the boot schedule.

The number of tokens that reside on edge (u, v) after booting can be written as $i\,(u) \times \mathit{prd}\,((u, v)) - i\,(v) \times \mathit{cns}\,((u, v))$. Therefore, we can derive Definition 3.

Definition 3. $s\,((u, v)) = i\,(u) \times \mathit{prd}\,((u, v)) - i\,(v) \times \mathit{cns}\,((u, v)),\ \forall (u, v) \in E$

To compute the vector $\mathbf{i}$ of a boot schedule, we introduce the *level* [*] of node, which indicates the depth of the node from the sink node. We define vector $\mathbf{l}$ whose elements correspond to the levels of node in the SDF graph.

$$l\,(v) = \begin{cases} 0 & \text{if } v \in \mathit{sink}(G), \\ \left(\max\limits_{w \in \mathit{succs}(v)} l\,(w) \right) + 1 & \text{otherwise.} \end{cases} \tag{4.1}$$

[*]In [49], Lee et al. defined the level of a given node is the longest number of invocations of a given node on a path from the node to the sink node of a graph.

Thus the number of invocations of node v for booting is

$$i\left(v\right) = l\left(v\right) \times r\left(v\right). \tag{4.2}$$

Lemma 1. *Equation (4.2) is equivalent to* $i\left(v\right) = \max\limits_{w \in succs(v)} \left(\left(i\left(w\right) + r\left(w\right)\right) \times \frac{cns((v,w))}{prd((v,w))}\right)$.

Proof.

$$\begin{aligned}
i\left(v\right) &= l\left(v\right) \times r\left(v\right) \\
&= \left(\max_{w \in succs(v)} \left(l\left(w\right)\right) + 1\right) \times r\left(v\right) && \text{by Equation (4.1)} \\
&= \left(\max_{w \in succs(v)} \left(l\left(w\right) + 1\right)\right) \times r\left(v\right) \\
&= \max_{w \in succs(v)} \left(\left(l\left(w\right) + 1\right) \times r\left(v\right)\right) \\
&= \max_{w \in succs(v)} \left(\left(l\left(w\right) + 1\right) \times r\left(w\right) \times \frac{cns\left(\left(v,w\right)\right)}{prd\left(\left(v,w\right)\right)}\right) && \text{by Definition (2)} \\
&= \max_{w \in succs(v)} \left(\left(l\left(w\right) \times r\left(w\right) + r\left(w\right)\right) \times \frac{cns\left(\left(v,w\right)\right)}{prd\left(\left(v,w\right)\right)}\right) \\
&= \max_{w \in succs(v)} \left(\left(i\left(w\right) + r\left(w\right)\right) \times \frac{cns\left(\left(v,w\right)\right)}{prd\left(\left(v,w\right)\right)}\right). && \text{by Equation (4.2)}
\end{aligned}$$

□

Lemma 2. *Based on Definition 1,* $i\left(u\right) \times prd\left(\left(u,v\right)\right) \geq i\left(v\right) \times cns\left(\left(u,v\right)\right)$ *holds.*

Proof.

$$s\left(\left(u,v\right)\right) = i\left(u\right) \times prd\left(\left(u,v\right)\right) - i\left(v\right) \times cns\left(\left(u,v\right)\right).$$

Thus,

$$\begin{aligned}
s\left(\left(u,v\right)\right) &\geq r\left(v\right) \times cns\left(\left(u,v\right)\right) \\
i\left(u\right) \times prd\left(\left(u,v\right)\right) - i\left(v\right) \times cns\left(\left(u,v\right)\right) &\geq r\left(v\right) \times cns\left(\left(u,v\right)\right) \\
i\left(u\right) \times prd\left(\left(u,v\right)\right) &\geq \left(i\left(v\right) + r\left(v\right)\right) \times cns\left(\left(u,v\right)\right) \\
&\geq i\left(v\right) \times cns\left(\left(u,v\right)\right).
\end{aligned}$$

□

We assume a single appearance schedule for booting. Therefore, the execution order of nodes is in topological order of the SDF graph G.

Lemma 3. *For an SDF graph $G = (V, E)$, the boot schedule expressed by the vector* ***i*** *satisfies $\mathit{s}((u,v)) \geq \mathit{r}(v) \times \mathit{cns}((u,v))$ for all $v \in V$ and predecessor $u \in \mathit{preds}(v)$.*

Proof. By structural induction on G in reverse topological order.

Base case: For the sink node $v \in \mathit{sink}(G)$, it holds that $\mathit{l}(v) = 0$ and $\mathit{i}(v) = 0$. Thus Definition (1) trivially holds.

Hypothesis: For any node $v \in V$ with level $\mathit{l}(v) = n$ and any successor $w \in \mathit{succs}(v)$, we assume $\mathit{s}((v,w)) \geq \mathit{r}(w) \times \mathit{cns}((u,w))$.

Then $\forall u \in \mathit{preds}(v)$,

$$\begin{aligned}
\mathit{l}(u) &\geq \mathit{l}(v) + 1 && \text{by Equation (4.1)}\\
\mathit{l}(u) \times \mathit{r}(v) \times \mathit{cns}((u,v)) &\geq (\mathit{l}(v)+1) \times \mathit{r}(v) \times \mathit{cns}((u,v)) &&\\
\mathit{l}(u) \times \mathit{r}(u) \times \mathit{prd}((u,v)) &\geq (\mathit{l}(v)+1) \times \mathit{r}(v) \times \mathit{cns}((u,v)) && \text{by Definition 2}\\
\mathit{l}(u) \times \mathit{r}(u) \times \mathit{prd}((u,v)) - \mathit{l}(v) \times \mathit{r}(v) \times \mathit{cns}((u,v)) & &&\\
&\geq \mathit{r}(v) \times \mathit{cns}((u,v)) &&\\
\mathit{i}(u) \times \mathit{prd}((u,v)) - \mathit{i}(v) \times \mathit{cns}((u,v)) & &&\\
&\geq \mathit{r}(v) \times \mathit{cns}((u,v)). && \text{by Equation (4.2)}
\end{aligned}$$

Thus $\mathit{s}((u,v)) \geq \mathit{r}(v) \times \mathit{cns}((u,v))$. □

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

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

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[†] 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 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.

We employ the function $\textit{head}_i(s_{i-1})$ that accesses the input tokens of the actor firing, i.e., $\textit{head}_{(c_{u_i}(1), c_{u_i}(2), \ldots, c_{u_i}(m))}(s_{i-1})$ where $c_{u_i}(j)$ denotes the consumption rate of actor u_i on channel j to extract the input tokens for the current actor firing f_i. 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 $\textit{tail}_i(s_{i-1}) = \textit{tail}_{(c_{u_i}(1), c_{u_i}(2), \ldots, c_{u_i}(m))}(s_{i-1})$, 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.

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

An implementation of the concrete semantics is shown in Listing 4.4. 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 4.4: Implementation of concrete semantics of SDF

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

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

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 5. *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*

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

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 5. *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

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

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 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[‡] 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:

$$\begin{array}{rrl} & \mathbf{v}_0 & \leftarrow \textit{delay} \\ \textit{loop}: & \textit{new}_1 & \leftarrow f_1(head_1(\mathbf{v}_0)) \\ & \textit{new}_2 & \leftarrow f_2(head_2(\mathbf{v}_1)) \\ & & \ldots \\ & \textit{new}_k & \leftarrow f_k(head_k(\mathbf{v}_{k-1})) \\ & \mathbf{v}_0 & \leftarrow \mathbf{v}_k \\ & \textbf{goto} & \textit{loop} \end{array}$$

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 re-distributes 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.

[‡]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 4.1: 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)

4.5 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 4.2. 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 syn-

[§]Shared by 2 cores.

Table 4.2: Hardware configuration

CPU	Intel i7-2600K	AMD Opteron 6378	ARM Cortex-A15	Intel Xeon Phi 3120A
Clock Freq.	3.4 GHz	2.4 GHz	1.7 GHz	1.1 GHz
I-Cache	32 kB	64 kB[§]	32 kB	32 kB, unified data and instruction cache
L1 D-Cache	32 kB	16 kB	32 kB	
OS	Ubuntu 12.04	CentOS 6.5	Linaro 13.08	Centos 6.5
Kernel ver.	3.2.0	2.6.32	3.11.0	2.6.32
C Compiler	LLVM/Clang 3.5.0			ICC 14.0.3

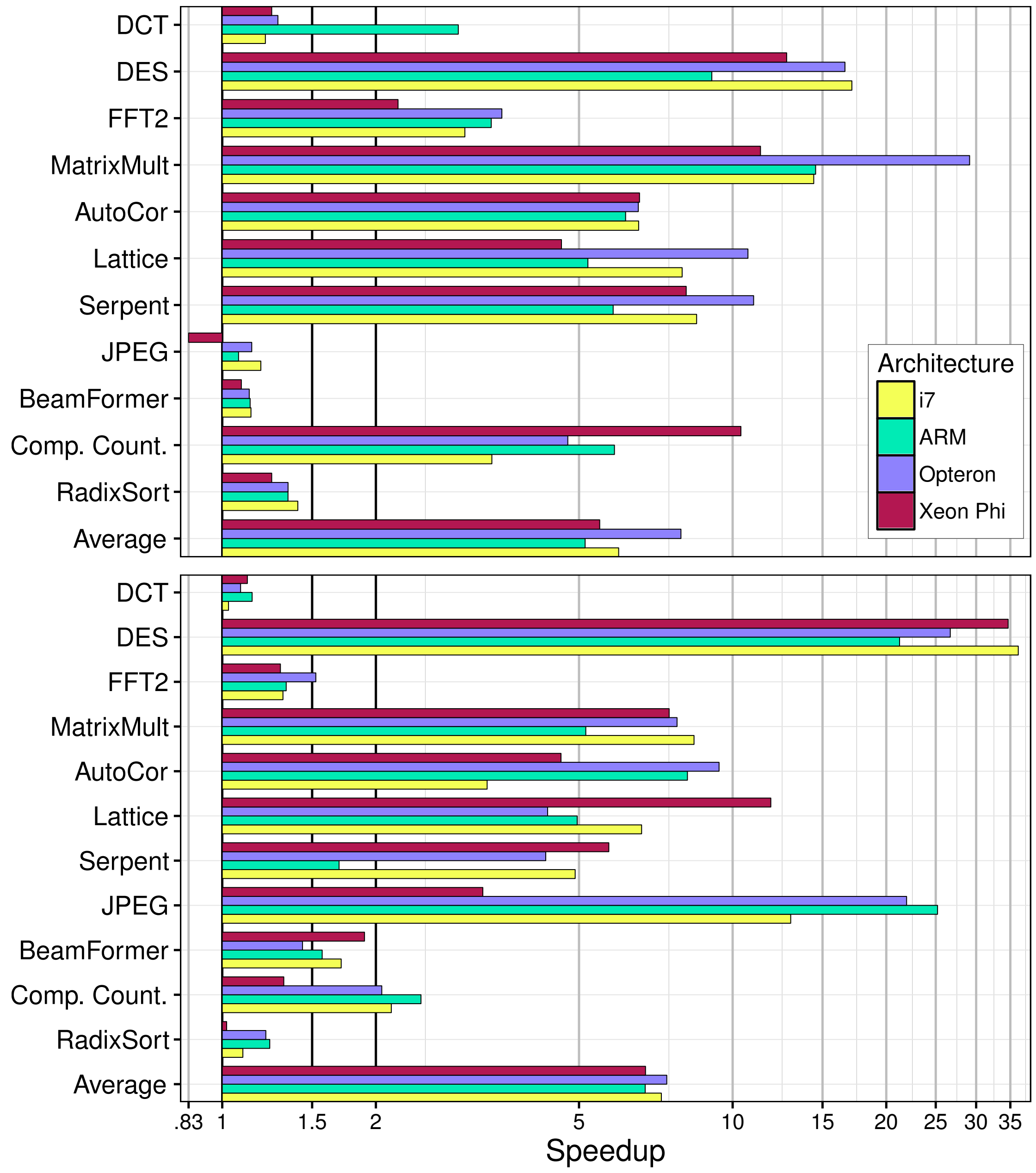
chronization and inter-processor communication overhead, which are orthogonal to our optimization. StreamIt's backend was thus used with the default setting that compiles for a uniprocessor. Measurement data was collected from hardware performance counters using PAPI [16]. Except with the Intel Xeon 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 4.2. As a result, the coefficient of variation of each measurements is close to 0%. Our list of representative benchmarks from the StreamIt benchmark suite [76] is stated in Table 4.1. Five benchmarks are from the StreamIt Core Benchmark Suite proposed by the MIT-StreamIt-group [30]. 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 4.1a). All evaluations shown in this chapter 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.

4.5.1 Performance

Figure 4.5 and Figure 4.6 show the speedups achieved by the LaminarIR direct access format over FIFO queues and StreamIt respectively. 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 4.5.2. 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.

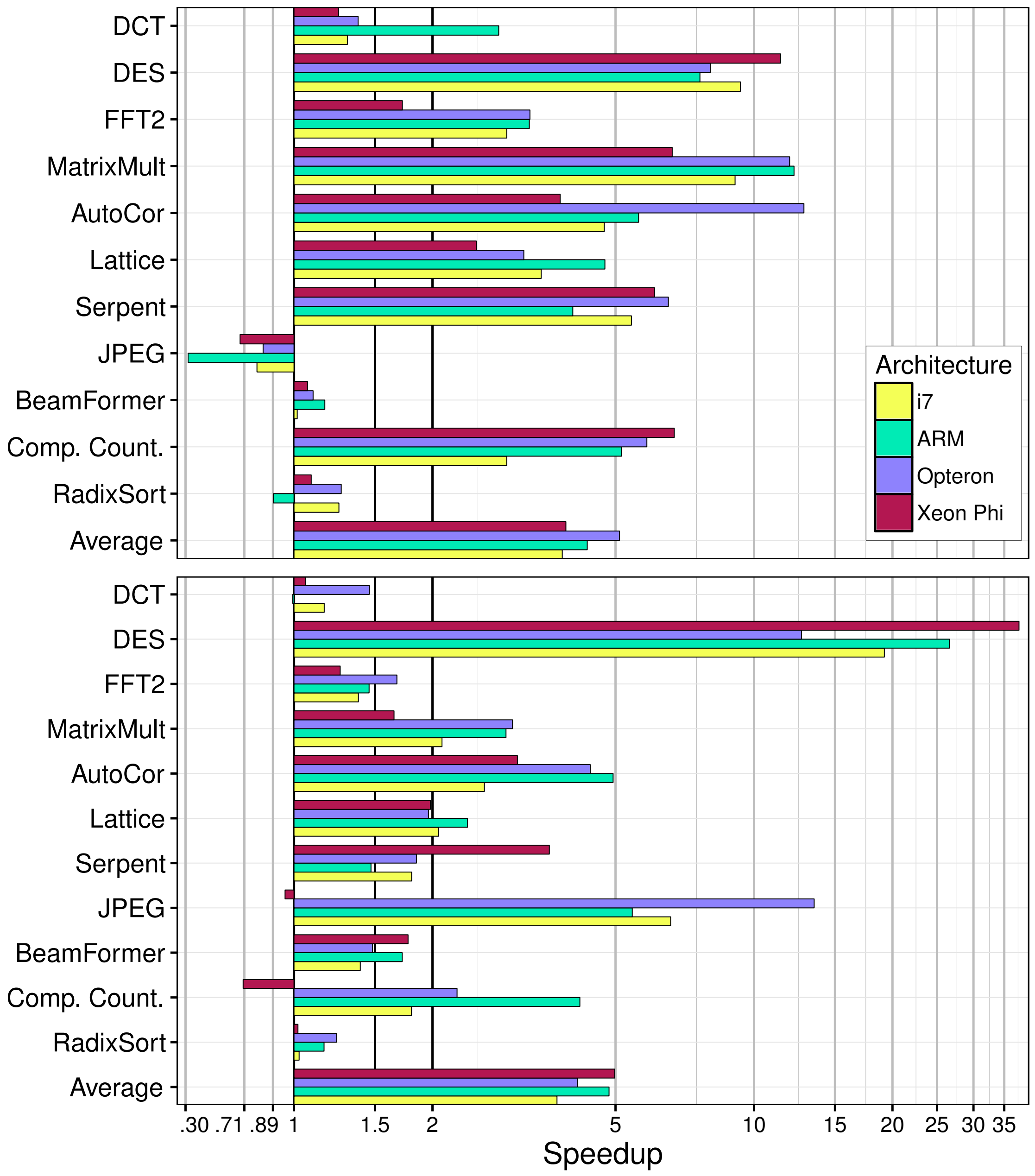
(a) without compiler optimization



(b) with compiler optimization

Figure 4.5: Speedup of LaminarIR vs. FIFO queues without compiler optimization (top), and with compiler optimization (bottom)

(a) without compiler optimization



(b) with compiler optimization

Figure 4.6: Speedup of LaminarIR vs. StreamIt without compiler optimization (top), and with compiler optimization (bottom)

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, 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 4.4 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 4.7 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.

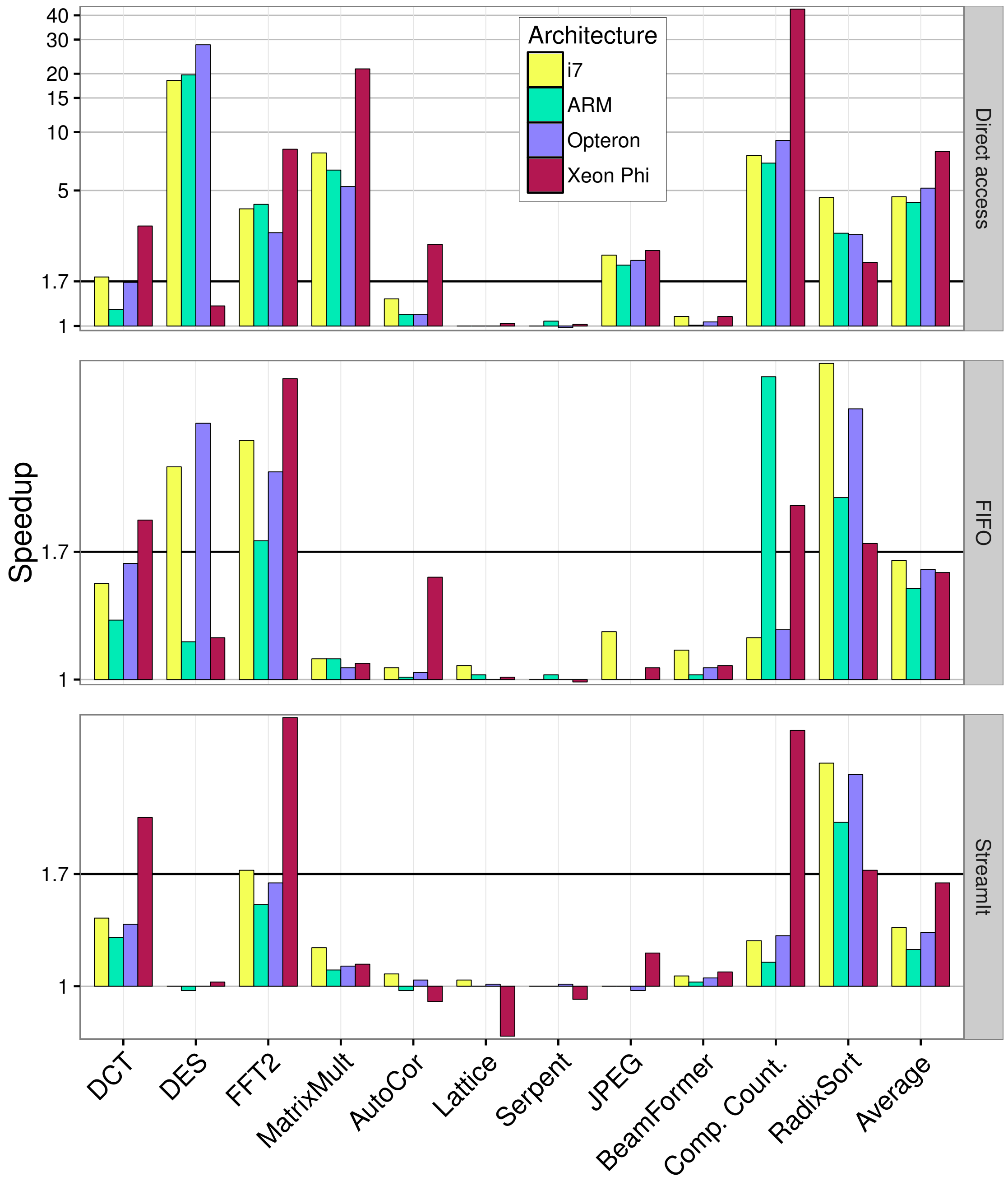


Figure 4.7: Effectiveness of the LaminarIR with compiler optimizations: static input data enables compilers to compute partial results already at compile-time, showing a 5.53x average speedup over dynamic input when using LaminarIR. This effect is observed to a much lesser degree with our FIFO queue implementation (1.56x average speedup) and StreamIt (1.34x average speedup).

Table 4.3: 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%

4.5.2 Communication Elimination

Table 4.3 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 with those two benchmarks (see Figure 4.5, Figure 4.6 and Table 4.4), which implies that it is more efficient even with the same amount of data communication.

Table 4.4: Improvements of the LaminarIR direct access format over FIFO queues and StreamIt on the Intel i7-2600K

Benchmarks	LaminarIR over FIFO queues			
	Inst.	Mem. Acc.		Energy
		Loads	Stores	Cons.
DCT	88.29%	95.78%	99.14%	107.65%
DES	7.00%	11.85%	4.08%	3.51%
FFT	60.71%	50.25%	43.26%	73.47%
MatrixMult	14.55%	14.99%	6.34%	11.50%
AutoCor	22.26%	25.16%	9.01%	27.47%
Lattice	11.12%	12.00%	9.64%	14.26%
Serpent	22.87%	34.60%	33.37%	20.39%
JPEG	6.04%	4.85%	5.03%	8.21%
BeamFormer	59.14%	59.28%	43.73%	58.15%
Comp. Count.	47.82%	34.92%	17.15%	46.49%
RadixSort	81.91%	61.04%	79.21%	98.64%
Average	38.34%	36.79%	31.81%	42.70%

Benchmarks	LaminarIR over StreamIt			
	Inst.	Mem. Acc.		Energy
		Loads	Stores	Cons.
DCT	65.11%	58.77%	97.87%	95.54%
DES	12.98%	14.09%	5.72%	6.44%
FFT	62.09%	55.49%	46.58%	72.53%
MatrixMult	29.59%	23.77%	12.13%	47.80%
AutoCor	40.30%	40.00%	26.31%	35.91%
Lattice	20.16%	15.88%	15.68%	47.38%
Serpent	45.88%	41.34%	42.94%	55.25%
JPEG	8.53%	6.30%	7.00%	16.34%
BeamFormer	59.44%	58.85%	44.22%	72.01%
Comp. Count.	57.24%	44.78%	35.08%	60.16%
RadixSort	78.95%	85.65%	77.81%	104.43%
Average	43.66%	40.45%	37.40%	55.80%

Table 4.5: 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%

4.5.3 LLVM Optimization Statistics

Table 4.5 shows the absolute number of variables promoted to SSA form with the LaminarIR direct access format (column “Abs.”), and proportional SSA promition rate with the LaminarIR direct access over the number of SSA variables 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 data-flow 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 [6]. Lower numbers of promoted SSA variables with the FIFO queues and StreamIt representations indicate the pres-

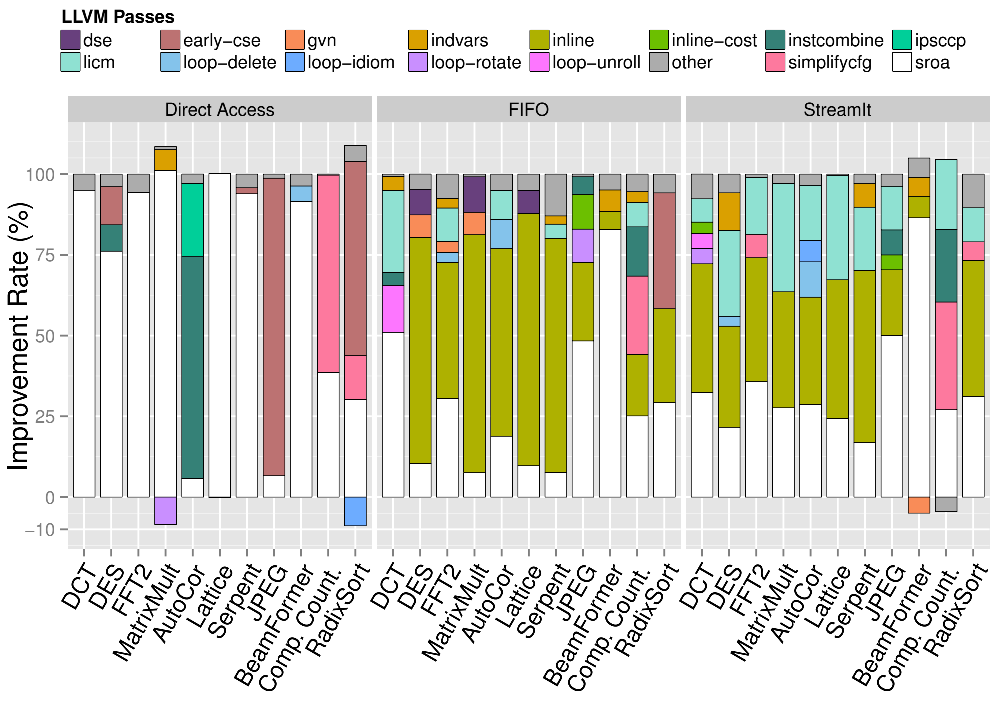


Figure 4.8: Contribution rate of particular LLVM optimization passes

ence 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 stream graph.

Figure 4.8 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.

Chapter 5

Abstract Interpretation-based Static Analysis to Resolve FIFO Queue Access Overhead

5.1 Complex Control-flow and Direct Memory Access of Stream Programs

The advent of multicore architectures and the emergence of Big-Data streaming applications [48, 73] have created an increased interest in stream-parallel programming languages specifically. Stream programs process continuous data streams through independent actors that communicate via FIFO data channels. Synchronous data-flow (SDF, [11, 49]) requires the data-rates of actors to be known a-priori. This information has been traditionally leveraged by compilers to transform the underlying stream graph structure to maximize data throughput [26, 31, 46, 54, 62]. In spite of all those efforts, little attention has been devoted to optimize the data transfer mechanism between communicating actors. The FIFO queue abstraction isolates the producer from the consumer, which facilitates modularity and separation of concerns across actors. Nevertheless, because FIFO queues are accessed indirectly via read- and write-pointers, they obscure the data-flow between actors. Hence, compilers are not able to optimize the information flow between actors. Standard compiler optimizations including register allocation and constant propagation become ineffective across actor boundaries.

The LaminarIR framework [45] introduced an intermediate representation (IR) for SDF [11]. LaminarIR converts FIFO queues to named token accesses. A named access is a concrete token in either an input or output queue of an actor. By naming the tokens, the FIFO buffer management can be shifted from run-time to compile-time. As a side-effect of this technique, splitters and joiners are eliminated whose task is to distribute and merge streams. The LaminarIR performs the transformation in two phases: a local and global direct access transformation. The local direct access transformations explicates the token-flow within an actor by replacing FIFO queues operations to local scalar variables. The global direct access dissolves the FIFO buffers between actors using the scalar variables instead of buffers, and dissolves splitters and joiners.

Although the framework in [45] gives superior performance to systems using run-time FIFO queues, the local access transformation is sometimes not successful. To still enable the global direct access transformation, the local direct-memory transformation has a fall back solution: inside the actor, local queues are instantiated as an interface between named tokens for the global direct memory transformation of a *program*, and the FIFO operations of the *actor*. This fall back solution obscures data-flows that passes through the actors with local queues, disrupting to exploit full advantage of the direct memory accesses of LaminarIR. To overcome this problem, an *enhanced actor transformation* is required.

The key research problem in the local direct memory transformation is that the queue position of a queue operation is dependent on the control-flow of the actor. However, the control-flow of an actor is in general undecidable in the presence of conditional branches and loops. To get a handle on the problem, we use the number of produced and consumed tokens of an actor that were specified by the programmer. If loops consume/produce tokens, the loops become bounded by the the production and consumption rates of the actor, respectively. This insight is used in this work to transform a program with data dependent control-flow into the form that local direct access transformation is applicable. Our work is based on an abstract interpretation framework [21, 22, 81], that deduces queue position of actor statements. An interval semantics is used to find the queue positions. Partial constant propagation for loop bounds and a narrowing operator [20, 22] are provided to sharpen the intervals. The contribution of this work is as follows:

- an enhanced actor transformation for the local access transformation in the LaminarIR framework,
- an abstract interpretation framework that gives precise queue positions employing intervals, and
- an experiment showing the improved performance of the enhanced actor transformation.

This chapter is organised as follows: In Section 5.2, we present a motivating example for our enhanced actor transformation. In Section 5.3, we describe the abstract interpretation analysis to obtain precise queue positions. In Sec. 5.4, we outline the local access transformation. In Section 5.5, we present the experimental findings of our enhanced actor transformation.

5.2 Motivating Example

In this section, we will present how an example actor of a stream program, RadixSort shown in the Figure 5.1a, is analyzed by abstract interpretation (Figure 5.2a, 5.2b, and 5.2c) and transformed to replace an actor's queue operations by named tokens (Figure 5.3b). This chapter also shows generated LaminarIR code without the proposed static analysis beside (Figure 5.1b to highlight the effectiveness of the new approach.

Input Stream Progrm Figure 5.1a shows our motivating example. The depicted node is a key actor in RadixSort, a benchmark from the StreamIt benchmark [76], which produces (aka. pushes) and consumes (aka. pops) `N` tokens per invocation of the actor's work function. The push and pop data rates must be specified already in the actor's source code (line 3). The RadixSort actor is parameterized by two stream parameters: (1) `N` denotes the number of tokens to sort, and (2) `radix` denotes the specific bit to sort on. Both parameters are compile-time constants, which are allowed to differ across actor instances.

```
int->int filter RadixSort
          (int N, int radix){
  work pop N push N {
    int[N] ordering;
    int i=0, j=0;
    // bounded loop:
    for(i=0; i<N; i++){
      int current=pop();
     //unbalanced if-else stmt:
      if((current & radix)==0){
        push(current);
      }else{
        ordering[j]=current;
        j++;
      }
    }
    // unbounded loop:
    for(i=0; i<j; i++){
      push(ordering[i]);
    }
  }
}
```

(a)

```
actor RadixSort {
  // int N=2; int radix=...;
  int out_array[2] = {0,0};
  int out_ctr = 0;
  int[2] ordering;
  int i=0, j=0;
  //unrolled bounded loop 1:
  int current=x1;
  if((current & radix)==0){
    out_array[out_ctr++] = current;
  }else{
    ordering[j]=current;
    j++;
  }
  //unrolled bounded loop 2:
  int current=x2;
  if((current & radix)==0){
    out_array[out_ctr++] = current;
  }else{
    ordering[j]=current;
    j++;
  }
  for(i=0; i<j; i++){
    out_array[out_ctr++] = current;
  }
  x1 = out_array[0];
  x2 = out_array[1];
}
```

(b)

Figure 5.1: (a) Original actor code and (b) corresponding plain LaminarIR code

```
int->int filter RadixSort
              (int N, int radix){
  work pop 2 push 2 {
    // int N=2; int radix=...;
    int[2] ordering;                    1:  ⟨[0,0],[0,0]⟩     1:  ⟨[0,0],[0,0]⟩
    int i=0, j=0;                       1:  ⟨[0,0],[0,0]⟩     1:  ⟨[0,0],[0,0]⟩
    i=0;//unrolled bounded loop 1:      1:  ⟨[0,0],[0,0]⟩     1:  ⟨[0,0],[0,0]⟩
    int current=pop();                  1:  ⟨[1,1],[0,0]⟩     1:  ⟨[1,1],[0,0]⟩
    if((current & radix)==0){
      push(current);                    2:    ⟨[0,0],[1,1]⟩   2:    ⟨[0,0],[1,1]⟩
    }else{
      ordering[j]=current;              2:    ⟨[0,0],[0,0]⟩   2:    ⟨[0,0],[0,0]⟩
      j++;                              2:    ⟨[0,0],[0,0]⟩   2:    ⟨[0,0],[0,0]⟩
    }                                   1:  ⟨[0,0],[0,1]⟩     1:  ⟨[0,0],[0,1]⟩
    i=1;//unrolled bounded loop 2:      1:  ⟨[0,0],[0,0]⟩     1:  ⟨[0,0],[0,0]⟩
    int current=pop();                  1:  ⟨[1,1],[0,0]⟩     1:  ⟨[1,1],[0,0]⟩
    if((current & radix)==0){
      push(current);                    2:    ⟨[0,0],[1,1]⟩   2:    ⟨[0,0],[1,1]⟩
    }else{
      ordering[j]=current;              2:    ⟨[0,0],[0,0]⟩   2:    ⟨[0,0],[0,0]⟩
      j++;                              2:    ⟨[0,0],[0,0]⟩   2:    ⟨[0,0],[0,0]⟩
    }                                   1:  ⟨[0,0],[0,1]⟩     1:  ⟨[0,0],[0,1]⟩
    for(i=0; i<j; i++){
      push(ordering[i]);                2:    ⟨[0,0],[1,1]⟩   2:    ⟨[0,0],[1,1]⟩
    }                                   1:  ⟨[0,0],[0,∞]⟩     1:  ⟨[0,0],[0,2]⟩
  }                                     0:⟨[2,2],[0,∞]⟩       0:⟨[2,2],[2,2]⟩
}
```

(a) (b) (c)

Figure 5.2: (a) After partial constant propagation and loop unrolling, (b) initial data rate intervals and, and (c) improved data rate intervals after narrowing derived by static analysis.

During execution of actor's work function, `pop()`-statement (line 8) reads tokens from the actor's input-channel and `push()`-statement (lines 11 and 19) writes tokens to the actor's output channel. Conventionally, data channels are implemented by FIFO-queues that access data tokens via indirect memory access. In contrast, LaminarIR [45] is an IR that exposes the queue-positions accessed by an actor through *named tokens*.

However, not all queue operations can be statically mapped to a static queue-position. For instance, matching queue-positions of queue operations in a loop which is not statically bounded can be arbitrary as shown in lines 18–20. Queue operation's in the if-else statement from lines 10–15 cannot be statically mapped to a static queue-position either, because its branches have different numbers of queue operations and condition is dynamic. We call such if-else statement an *unbalanced if-else statement*. For such queue operations involved with data dependent control-flows, LaminarIR falls back to local array and its index counter as shown in Figure 5.1b.

Our proposed static analysis resolves queue-positions of queue operations in dynamic control-flows as above in compile-time. To the best of our knowledge, the proposed work is the first approach that can resolve queue positions in dynamic control-flows in compile-time including unbalanced if-else statements.

Static Analysis for Queue Positions Because all stream parameter are compile-time constants, stream parameters are propagated and loops with static loop bounds are unrolled. We call this step *partial constant propagation and loop unrolling*. Figure 5.2a shows the result of partial constant propagation and loop unrolling of the original actor code, considering `N=2` for our running example. Note that `radix` has not been propagated for the sake of readability.

However, partial constant propagation and loop unrolling is not sufficient to resolve all queue operations, due to loops with data dependent bounds and the unbalanced if-else statement. To resolve the remaining data-dependent queue operations, we propose a static program analysis technique which is based on the observation that the variability of dynamic, data-dependent control-flow is restricted by the static data rates. E.g., because the for-loop depicted in lines 23–25 of Figure 5.2a produces one token per loop

```
actor RadixSort { // int N=2; ...
  int ordering[2]={0};
  int i=0, j=0;
  // root if-else stmt:
  int current=pop();
  if((current & radix)==0){
    push(current);
    int current=pop();
    // lifted if-else stmt 1:
    if((current & radix)==0){
      push(current);
      // lifted loop 1:
      for(i=0; i<j; i++){
        push(ordering[i]);
      }
    }else{
      ordering[j]=current;
      j++;
      // lifted loop 2:
      for(i=0; i<j; i++){
        push(ordering[i]);
    } }
  }else{
    ordering[j]=current;
    j++;
    int current=pop();
    // lifted if-else stmt 2:
    if((current & radix)==0){
      push(current);
      // lifted loop 3:
      for(i=0; i<j; i++){
        push(ordering[i]);
      }
    }else{
      ordering[j]=current;
      j++;
      // lifted loop 4:
      for(i=0; i<j; i++){
        push(ordering[i]);
} } } }
```

(a)

```
actor RadixSort { // int N=2; ...
  int ordering[2]={0};
  int i=0, j=0;
  // root if-else stmt:
  int current=x1;
  if((current & radix)==0){
    y1=current;
    int current=x2;
    // lifted if-else stmt 1:
    if((current & radix)==0){
      y2=current;
      // lifted loop 1:
      // dead code elimination
      // by static analysis

    }else{
      ordering[j]=current;
      j++;
      // lifted loop 2:
      y2=ordering[0];

    }
  }else{
    ordering[j]=current;
    j++;
    int current=x2;
    // lifted if-else stmt 2:
    if((current & radix)==0){
      y1=current;
      // lifted loop 3:
      y2=ordering[0];


    }else{
      ordering[j]=current;
      j++;
      // lifted loop 4:
      y1=ordering[0];
      y2=ordering[1];
} } }
```

(b)

Figure 5.3: (a) Intermediates actor code after complete AST transformation, and (b) LaminarIR code with named tokens after local direct access transformation

iteration, `N=2` is an upper bound for the number of loop iterations. For the static analysis, we abstract each statement of an actor's source-code by a pair of data rate intervals, $\langle[c_{\min}, c_{\max}], [p_{\min}, p_{\max}]\rangle$, which represent the minimum and maximum number of queue operations for consumption ($[c_{\min}, c_{\max}]$) and for production ($[p_{\min}, p_{\max}]$), respectively.

Figure 5.2b shows the initial data rate intervals for our running example. E.g., the pop statement in line 8 is abstracted as $\langle[1, 1], [0, 0]\rangle$ because the pop statement unconditionally pops one token and does not produce a token. Similarly, the data rate interval of the push statement in line 10 is $\langle[0, 0], [1, 1]\rangle$. The if-else statement in lines 9–14 is abstracted as $\langle[0, 0], [0, 1]\rangle$, as depicted on the last source-line of the compound statement in the line 14 of Figure 5.2b: the if-else statement does not consume a token, and may perform 0 or 1 push operations. The data rate intervals of the loop with data dependent bounds (from lines 23–25) are $\langle[0, 0], [0, \infty]\rangle$, because the loop contains a push operation and the loop's upper bound is not known yet.

As seen in the example, the data rate intervals of a compound statement are derived from the data rate intervals of its contained statements. Our analysis therefore derives the initial data rate intervals by a bottom-up traversal of an actor's abstract syntax tree (AST). The labels shown on the left side in Figures 5.2b and 5.2c denote the nesting depth of a statement in the AST. The data rate interval of a compound statement on nesting level L is decided by summing up the data rate intervals of the enclosed statements on nesting level $L + 1$. In our running example, the data rate interval of the AST's root node (nesting level 0 in line 27 of Figure 5.2b) is then decided by summing up all data rate intervals of nodes on nesting level 1, resulting in the data rate intervals $\langle[2, 2], [0, \infty]\rangle$.

The data rate interval narrowing phase is applied on the initialized data rate intervals to refine infinite data rate intervals as shown in Figure 5.2c. By the actor's data rate specification, the data rate intervals of the root node (representing the entire work-function) must be $\langle[2, 2], [2, 2]\rangle$. We apply a narrowing operation (to be introduced in Section 5.3.3) in a top-down fashion on AST. For example, the summation of all data rate intervals of nodes on nesting-level 1 cannot exceed the data rate interval of the root. Thus the interval for the production rate of the data dependent loop in lines 23–25 can be narrowed from $[0, \infty]$ to $[0, 2]$.

Actor Transformation for LaminarIR According to the analysis result by the proposed static program analysis, a program transformation is applied on the AST to shorten data rate intervals and make them singleton eventually. For instance, the first data rate interval divergence happens at the first if-else statement in line 9 of Figure 5.2a, and the diverged data rate interval obscures the data access indices of the following statements from lines 15–25. In this case, two code versions of the following statements are generated and lifted into each branch of the if-else statement as shown in lines 11–24 and lines 30–43 of Figure 5.3a. In the next iteration of the program transformation, a similar transformation is again performed on the nested if-else statements, generating different code versions of the following statement, which is a for-loop. Thus four code versions of the for-loop are generated and lifted into each branch of the nested if-else statement as shown in lines 14, 21, 33 and 40.

After the AST transformation, the validity of the transformed AST is examined by the *inspector*. During this phase, the inspector can detect dead code portions. E.g., the first code version of the for-loop in lines 14–16 of Figure 5.3a is eliminated, because the specified `N=2` number of push operations have been performed already on this program path and no more token production is allowed. Because each program transformation may affect the data rate intervals of other nodes, we repeat the program analysis and transformation pass until no more AST transformations are possible or needed (see Figure 5.7).

Finally, the local direct access transformation of LaminarIR is applied on the transformed AST (Figure 5.3a), which replaces queue operations with named tokens (Figure 5.3b).

5.3 SDF Program Analysis

The purpose of the proposed program analysis is to decide 1) whether the queue position of a queue operation can be determined at completing, and 2) what is queue position if it can be converted to a named token access. We use an abstract interpretation framework for answering the aforementioned questions. The abstract interpretation

$$
\begin{array}{rcl}
P & ::= & \textbf{actor}\ \mathit{id}\ \textbf{pop}\ \mathit{rate}\ \textbf{push}\ \mathit{rate}\ S \\
S & ::= & \textbf{skip} \\
& | & \mathit{var} = E \\
& | & \textbf{push}\ E \\
& | & S_1 ; S_2 \\
& | & \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \\
& | & \textbf{while}\ E\ \textbf{do}\ S
\end{array}
\qquad
\begin{array}{rcl}
E & ::= & \mathit{var} \\
& | & \mathit{val} \\
& | & \textbf{pop} \\
& | & \mathit{op}\ E \\
& | & E_1\ \mathit{op}\ E_2
\end{array}
$$

Figure 5.4: Abstract syntax of actor codes. For brevity, constructs not relevant for queue operations have been omitted.

framework finds sound intervals for queue positions of each statement in the actor.

For sake of simplicity, we provide a simplified programming language for actors that uses standard semantics. Figure 5.4 shows the abstract syntax of statements in an actor. An actor *Program P* requires the definition of static push and pop rates denoted by "**push** *rate*" and "**pop** *rate*". We have statement **push** and function **pop** for enqueuing and dequeuing tokens, respectively. The statements of an actor *P* are defined by nonterminal *S*, which are conditional branches, loops, and sequences. Non-terminal *E* represents *Expression*s used in the statements.

5.3.1 Determining Queue Positions by Abstract Interpretation

Figure 5.5 shows the abstract semantics that provides bounds on the number of queue operations from bottom to top of an AST. In this abstract semantics, the number of **push**s or **pop**s of the concrete semantics is abstracted by an integer interval $[l, u]$, where l indicate lower bound and u indicates upper bound of the number of tokens that were pushed and popped, respectively. Hence, the abstract state is a pair of intervals where each interval is associated either to the input and output queue of the actor, respectively. The abstract semantic domain $\hat{D}$ is defined as

$$\hat{D} = \hat{\mathbb{N}} \times \hat{\mathbb{N}},$$

$$\begin{array}{rcl}
[\![\textbf{actor}\ \textit{id}\ \textbf{pop}\ \textit{rate}\ \textbf{push}\ \textit{rate}\ S]\!] & = & [\![S]\!] \\
[\![\textbf{skip}]\!] & = & \langle [0,0],[0,0] \rangle \\
[\![\textit{var} = E]\!] & = & [\![E]\!] \\
[\![\textbf{push}\ E]\!] & = & [\![E]\!] \hat{+} \langle [0,0],[1,1] \rangle \\
[\![S_1;S_2]\!] & = & [\![S_1]\!] \hat{+} [\![S_2]\!] \\
[\![\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2]\!] & = & [\![E]\!] \hat{+} ([\![S_1]\!] \sqcup [\![S_2]\!]) \\
[\![\textbf{while}\ E\ \textbf{do}\ S]\!] & = & ([\![E]\!] \hat{+} [\![S]\!]) \hat{\times} \langle [0,\infty],[0,\infty] \rangle \\
[\![\textit{var}]\!] & = & \langle [0,0],[0,0] \rangle \\
[\![\textit{val}]\!] & = & \langle [0,0],[0,0] \rangle \\
[\![\textbf{pop}]\!] & = & \langle [1,1],[0,0] \rangle \\
[\![\textit{op}\ E]\!] & = & [\![E]\!] \\
[\![E_1\ \textit{op}\ E_2]\!] & = & [\![E_1]\!] \hat{+} [\![E_2]\!]
\end{array}$$

Figure 5.5: Abstract semantics of actor codes

where

$$\hat{\mathbb{N}} = \{[l,u] \mid l \le u \wedge l,u \in \mathbb{N}_\infty\} \cup \{\bot, \top\}$$

and $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$. $\hat{\mathbb{N}}$ is a partial ordered set where $[l_1,u_1] \sqsubseteq [l_2,u_2]$ iff $l_2 \le l_1 \wedge u_1 \le u_2$, and $\bot \sqsubseteq [l,u] \sqsubseteq \top$, $\forall [l,u] \in \hat{\mathbb{N}}$. By the algebraic product, semantic domain $\hat{D}$ is a partial ordered as well since the intervals are partial orders.

The γ-function of the Galois connection for the abstract interpretation is defined as

$$\begin{array}{rcll}
\gamma[l,u] & = & \{n \in \mathbb{N} \mid l \le n \le u\} & \text{when } u \ne \infty \\
 & = & \{n \in \mathbb{N} \mid l \le n\} & \text{when } u = \infty \\
\gamma\top & = & \mathbb{N}. &
\end{array}$$

The element infinity (∞) of the domain is an upper bound of any element in $\mathbb{N}$, and is introduced to represent an unbounded quantity. For example, the number of loops iterations of a loop may be unbounded without knowing the loop invariants. Hence, the production and consumption of tokens can be still be described by an unbounded number of tokens using the symbol ∞. This information loss will be restored in part by partial constant propagation for the loop bounds (Section 5.3.2) and narrowing operator of the abstract interpretation (Section 5.3.3). Both approaches exploit static properties of the SDF program.

The arithmetic operations on the infinity are defined below,

$x + y$		y	
		$\mathbb{N}$	∞
x	$\mathbb{N}$	$x + y$	∞
	∞	∞	∞

$x \times y$		y		
		0	$\mathbb{N}^+$	∞
	0	0	0	0
x	$\mathbb{N}^+$	0	$x \times y$	∞
	∞	0	∞	∞

Binary operators $\hat{+}$, $\hat{\times}$ and $\sqcup$ are defined on $\hat{\mathbb{N}}$ as below and the operators on $\hat{D}$ are defined compositionally. The operators are monotonic.

$$\begin{aligned}
[l_1, u_1] \hat{+} [l_2, u_p] &= [l_1 + l_2, u_1 + u_2] \\
[l_1, u_1] \hat{\times} [l_2, u_2] &= [l_1 \times l_2, u_1 \times u_2] \\
[l_1, u_1] \sqcup [l_2, u_2] &= [\min(l_1, l_2), \max(u_1, u_2)]
\end{aligned}$$

5.3.2 Derivation of Loop Bounds by Partial Constant Propagation

As shown in Figure 5.5, the abstract semantics of "**while** *E* **do** *S*" evaluates any non-zero number of queue operations of the child nodes to infinity assuming loop bounds of the statement are unknown ($\langle [0, \infty], [0, \infty] \rangle$). This approach is sound, may prohibit a local actor transformation.

The major advantage of actors in SDF graphs is that they have static data rates for pushes and pops. Thus, if a given actor code is correctly specified, a loop that includes queue operation in the loop body becomes bounded since the production and consumption rate are bounded by the actor specification. Because of this property, loop bounds are likely to be determined even without considering the loop conditions.

For example, StreamIt [77] is a stream programming language that shares a number of properties with the SDF programming model. When loop bounds are known during compile-time, most likely by static stream parameters that are evaluated at the level of stream graph generation of a program, then the unknown loop bounds $\langle [0, \infty], [0, \infty] \rangle$

$$\begin{aligned}
[\![\textbf{actor}\ \mathit{id}\ \textbf{pop}\ \mathit{rate}\ \textbf{push}\ \mathit{rate}\ S]\!]^{\downarrow}\sigma &= [\![S]\!]^{\downarrow}\sigma[[\![S]\!] \mapsto [\![S]\!]\Delta\mathcal{R}] \\
[\![\textbf{skip}]\!]^{\downarrow}\sigma &= \sigma([\![\textbf{skip}]\!]) \\
[\![\mathit{var} = E]\!]^{\downarrow}\sigma &= \sigma([\![\mathit{var} = E]\!]) \\
[\![\textbf{push}\ E]\!]^{\downarrow}\sigma &= \sigma([\![\textbf{push}\ E]\!]) \\
[\![S_1 ; S_2]\!]^{\downarrow}\sigma &= [\![S_1]\!]^{\downarrow}\sigma_1 [\![S_2]\!]^{\downarrow}\sigma_2 \\
&\text{where } \sigma_1 = \sigma[[\![S_1]\!] \mapsto [\![S_1]\!]\Delta(\sigma([\![S_1 ; S_2]\!]) - [\![S_2]\!])] \\
&\phantom{\text{where }} \sigma_2 = \sigma[[\![S_2]\!] \mapsto [\![S_2]\!]\Delta(\sigma([\![S_1 ; S_2]\!]) - [\![S_1]\!])] \\
[\![\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2]\!]^{\downarrow}\sigma &= [\![E]\!]^{\downarrow}\sigma[\![S_1]\!]^{\downarrow}\sigma_1[\![S_2]\!]^{\downarrow}\sigma_2 \\
&\text{where } \sigma_1 = \sigma[[\![S_1]\!] \mapsto [\![S_1]\!]\Delta(\sigma([\![\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2]\!]) - [\![E]\!])] \\
&\phantom{\text{where }} \sigma_2 = \sigma[[\![S_2]\!] \mapsto [\![S_2]\!]\Delta(\sigma([\![\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2]\!]) - [\![E]\!])] \\
[\![\textbf{while}\ E\ \textbf{do}\ S]\!]^{\downarrow}\sigma &= [\![E]\!]^{\downarrow}\sigma\ [\![S]\!]^{\downarrow}\sigma \\
[\![\mathit{var}]\!]^{\downarrow}\sigma &= \sigma([\![\mathit{var}]\!]) \\
[\![\mathit{val}]\!]^{\downarrow}\sigma &= \sigma([\![\mathit{val}]\!]) \\
[\![\textbf{pop}]\!]^{\downarrow}\sigma &= \sigma([\![\textbf{pop}]\!]) \\
[\![\mathit{op}\ E]\!]^{\downarrow}\sigma &= \sigma([\![\mathit{op}\ E]\!]) \\
[\![E_1\ \mathit{op}\ E_2]\!]^{\downarrow}\sigma &= \sigma([\![E_1\ \mathit{op}\ E_2]\!])
\end{aligned}$$

Figure 5.6: Semantic definitions for narrowing operator

in the $[\![\textbf{while}\ E\ \textbf{do}\ S]\!]$ can be specified by $\langle[l_k, u_k], [l_k, u_k]\rangle$ when l_k means lower bound and u_k means upper bound of the loop.

5.3.3 Narrowing Operator

We define a narrowing operator [20, 22] for the abstract semantics to enhance the quality of the analysis. The aforementioned static property of the SDF graphs will be applied on a whole analysis result of an actor code by a narrowing operator Δ.

Figure 5.6 shows semantic definitions for the narrowing operator. The narrowing operator is evaluated from top to bottom of the AST. An environment σ is introduced which is a set of bindings that maps each node of an AST to an element of $\hat{D}$. Each node in AST is initially mapped to its analysis result by the abstract semantics. If N denotes a set of nodes in an AST and Σ is a set of environments, then we define a function $\sigma : N \to \Sigma \to \hat{D}$ by abuse of notation. Notation '$f[x \mapsto w](y)$' denotes a function that returns value w when x is given to the function f and agrees with f otherwise. Note that data rate intervals on *Expression*s is always singleton. Thus narrowing operator is omitted for $[\![E]\!]$.

The operator Δ on $\hat{\mathbb{N}}$ is defined as

$$[l_1, u_1]\Delta[l_2, u_2] = [\max(l_1, l_2), \min(u_1, u_2)]$$

and the operator Δ on $\hat{D}$ is defined compositionally. The operator Δ satisfies

$$\forall x, y \in \hat{D} : x \sqsubseteq y \Rightarrow x \sqsubseteq (x\Delta y) \sqsubseteq y$$

and the decreasing chain $y_0 = x_0, y_{i+1} = y_i \Delta x_{i+1}$ for all decreasing chain $x_0 \geq x_1 \geq \cdots$ is finite obviously. Thus the operator Δ is a narrowing operator. The semantics for the narrowing operator also introduces a new operator $-$ which is defined as

$$\begin{aligned} [l_1, u_1] - [l_2, u_2] =& [l_1 - u_2, u_1 - l_2] && \text{when } l_1 < u_2 \wedge l_2 < u_1 \\ =& [0, u_1 - l_2] && \text{when } l_1 \leq u_2 \wedge l_2 < u_1 \\ =& [l_1 - u_2, 0] && \text{when } u_2 < l_1 \wedge u_1 \leq l_2 \\ =& [0, 0] && \text{when } l_1 \leq u_2 \wedge u_1 \leq l_2 \end{aligned}$$

on $\hat{\mathbb{N}}$, and the operator is defined on $\hat{D}$ compositionally. By apply the narrowing operator from the root of the AST, the intervals for the number of queue operations are bounded by the static data rate $\mathcal{R} \in \hat{D}$ which is $\langle [r_p, r_p], [r_c, r_c] \rangle$ where r_p and r_c denote push and pop data rates specified by the actor declaration respectively.

Using the refined analysis, loop bounds are deduced. A subsequent transformation unrolls loop bounds and performs the local access transformation as outlined in [45].

5.4 Program Transformation

This section describes each step in program transformation phase, AST transformation, inspector and local direct access transformation in Figure 5.7.

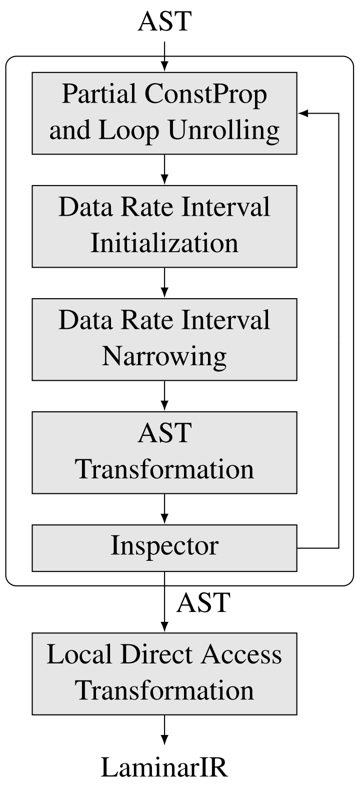

AST
Partial ConstProp and Loop Unrolling
Data Rate Interval Initialization
Data Rate Interval Narrowing
AST Transformation
Inspector
AST
Local Direct Access Transformation
LaminarIR

Figure 5.7: Analysis compilation path

5.4.1 AST Transformation

AST transformation is to flatten control-flow paths, so more queue operations can be mapped with a static queue position as a result. In other words, AST transformation is performed to shorten data rate intervals and make them singleton eventually. Data rate intervals grow when multiple control-flows or branches with different number of queue operations join at a point, such as if-else statements and loops. Thus, two standard program transformation techniques are applied in the AST transformation to flatten each of if-else statements and loops.

The first standard technique is loop-unrolling [5] where a single unrolling step is depicted by

$$\texttt{while(C)\{S1;\}} \rightarrow \texttt{if(C)\{S1; while(C)\{S1;\}\}}.$$

Unrolling is performed when the loop bounds are constant.

The second standard technique is trace partitioning [10, 56, 68] which is a technique that delays joining point of multiple branches to prevent growing abstract information. Trace partitioning is implemented by lifting all following statements of an if-else statement into every branches, such as,

`if(C){S1;}else{S2;}S3;` $\rightarrow$ `if(C){S1;S3;}else{S2;S3;}`.

Performing loop unrolling with trace partitioning can arouse interactive analysis improvement especially when loops or if-else statements are nested to the others. However, generating new code versions for each transformation can expand the code size. Thus, we limit the maximum number of transformation rounds to 100 per actor instance and exit the transformation loop. When an actor code needs more than 100 rounds of transformation, a half-way flattened AST is passed to local direct access transformation. The way to transform a half-way flattened AST with the mixture of resolved and un-resolved queue positions for queue operations will be covered in Section 5.4.3.

5.4.2 Inspector

After the AST transformation, the generated code is passed on to the *inspector*. The inspector examines the validity of the transformation by simulating queue operations. A valid transformation should map a static queue position to a queue operation, and the static queue position should not exceeds specified data rate. For instance, if the minimum number of queue operations of a control-flow path exceeds the number of remaining queue operations to be performed, then the path is examined to be a dead code and eliminated.

Because each program transformation and dead code elimination performed by inspector can affect the data rate intervals of other nodes, we repeat the program analysis and transformation pass until no more AST transformations are possible or needed. As long as the data rates are static, the number of iterations is constant.

5.4.3 Local Direct Access Transformation

Local direct access transformation is an actor-wise program transformation phase which reads transformed AST of an actor code and generates corresponding LaminarIR

Table 5.1: Hardware configuration

CPU	Intel i5-4690	AMD Opteron 6378	ARM Cortex-A15
Clock Freq.	2.3 GHz	2.4 GHz	1.7 GHz
Inst. Cache	32 kB	64 kB, shared by 2 cores	32 kB
L1 Data Cache	32 kB	16 kB	32 kB
OS (Kernel ver.)	Ubuntu 14.04 (3.13)	CentOS 7.2 (3.10)	Linaro 13.09 (3.11)
C Compiler	LLVM/Clang 3.7		

code, where queue operations are replaced with named tokens.

Preciously, local direct access transformation examined whether queue positions of all push or pop operations are resolved in an actor to replace queue operations with named tokens. If it was not possible to match every each pop/push operation with a static queue position, then a read/write counter and a local array for the pop/push operations were introduced to simulate the FIFO operations at run-time through the whole actor code.

With the proposed approach, we extended the local direct access transformation that can replace part of the queue positions that have been resolved with named tokens. In addition, the *minimum resolved queue position* that is guaranteed to be resolved for all control-flow paths is calculated. Named tokens are used for resolved queue operations that are mapped with smaller queue positions than the minimum resolved queue position, and otherwise FIFO queues are employed that accesses local arrays. Calculating the minimum resolved queue position enable to reduce size required for the local arrays, meaning that less number of copy operations in between named tokens and local arrays.

5.5 Experimental Results

To evaluate the efficiency of the proposed technique, we implemented the enhanced actor transformation for the LaminarIR framework [1, 45] and assessed the performance improvements. Because the local access transformations that we apply in this work are

Table 5.2: Benchmark characteristics

Benchmarks	Parameters	# Trans.		AST Trans. Stats	
		Eval.	Full	If-else	Loop
BubbleSort	data size (16)	2		32	0
MergeSort	data size (16)	33		64	79
RadixSort	data size (8)	255		2805	0
RunLength Enc.	data size (16)	100	>1000	100	0
RunLength Dec.	data size (16)	100	>1000	57	44
JPEG Enc.	window size (64)	100	>1000	100	0

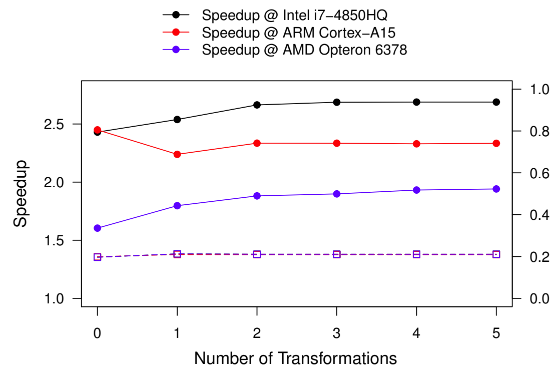
reported to increase a program's code size [9], we also state the code size increase of the generated executable binaries.

We evaluate the proposed technique on two mobile processors, an Intel i5-4690, and an ARM Cortex-A15 platform, and on an AMD Opteron 6378 x86 server CPU. The characteristic features of the tested platforms are summarized in Table 5.1. We employ six StreamIt benchmarks [76] with distinctive control-flow patterns to evaluate how our technique is affected by varying control-flow patterns. To generate ASTs from the StreamIt benchmark sources, we adopted the StreamIt framework's compiler frontend. After applying our transformation, the LaminarIR intermediate representation [45] is generated from the transformed AST. The LaminarIR framework translates LaminarIRs to C codes.

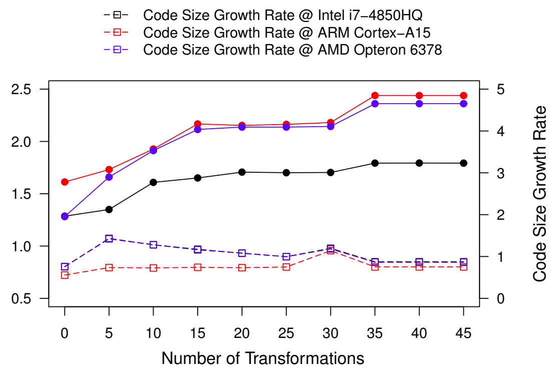
We employ the LLVM compiler infrastructure to compile the generated C code for all evaluated platforms. Execution times are measured using the PAPI hardware performance counter library [16]. We used randomized program input for all experiments, to prevent LLVM to compute partial results already at compile-time (see [45]).

5.5.1 Performance Evaluation

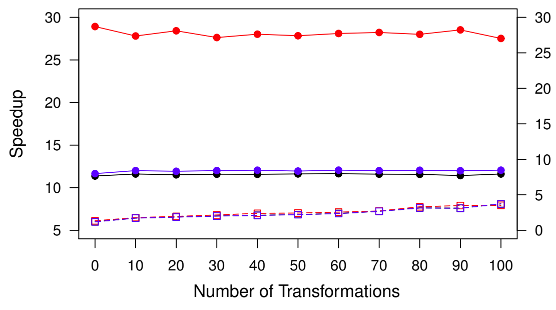
In this section, we describe each control-flow case that we observed in six StreamIt benchmarks (Table 5.2). RadixSort contains a loop with static bounds and an unbalanced



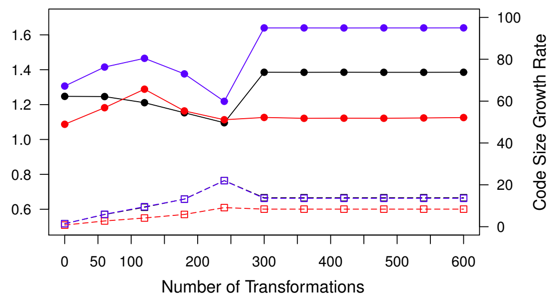
(a) BubbleSort



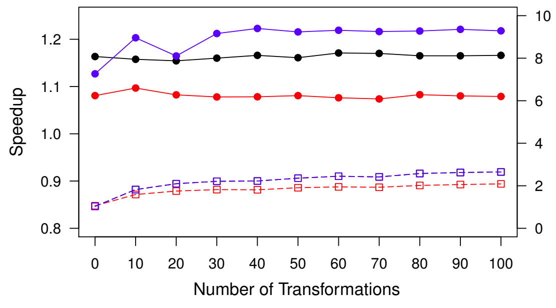
(b) MergeSort



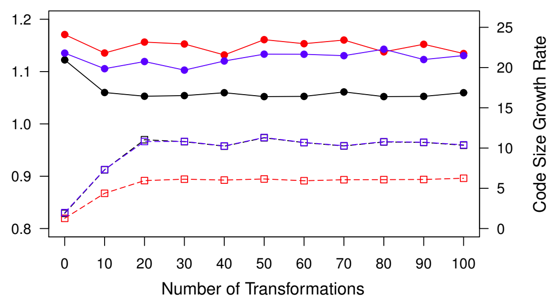
(c) JPEG Encoder



(d) RadixSort



(e) RunLength Encoder



(f) RunLength Decoder

Figure 5.8: Speedup and code size growth rate of LaminarIR over FIFO queues by number of transformations

Table 5.3: Speedup of LaminarIR over FIFO queue with base ASTs (column “base”), transformed ASTs (column “overall”), and the speedup differences by the proposed static analysis (column “SA”).

Benchmarks	Intel i5-4690		
	base	SA	overall
BubbleSort	243.03%	+25.85%	268.87%
MergeSort	128.56%	+50.78%	179.33%
RadixSort	124.73%	+13.82%	138.55%
RunLength Enc.	116.35%	+0.25%	116.60%
RunLength Dec.	112.23%	-6.27%	105.96%
JPEG Enc.	1137.26%	+24.0%	1161.27%
Benchmarks	ARM Cortex-A15		
	base	SA	overall
BubbleSort	244.96%	-11.56%	233.40%
MergeSort	161.26%	+82.63%	243.90%
RadixSort	108.71%	+3.86%	112.57%
RunLength Enc.	108.08%	-0.18%	107.09%
RunLength Dec.	117.08%	-3.62%	113.45%
JPEG Enc.	2891.89%	-139.64%	2752.25%
Benchmarks	AMD Opteron 6378		
	base	SA	overall
BubbleSort	160.42%	+33.71%	194.21%
MergeSort	128.16%	+107.96%	236.12%
RadixSort	130.63%	+33.38%	164.01%
RunLength Enc.	112.69%	+9.10%	121.79%
RunLength Dec.	113.53%	-0.46%	113.06%
JPEG Enc.	1165.44%	+40.46%	1205.90%

if-statement nested within the loop. Because of the nested unbalanced if-statement of the loop, another loop which follows the first loop automatically has data dependent bounds. BubbleSort is composed of selective if-else statements, and MergeSort has a sequence of data dependent loops including a while loop. RunLength decoder contains a nested data dependent loop in another loop with static loop bounds. Both the RunLength encoder and the JPEG encoder have a sequence of data dependent loops with nested and unbalanced if-else statements.

BubbleSort BubbleSort (Figure 5.8a) has two consecutive unbalanced if-else branches. That means that the static analysis can narrow liveness of branches of latter unbalanced if-else statement because branches become more specific by trace partitioning. First AST transformation copies latter unbalanced if-else statement into every branch of the former unbalanced if-else statement. Code size increment due to the code copy is negligible, however, eviction rate of the first-level instruction cache increases. A noticeable speed down with a single AST transformation on ARM Cortex-A15 shown in Figure 5.8a depicts such case. Intel i7-4850HQ and AMD Opteron 6378 accommodate instruction queues which eventually effect as larger L1 instruction cache. After two AST transformations, AST of BubbleSort is fully flattened and all queue operations are replaced with named tokens.

MergeSort MergeSort (Figure 5.8b) contains two serial loops with data dependent loop bounds, but the sum of iterations of the two serial loops should be constant. It means loop bounds of the second loop are dependent on the behavior of the first loop. Due to this characteristics, continuous trace partitioning on the sequence of unrolled loop iterations did not increase code size, but improved performance gradually up to 2.0$\times$more performance with 33 AST transformations to a fully flattened program. Nevertheless, a local input array with a read counter had to be introduced for MergeSort despite of the full flattening due to data dependent peeking indices.

RadixSort RadixSort (Figure 5.8d) is similar to MergeSort, but the first loop of the RadixSort contains an unbalanced if-else statement in addition. Because of additional trace partitioning required for the nested unbalanced if-else statement, code size increment over AST transformations begins to overwhelm gains from the direct memory accesses at around a point when the code size becomes larger than $60\,\mathrm{kB}$ after 275 rounds of AST transformations. But after 510 rounds of AST transformations, program is fully flattened replacing all queue operations with named tokens. Complete FIFO queue operation replacement with named tokens maximizes benefits of direct memory access, recovering code size to the one with FIFO queues.

RunLength Decoder RunLength Decoder shown in Figure 5.8f contains a nested loop where loop bounds of the inner loop are data dependent. This is the most exhaustive case among other cases in terms of the number of required AST transformations to fully flatten a program. In this case, the first few AST transformation resulted slight performance improvement and more AST transformations resulted in no notable performance changes.

RunLength Encoder and JPEG Encoder RunLength Encoder is the last step of the JPEG Encoder. Control-flow pattern of RunLength Encoder is an advanced version of RadixSort with a two-level unbalanced if-else statements nested in the first loop. When our proposed technique is applied on RunLength Encoder, a 3.0% performance improvement is obtained from 100 rounds of AST transformations resulting in a $1.15\times$overall speedup on average on the three processors (Figure 5.8e). With JPEG Encoder, which is a real world application that exploits RunLength Encoder, the proposed static analysis enables maximum 40.46% additional performance improvement (Figure 5.8c). Considering that RunLength Encoder is the only actor that is applicable to the proposed static analysis in JPEG Encoder, we can assess that effectiveness of a same AST transformation can grow if the clarified data-flow by direct memory access influences more part of a program.

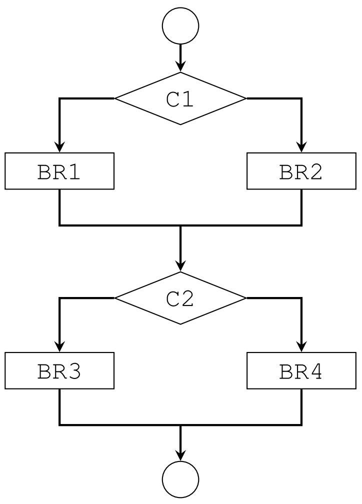
Table 5.3 shows effectiveness of the proposed static analysis in speedups of LaminarIR over FIFO queues. Column “base” shows the speedups of LaminarIR with the base AST where only partial constant propagation is applied, and column “SA” shows speedup changes of LaminarIR with transformed AST. Column "overall" shows the overall speedup after indicated number of transformations in Table 5.2, which are summation of "base" and "SA" of each benchmark.

Table 5.2 shows benchmark characteristics with statistics of static analysis. Column "# Trans" denotes rounds of AST transformations required to fully flatten control-flow of a program. ">1000" in Column "# Trans" means the control-flow of the benchmark cannot be fully flattened in 1000 times of AST transformations, and ASTs after 100 rounds of AST transformation are used for performance evaluation in such cases. Column "AST Trans. Stats" shows whether loop-unrolling or trace partitioning has happened over the designated number of AST transformations. A single round of static analysis traverses all actors in a program, thus AST Trans. Stats of BubbleSort is interpreted as two AST transformation is applied on each of 16 actors, and all of the AST transformations are trace partitioning.

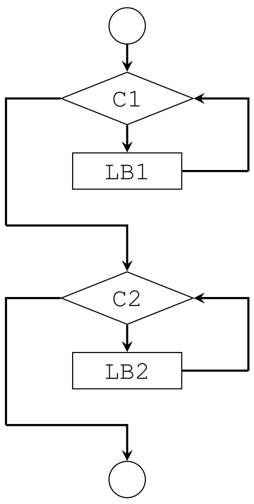
5.5.2 Case Studies

To categorize representative control-flow patterns of the observed benchmarks and estimate further behaviours of each control-flow pattern over the proposed optimizations, we introduce four synthetic benchmarks where each benchmark contains a single computing actor of a specific representative control-flow pattern. The four control-flows patterns are made by compositions of two control-flow units, If-else and loop.

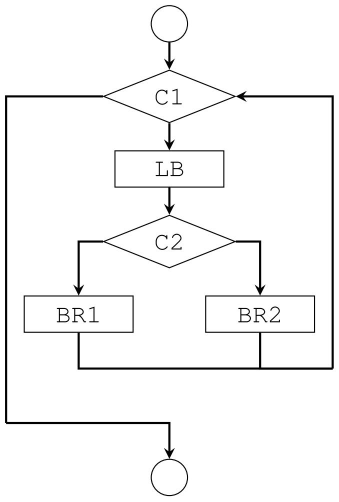
Figure 5.9 shows control-flow diagrams of the four control-flow patterns called Case 1 to Case 4, which also represent characteristics of the real world benchmarks described in Section 5.5.1. Case 1 shows the simplest control-flow pattern which is a serial of two or more If-elses. The number of possible control-flows is dependent on number of If-elses. Thus we varied the numbers of If-elses in an actor by 2, 4, 8, 16 and 32 to see how the increased control-flow complexity by the serial If-else statements are affected by the AST transformation. Case 2 shows a little bit more advanced control-flow pattern that contains two loops, one followed by the other. Loop bounds of the latter loop are dependent on the former loop. It means the number of possible control-flows is dependent on the loop bounds of the former loop, thus we varied loop bounds



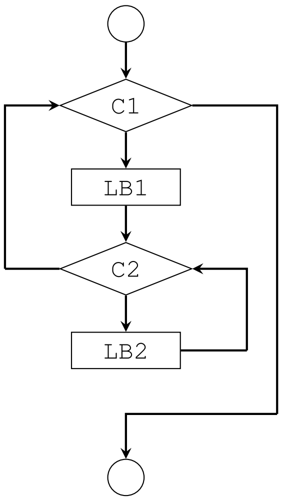
(a) Case 1: If-else and Loop



(b) Case 2: Loop and Loop



(c) Case 3: If-else in Loop



(d) Case 4: Loop in Loop

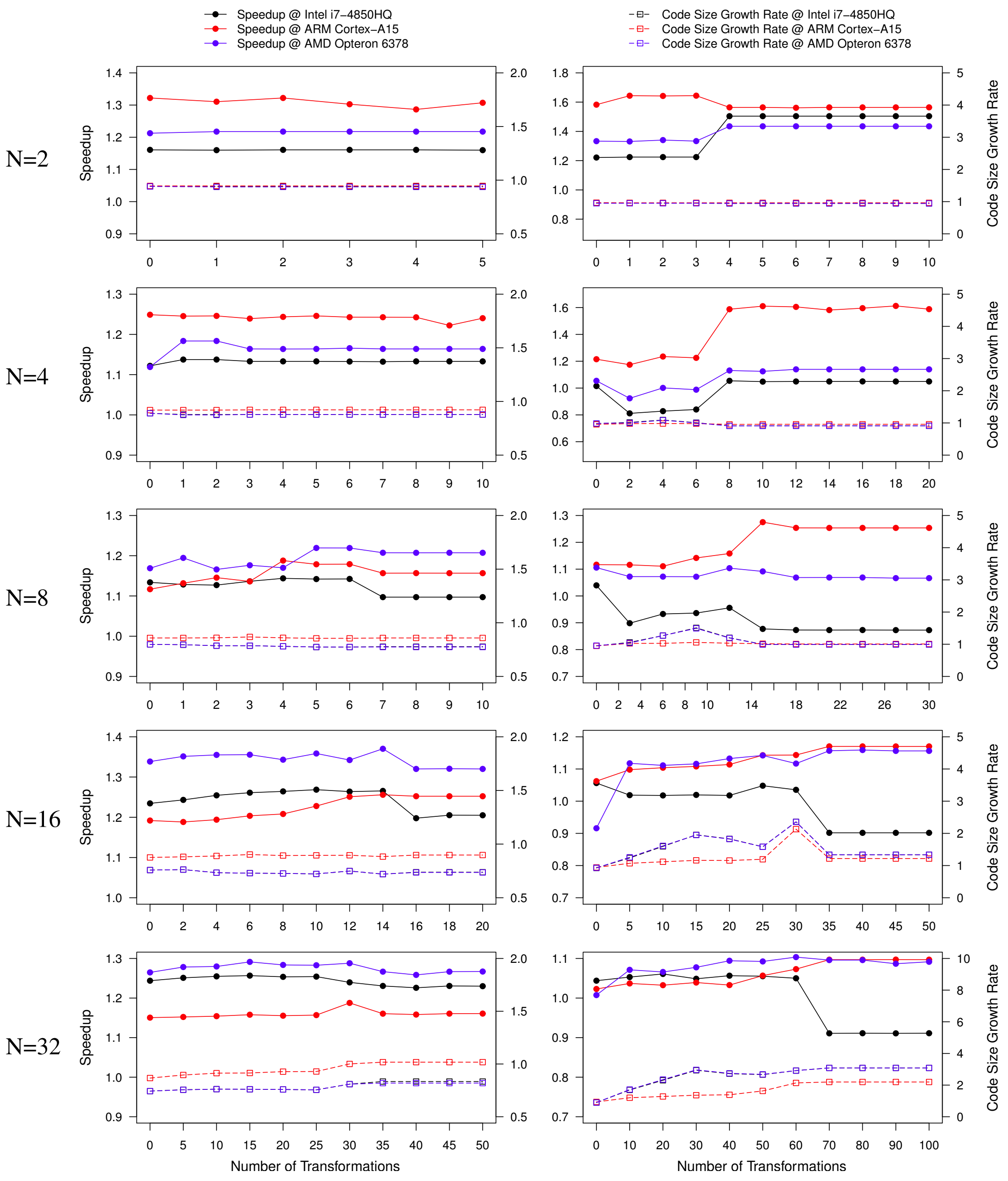
Figure 5.9: Representative control-flow cases in stream programs

for the test to the same factors aforementioned. Case 3 and Case 4 represent a nested If-else and Loop in an outter Loop respectively. Unrolling a Loop with a nested control-flow diverging component increases number of possible control-flows exponentially. In Case 3, we fixed the exponent by not differenciating number of nested If-else statements, and exponent in Case 4 was varied by increasing loop bounds of the nested loops as well as the ones of the outter loop.

Figure 5.10 and 5.11 show the performance and code size changes of the four cases over number of AST transformations. The numbers are proportional to FIFO versions, and the numbers with 0 number of AST transformation represents result of plain LaminarIR without static analysis and AST transformation. Randomly generated inputs are used for all experiments. However, because control-flows are input data dependent, the behavior of four cases over different number of AST transformations may differ by different input data sequences. Graphs with x-axis that ranges from 0 to 100 mean that the particular testing case is not fully flattened in 1000 AST transformations, and the other graphs shows the behaviour of programs which can be fully flattened within the shown x-axis range.

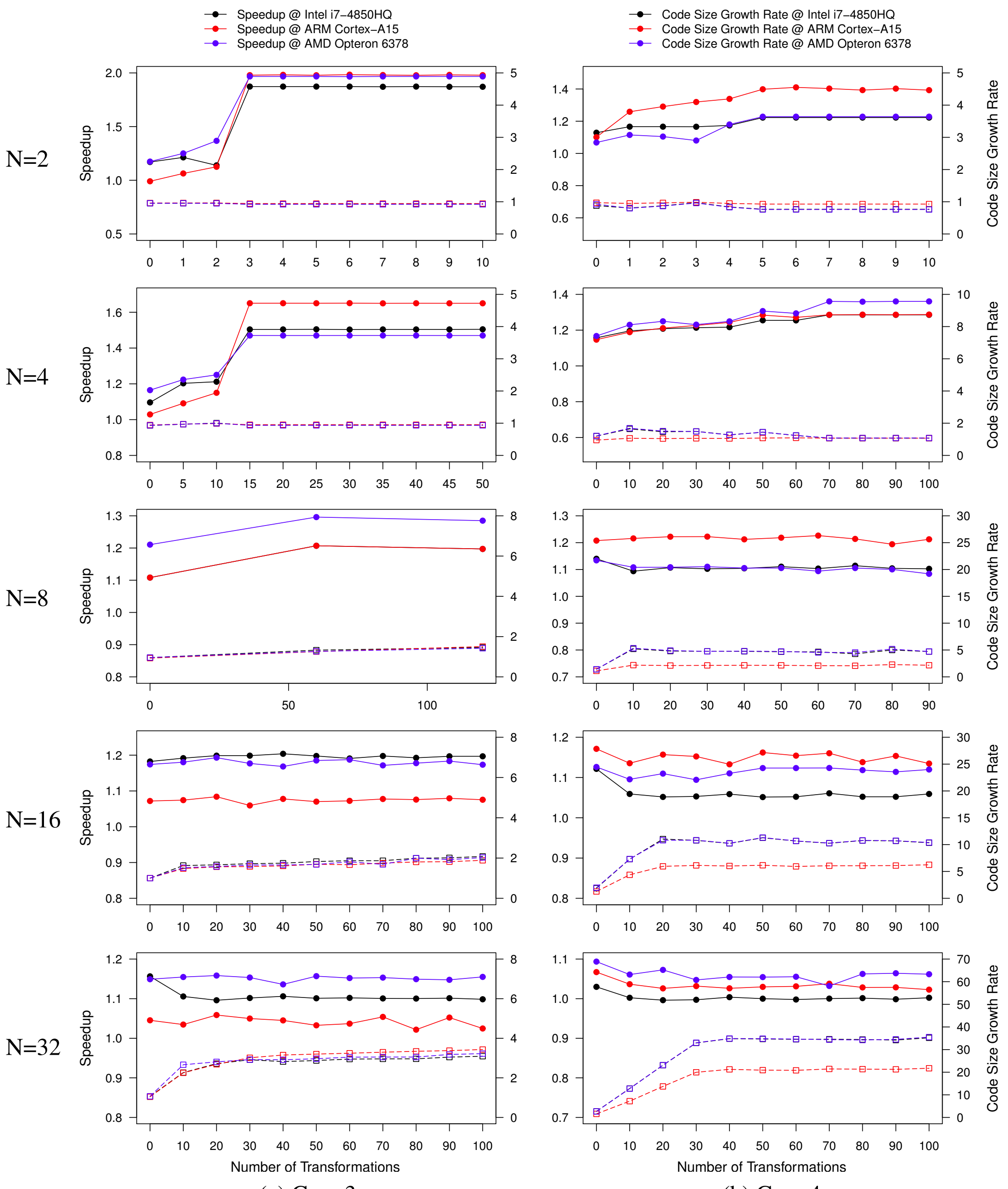
In perspective, If-else only control-flows (Figure 5.10a are too simple to benefit from the static analysis, and control-flows with Loops (Figure 5.10b and 5.11) benefit from the static analysis if the complexity size fits into the instruction cache size. Despite of the burst of lines of code by a hundred AST transformations, the compiled code size remained around factor of 2 over the code size of FIFO queues for all cases except Case 4. The steady code size is because the flattened AST enabled more queue operations to be replaced by scalar variable, which helped compiler to optimize more on flattened control-flows.

This case study also reveals shortcomings of the approach. One is the sharp drop of performance on Intel i5-4690 when a program is fully flattened in few cases of Case 2. Case 2 contains three Loops, two as shown in the Figure 5.9b and the last loop to flush remain input queue data or fill up remain output queue data if there is any. Against to the intuitive expectation that the instruction cache miss by the three unrolled loops would en the origin of the performance degradation, we observed that the scalar replace-



(a) Case 1 (b) Case 2

Figure 5.10: Experimental results on (a) Case 1 and (b) Case 2: Speedup and code size growth rate of LaminarIR over FIFO queues by number of transformations



(a) Case 3 (b) Case 4

Figure 5.11: Experimental results on (a) Case 3 and (b) Case 4: Speedup and code size growth rate of LaminarIR over FIFO queues by number of transformations

ment queue operations hampers two modern microarchitectural techniques, instruction queueing and instruction re-ordering [41]. Instruction queuing is designed to reduce time consumed to fetch instructions. Instruction queuing can behave as a instruction cache when executing over a loop with a small loop-body. Thus small loops can not exploit the instruction queue when unrolled and even can overfull the instruction queue which increases instruction fetch latency. Similar instruction queuing is required for instruction re-ordering as well. Instruction re-ordering engine queues micro operations until all source operands are ready, schedules and dispatches ready micro operations to the available execution units. The queue overfulls when the dependency chain of operands is too long. The clarified data-flow of the fully flattened Case 2 generated too long data dependency chain which overfulls instruction queues in microarchitectures.

Chapter 6

Communication Cost Aware Orchestration

6.1 Communication Overhead of Stream Programs from Parallelization

Recently, the mapping of stream graphs on processors (also known as orchestration) has received a lot of attention [18, 19, 26, 27, 31, 33, 44, 47, 75, 78, 79, 80, 84]. In this work, we introduce an actor placement algorithm that maps actors of stream graphs to processors by considering communication costs of data channels. The algorithm balances the workload of processors such that the *makespan* becomes minimal. The theory of approximation algorithms [82] was used to design our actor placement. Approximation algorithms are an active field of research in optimization theory and theoretical computer science. Unlike heuristics, approximation algorithms provide solutions whose value is within a factor of the optimal solution and their solution can be computed in polynomial time. Other exhaustive search techniques including integer linear programming (ILP) and branch&bound techniques compute optimal solutions, however, for larger input sizes exhaustive search techniques become intractable.

We employ the semantics of *Synchronous Data Flow* (SDF) [11] that restricts the general Kahn's processes [43]. In SDF, an actor consumes and produces a fixed number of tokens when it is fired. The fixed number of tokens consumed and produced in an actor firing permits the computation of a *static finite periodic schedule* [11], making the processing model simple. An example program of StreamIt, which employs SDF semantics, is depicted In Figure 6.1. StreamIt programs use structured stream graphs that are composed of filters, pipelines, split-joins, and feedback loops. Dependencies between actors caused by data channels are deferred between iterations of the finite periodic schedule, making the parallel execution of an actor independent of each other in an iteration.

Although stream programs contain an abundance of parallelism, obtaining an efficient mapping onto parallel architectures is nevertheless a challenging problem. The gains obtained from parallel execution are easily overshadowed by communication. The workload of a processor comprises both the execution time of actors and the communication overheads of data channels. In [27], it was shown that the makespan can worsen by up to 346% if communication costs of data channels are not considered.

The main contributions of this work are:

- an approximation algorithm for the actor placement problem that considers communication costs of data channels,
- *instance bounds* that provide better bounds than $\log_2 n$ for a concrete problem instance[*], and
- an evaluation of our algorithm for StreamIt.

The chapter is organized as follows: In Section 6.2, we provide a motivating example. In Section 6.3, we describe the Actor Placement Problem (APP), prove its NP-hardness, and formally introduce structured stream graphs that are at the core of our approximation algorithm. In Section 6.4, we present the approximation algorithm that exhibits a polynomial run-time, and prove its correctness and approximation bounds. In Section 6.7, we present the experimental evaluation of our approximation algorithm using StreamIt.

[*]For a concrete instance, a better bound can be computed which is called an "instance bound". The approximation ratio $\log_2 n$ is a worst case for all problem instances.

6.2 Motivating Example

We motivate our method by the StreamIt [3] program example in 6.1(a). The program consists of four actors A_1, A_2, A_3 and A_4 that emit and consume sequences of integer tokens. The counter variable `x` is initialized in the `init`-section of actor A_1. In the `work`-section of actor A_1, the counter value is pushed onto the output stream and the counter is incremented with each actor invocation. The `push 1` statement within the work function specifies the *production rate* of 1 token per actor invocation. Similarly, Actor A_2 has a *consumption rate* of 1 token per actor invocation (specified by the `pop 1` statement) and a production rate of 6 tokens. Actor A_3 has reversed consumption and production rates, i.e., it consumes and produces 6 and 1 tokens respectively. Actor A_4 represents the sink of the stream graph, which simply prints the received integer tokens.

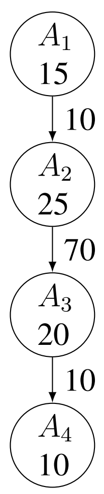
The stream graph for this example is depicted in 6.1(b). The numbers associated with the nodes and the edges of the graph represent actor execution times and data communication overheads respectively. We assume that pairwise communication costs between actors are semi-metric[†]. For instance, if actors A_1 and A_2 are assigned to two different processors, a communication overhead of 5 time units is incurred.

Let us assume that we want to assign the actors of the stream program to two processors of a cache-coherent multicore CPU such that the attained makespan is minimal. In computing the assignment, we must take into account the communication overhead caused by data transfers between the actors. The communication overhead can drastically increase the execution times of actors and outweigh the benefits obtained through parallel execution [27] Thus, it is important to incorporate communication overheads into the makespan to efficiently balance the workload of the processors.

In Figure 6.1(b), actor A_2 has the highest execution time of 25 time units among all actors and channel (A_2, A_3) has the highest communication overhead of 70 time units. Let us first demonstrate the case where we ignore the communication overhead and optimally balance the workload among processors based on the execution times of actors

[†]Given a stream graph (V, E), for any two actors $u, v \in E$, the communication cost $c_{uv} = 0$, if $u = v$ and $c_{uv} = c_{vu}$, for all $u, v \in V$.

```
void->void pipeline Program() {
  add A1();
  add A2();
  add A3();
  add A4();
}
void->int filter A1() {
  int x;
  init {x=0;}
  work push 1 {push (x++);}
}
int->int filter A2() {
  work push 6 pop 1 {
      pop();
      // do some work
      push(); // 6 times
  }
}
int->int filter A3() {
  work push 1 pop 6 {
      pop(); // 6 times
      // do some work
      push();
  }
}
int->void filter A4() {
  work pop 1 {print(pop());}
}
```



(a) (b)

Figure 6.1: (a) An example source code and (b) corresponding stream graph.

only. In Figure 6.2(a), actors A_1 and A_3 are assigned to processor 1, while actors A_2 and A_4 are placed on processor 2. Note that such actor placement is optimal: the attained run-time is 35 time units on both processors, which constitutes the makespan for this placement. However, the edges (A_1, A_2), (A_2, A_3), and (A_3, A_4) of the stream graph are all placed across processor boundaries. An additional communication overhead of $5 + 35 + 5 = 45$ time units is incurred by data transfers between actors, resulting in a worst-case makespan of 80 time units.

Let us compare the placement in Figure 6.2a to the optimal placement that can be obtained by considering both the actor execution times and the communication overhead. If we place actors A_1 and A_4 on processor 1 and actors A_2 and A_3 on processor 2, there are two edges (A_1, A_2) and (A_3, A_4) placed across processor boundaries that contribute 10 time units to the overall makespan (Figure 6.2b). Since both actors A_2 and A_3 are placed on the same processor, there is no extra communication overhead. The makespan attained for this placement is thus 55 time units, which is almost 1.5 times smaller than the makespan obtained for placement in Figure 6.2a.

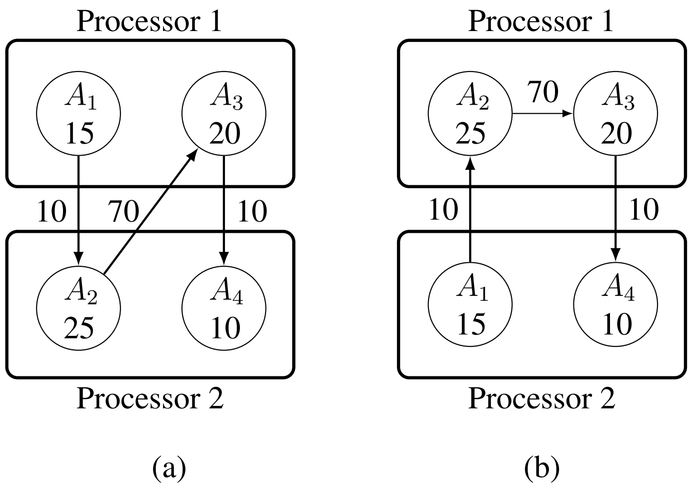


Figure 6.2: Actor placement: (a) considering actor execution times only (b) considering both actor execution times and communication overhead.

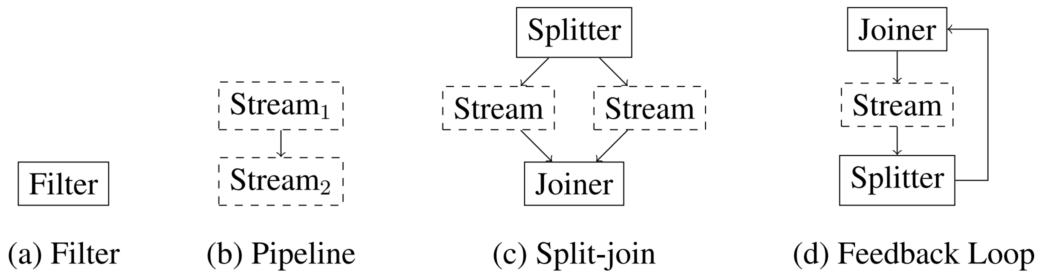


Figure 6.3: Composites of structured stream programs

6.3 Actor Placement Problem (APP)

A stream graph is defined by a tuple (V, E) where V is the set of actors and $E \subseteq V \times V$ is the set of channels, and $n = |V|$ and $m = |E|$. Execution time of an actor v is denoted by t_u and read and write communication costs of a data channel $(u, v) \in E$ is denoted by r_{uv} and w_{uv} respectively. We adopted the profiling scheme of [27] to measure execution times of actors, and read and write communication costs of edges. When $S \subseteq V$, execution times of set S is denoted by $t(S) = \sum_{v \in S} t_v$ by abuse of notation. Similarly, communication costs of two sets of actors $S_1, S_2 \subseteq V$ are defined by $c(S_1, S_2) = \sum_{(u,v) \in E \cap S_1 \times S_2} (r_{uv} + w_{uv})$ for all $u \in S_1$ and $v \in S_2$. If $u \in S_1, v \in S_2$ and $(u, v) \notin E$, then $r_{uv} = w_{uv} = 0$.

A placement is a partition of V into disjoint sets $S_1, \ldots, S_p$ ($\bigcup_i S_i = V$) where p is number of processors to utilize and S_i denotes set of actors placed on processor i. The run-time of processor i is denoted by r_i which is defined as,

$$r_i = t(S_i) + \frac{1}{2} c(S_i, V \setminus S_i). \tag{6.1}$$

Note that we assume that t_u of actor $u \in V$ and and communication cost of a channel $(u, v) \in E$ i.e., r_{uv} and w_{uv} are constant on all processors.

The goal of the actor placement problem is to minimize the run-time of the longest running processor (also known as *makespan*) $\Pi = \max_i r_i$.

6.3.1 NP-hardness of APP

Theorem 1. *The APP problem is NP-hard.*

Proof. We show the NP-hardness by reducing the partition problem ([SP12] in [29]) to the actor placement problem. An instance of the partition problem is given by a multiset A and a weight function $\omega : A \to \mathbb{N}$. The partition problem seeks for a subset A' such that $\omega(A') = \omega(A - A')$. We reduce an instance of the partition problem to a graph whose nodes represent the elements of A and the graph is edge free. The execution time function becomes the weight functions i.e., $t(a) = \omega(a)$ for all $a \in A$. Let $p = 2$ be the number of processors, hence, the APP problem reduces to:

$$\begin{aligned}\min_{S_1,S_2} \max(r_1, r_2) &= \min_{S_1,S_2} \max(t(S_1) + c(S_1, V \setminus S_1)\ ,\ t(S_2) + c(S_2, V \setminus S_2)) \\ &= \min_{A' \subseteq A} \max(\omega(A), \omega(A - A'))\end{aligned}$$

which resembles the partition problem where S_1 represents set A and S_2 represent set A' since S_1 and S_2 partition V into two disjoint sets. By minimizing the maximum cost cut, APP will attempt to *balance* the partition cost and it will output a partition if one exits, i.e., if $r_1 = r_2$. $\square$

Note that even for two processors (and no communication costs) the problem becomes NP-hard. For three or more processors, the APP problem is *strongly* NP-hard by reduction from the 3-partition problem. Given a set of $3n$ integers $\{a_1, ..., a_{3n}\}$ and a bound B such that $\sum_{i=1}^{3n} a_i = nB$, the goal is to partition these integers into n subsets of three integers such that each subset has a total of exactly B. This is problem [SP15] in [29]. To restrict APP to 3-partition, we set the execution time of each actor a_i, $\forall i \in \{1, ..., n\}$, to $\frac{B}{4} < t(a_i) < \frac{B}{2}$. We further assume that the graph is edge free and the number of actors is $n = 3k$. Note that all the communication cost coefficients are polynomially bounded in the input. Hence, the reduction establishes strong NP-hardness.

Thus to guarantee obtaining solution of AAP in quality bound within feasible time, we presents an *Oracle-based* approximation algorithm which uses binary search scheme. For each iteration of binary search, an Oracle subroutine determines whether an actor placement for a given makespan exists or not. Because the problem is NP-hard, a dynamic program is constructed to find an optimal solution in polynomial time by exploiting structural property of structured stream graphs.

6.4 Approximation

The goal of the approximation algorithm for the actor placement problem is to find a solution for an instance in polynomial time steps whose quality is bounded. The idea

Algorithm 1 Find Actor Allocation

```
Require: Stream Graph $G = (V, E)$, $UB$, $LB$
 1: Lower bound $LB \leftarrow 0$
 2: Upper bound $UB \leftarrow \sum_{u \in V} t(v)$
 3: while $UB - LB > \epsilon$ do
 4:     $\Pi \leftarrow \frac{LB+UB}{2}$
 5:     if ActorAllocationOracle($G, \Pi$) is YES then
 6:         $UB \leftarrow \Pi$
 7:     else
 8:         $LB \leftarrow \Pi$
 9:     end if
10: end while
11: return $UB$
```

of the proposed approximation algorithm is to use a binary search scheme, *Find Actor Allocation* (Algorithm 1) for the makespan Π which is given to a subroutine, *Actor Allocation Oracle* (Algorithm 2) as a criterion to determine whether there exists a placement for makespan Π.

The upper bound for the binary search is initially set to $t(V)$, which represents the total execution time for all actors placed on a *single* processor. The initial bound obviously corresponds to a feasible solution for the subroutine *Actor Allocation Oracle*. Assuming that all execution times are positive, if the *Actor Allocation Oracle* finds a valid actor allocation for a given makespan (i.e., answer *YES*), then the *Find Actor Allocation* routine searches the lower half range in the next binary search iteration. If the *Actor Allocation Oracle* fails to find a valid actor allocation for a given makespan (i.e., answer *NO*), on the other hand, then the next binary search iteration is performed on the upper half range. Binary search continues until difference of upper bound and lower bound becomes smaller than a threshold parameter ϵ which is proportional to $t(V)$ i.e., $0.001 \times t(V)$.

The subroutine *Actor Allocation Oracle* initially assumes an empty list of disjoint sets $S = [S_1, \ldots, S_p]$ where p represents number of processors to utilize. Then the subroutine starts allocating actors on each processor until all actors are allocated. For each actor allocation iteration for a processor i where $1 \le i \le p$, the subroutine *Actor Allocation Oracle* calls another subroutine *PackP* (line 5 in Algorithm 2) and accumulates the result set of the subroutine *PackP* on S_i. The subroutine *PackP* solves a discrete optimization problem and returns an optimal set X of actors, which is subset of set V. The optimal set X is computed based on dynamic programming that maximizes the number of newly allocated actors on processor i by still adhering to the makespan constraints i.e., $r_i \le \Pi$. The set X is defined by,

$$\begin{aligned} \max. \quad & |X \cap U| \\ \textit{s.t.} \quad & t(X) + c(X, V \setminus X) \le \Pi \\ & X \subseteq V, \end{aligned} \tag{6.2}$$

where set U denotes a set of actors that are not assigned on S yet.

Lemma 6. *After the ℓ-th iteration of the while-loop in Algorithm 2, $r_i \le \ell \cdot \Pi$ holds for all processors $i \in \{1, \ldots, p\}$.*

Proof. For the first iteration of the while-loop (line 3 in Algorithm 2), the value of $r_i = t(S_i) + \frac{1}{2}c(S_i, V \setminus S_i)$ is at most Π for each iteration i of the outer for-loop (line 4 in Algorithm 2). Each cut returned by the *PackP* subroutine is guaranteed to have $r_i \le \Pi$ due to the makespan constraint. Reassigning actors in the inner for-loop (line 6 in Algorithm 2) does not increase the value of r_i for either S_i or S. The set of actors that can be reassigned in each iteration of the while loop increases by at most Π. Hence, for the ℓ-th iteration of the while-loop, the value of r_i for each S_i is at most $\ell \cdot \Pi$. $\square$

Lemma 7. *If there exists a placement for makespan Π, then in each iteration of the while-loop in Algorithm 2 the number of unallocated actors halves at least.*

Proof. For the ℓ-th iteration of the while-loop where $\ell > 0$ and $U_0 = V$, let $U_{\ell-1}$ be the set of unassigned actors at the beginning of the while-loop iteration and U_ℓ be the set of unassigned actors at the end of the while-loop iteration. We claim that $|U_\ell| \le \frac{|U_{\ell-1}|}{2}$.

Algorithm 2 Actor Allocation Oracle

```
Require: Stream Graph $G = (V, E)$, Makespan $\Pi$
 1: $(S_1, \ldots, S_p) \leftarrow (\emptyset, \ldots, \emptyset)$
 2: $U \leftarrow V$
 3: while $U \neq \emptyset$ do
 4:     for all $i \in \{1, \ldots, p\}$ do
 5:         $X \leftarrow$ PackP$(G, U, \Pi)$
 6:         for all $j \in \{1, \ldots, p\} \setminus \{i\}$ do
 7:             $I_j \leftarrow X \cap S_j$
 8:             if $c(I_j, S_j \setminus I_j) < c(I_j, X \setminus I_j)$ then
 9:                 $S_j \leftarrow S_j \setminus I_j$
10:             else
11:                 $X \leftarrow X \setminus I_j$
12:             end if
13:         end for
14:         $S_i \leftarrow S_i \cup X$
15:         $U \leftarrow U \setminus X$
16:     end for
17:     if $\max_{i \in \{1,\ldots,k\}} r_i > \Pi \cdot \log_2 n$ then
18:         return NO
19:     end if
20: end while
21: return YES
```

For the ℓ-th iteration of the while-loop, let $U_{\ell i}$ denote number of unallocated actors at the end of the i-th iteration of the outer for-loop where $1 \leq i \leq p$ and $U_{\ell 0} = U_\ell$. We define $X_{\ell i} = U_{\ell(i-1)} - U_{\ell i}$, which means the set of newly allocated actors determined in the i-th iteration of the outer for-loop.

The *PackP* subroutine returns an optimal actor assignment such that $|X_{\ell i}| \geq |U_{\ell i}|$. Summing over all iterations of the outer for-loop, we obtain:

$$
\begin{aligned}
|U_{\ell-1}| - |U_\ell| &= \sum_{1\le i\le p} |U_{\ell(i-1)}| - \sum_{1\le i\le p} |U_{\ell i}| \\
&= \sum_{1\le i\le p} |X_{\ell i}| = |X_\ell| \\
&\ge \sum_{1\le i\le p} |U_{\ell i}| = |U_\ell|
\end{aligned}
$$

□

Theorem 2. *The algorithm is a* $\log_2 n$*-approximation.*

Proof. By Lemma 7, the algorithm should terminate in at most $\log n$ iterations of the while-loop. Combining this bound with Lemma 6 concludes the proof. □

6.5 Dynamic Program for *PackP* Subroutine

To reduce problem solving time of *PackP*, we exploit structured property of structured stream graphs. Instead of searching solution on a structured stream graph, we convert the structured stream graph into a binary tree decomposition which is known to speed up problem solving time of linear systems [53].

Section 6.5.1 defines structured stream graphs in binary tree decomposition approach, and Section 6.5.2 describes dynamic program for *PackP* that returns an optimal actor assignment under given conditions. Because LaminarIR does not implement decorative nodes of structured stream graphs such as splitters and joiners, Section 6.5.3 describes how to convert structured stream graphs into reduced stream graph without splitters and joiners, and shows how the dynamic programming can be expanded and applied on a corresponding reduced stream graph. Section 6.6 proves the problem solving time of *PackP* is polynomial due to binary tree decomposition.

6.5.1 Structured Stream Graphs

To get a handle on the actor placement problem we restrict the graphs to structured stream graphs as used in stream languages such as StreamIt [3]. Note that structured

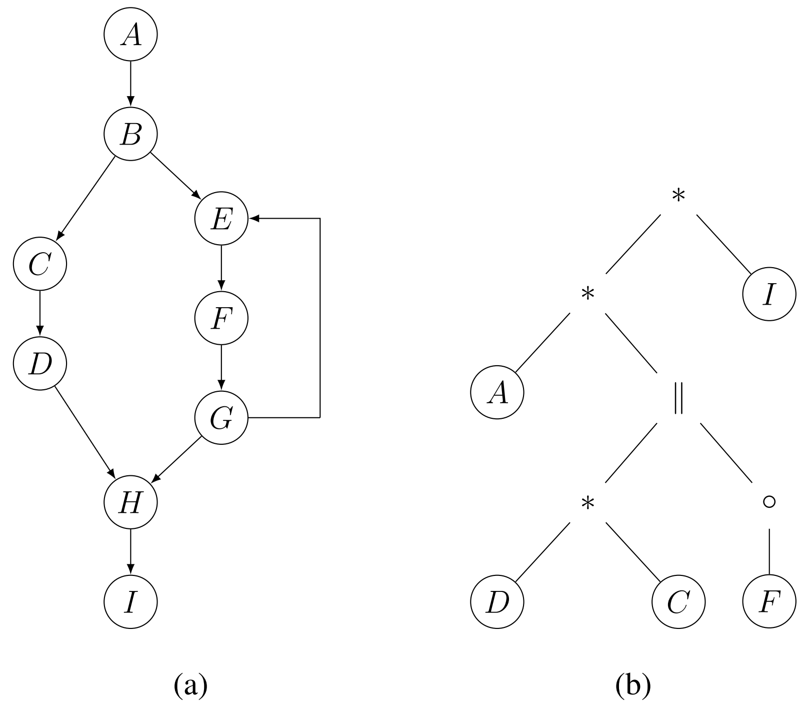


Figure 6.4: (a) An example structured stream graph and (b) corresponding binary tree decomposition.

stream graphs have a strong relationship to series-parallel graphs. It can be shown that structured stream graphs can be embedded in the class of series-parallel graphs.

A compositional stream is composed of composites as shown in Figure 6.3. The composites are either (1) an actor called *filter* that is a single discrete computational unit which filters data from one input channel and writes to one output channel, (2) a *pipeline*, which composes substreams in sequence, where the output of one substream becomes the input of another substream, (3) a *split-join*, where input data is split and given to multiple substreams to be consumed, the output of all these substreams are joined together and passed on as the split-join's output, and (4) a *feedback loop*, where the output of a substream is split and passed back to be input again to the substream. Split-joins and feedback loops are constructed using special actors called *splitters* and *joiners* that either splits the tokens of a communication channel or joins them.

Let $\mathbb{G}$ denote a class of compositional stream graphs. A compositional stream graph $G \in \mathbb{G}$ is specified by a tuple (V, E, s, e). V and E denote sets of actors and edges respectively and $s, e \in V$ are start and end nodes which are the linking points of the composite to the other composites. Definition 6 defines rules to denote each composite.

Definition 6. *The class of compositional stream graphs $\mathbb{G}$ is inductively defined:*

(R1) If $u \in V$, then a filter composite of u is denoted by

$$G = (\{u\}, \emptyset, u, u) \in \mathbb{G}.$$

(R2) If $G_1 = (V_1, E_1, s_1, e_1)$ and $G_2 = (V_2, E_2, s_2, e_2)$ are in $\mathbb{G}$ and V_1 and V_2 are disjoint, then a pipeline composite of two graphs are denoted by

$$G_1 \, * \, G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(e_1, s_2)\}, s_1, e_2) \in \mathbb{G}.$$

(R3) If $G_1 = (V_1, E_1, s_1, e_1)$ and $G_2 = (V_2, E_2, s_2, e_2)$ are in $\mathbb{G}$, and V_1, V_2, $\{s, e\}$ (where s and e are labels) are disjoint, then a split-join composite of two graphs are denoted by

$$G_1 \parallel G_2 = (V_1 \cup V_2 \cup \{s, e\}, E_1 \cup E_2 \cup \{(s, s_1), (s, s_2), (e_1, e), (e_2, e)\}, s, e) \in \mathbb{G}.$$

(R4) If $G_1 = (V_1, E_1, s_1, e_1) \in \mathbb{G}$ and $\{s, e\}$ is disjoint from V_1, then a feedback composite of G_1 is denoted by

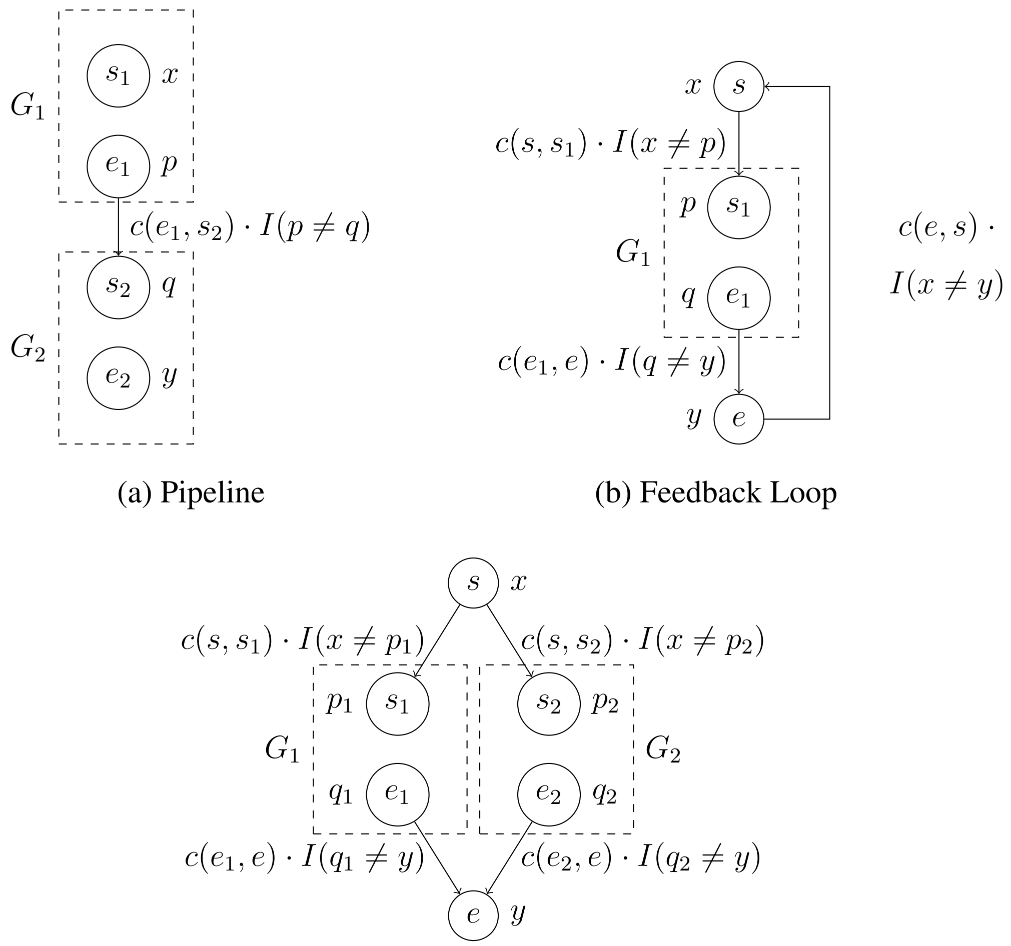
$$\circ G_1 = (V_1 \cup \{s, e\}, E_1 \cup \{(s, s_1), (e_1, e), (e, s)\}, s, e) \in \mathbb{G}.$$

(R5) Nothing else is in $\mathbb{G}$

6.5.2 Dynamic Programming Model

In this section, we transform *PackP* into an equivalent problem that is solved using dynamic programming. We first establish cost functions for each components of structured stream graphs and construct a dynamic program for the *PackP* subroutine, to demonstrate the *PackP* is equivalent to the suggested dynamic program.

We introduce a recursive function $h(G, x, y, k)$ whose first parameter is a structured stream graph $G = (V, E, s, e) \in \mathbb{G}$. G is constructed by a stream graph (V, E) that


G_1
s_1
x
e_1
p
$c(e_1, s_2) \cdot I(p \neq q)$
G_2
s_2
q
e_2
y
x
s
$c(s, s_1) \cdot I(x \neq p)$
p
s_1
G_1
q
e_1
$c(e, s) \cdot$
$I(x \neq y)$
$c(e_1, e) \cdot I(q \neq y)$
y
e
s
x
$c(s, s_1) \cdot I(x \neq p_1)$
$c(s, s_2) \cdot I(x \neq p_2)$
p_1
s_1
s_2
p_2
G_1
G_2
q_1
e_1
e_2
q_2
$c(e_1, e) \cdot I(q_1 \neq y)$
$c(e_2, e) \cdot I(q_2 \neq y)$
e
y

(a) Pipeline

(b) Feedback Loop

(c) Split-join

Figure 6.5: Dynamic program for composites

Table 6.1: Example of tabularized values of the objective function of *PackP*.

x	y	k		
		0	...	n
0	0		...	
0	1		...	
1	0		...	
1	1		...	

is given to *PackP* as a parameters. The second and third binary parameters x and y, indicate whether the start and end nodes, s and e respectively, should be packed as a result of *PackP*. The last parameter k is a integral value between 0 and n tabularizing the values of the objective function of *PackP*. The function h finds the minimal run-time for a single processor such that the total number of allocated actors is exactly k. For instance, in Figure 6.1(b), the minimal run-time returned by h such that $k = 2$ is 25 (i.e., if actors A_1 and A_4 are packed and no intra-processor communication cost occurs from actor A_1 and to actor A_4).

For every call $h(G, x, y, k)$, the dynamic algorithm constructs a table containing the minimal run-time given the input k and the information about whether the start and end nodes of G are packed.

Table 6.1 shows an example of tabularized values of the objective function of *PackP*. The rows of the table correspond to x and y values (i.e., there are 4 rows) and the columns represent k values ranging from 0 to n. The algorithm starts with the tables for the actors and then proceeds to generate tables for the composites. A complete set of rules for the dynamic program is given below:

Definition 7. The dynamic program is defined as a function $h : \mathbb{G} \times \mathbb{B} \times \mathbb{B} \times \mathbb{N} \to \mathbb{R}$ such that:

(R1) If $G = (\{u\}, \emptyset, u, u)$ is an actor, then

$$h(G,x,y,k) = \begin{cases} 0, & \text{if } x = y = 0, k = 0 \\ t(u), & \text{if } x = y = 1, u \notin U, k = 0, \\ t(u), & \text{if } x = y = 1, u \in U, k = 1, \\ \infty, & \text{otherwise.} \end{cases}$$

(R2) If $G = (V, E, s, e)$ is a pipeline, i.e.,
$G = G_1 * G_2 = (V_1, E_1, s_1, e_1) * (V_2, E_2, s_2, e_2)$ and $s = s_1, e = e_2$, then

$$\begin{aligned} h(G,x,y,k) = \min\{ & h(G_1, x, p, k_1) + h(G_2, q, y, k_2) \\ & + \frac{1}{2} c(e_1, s_2) \cdot I(p \neq q) \\ & : k = k_1 + k_2,\ p, q \in \{0,1\}\} \end{aligned}$$

where $I(p \neq q)$ is the indicator function of condition $p \neq q$ and it is 1 if the condition holds; 0 otherwise.

(R3) If $G = (V, E, s, e)$ is a split-join, i.e.,
$G = G_1 \parallel G_2 = (V_1, E_1, s_1, e_1) \,||\, (V_2, E_2, s_2, e_2)$, then

$$\begin{aligned} h(G,x,y,k) = \min\{ & h(G_1, p_1, q_1, k_1) + h(G_2, p_2, q_2, k_2) + x \cdot t(s) + y \cdot t(e) \\ & + \frac{1}{2}(c(s, s_1) \cdot I(x \neq p_1) + c(s, s_2) \cdot I(x \neq p_2) + c(e_1, e) \cdot I(q_1 \neq y) + c(e_2, e) \cdot I(q_2 \neq y)) \\ & : k = k_1 + k_2 + \delta_s(x) + \delta_e(y),\ p_1, q_1, p_2, q_2 \in \{0,1\}\} \end{aligned}$$

where $\delta_u(x) = \begin{cases} x, & \text{if } u \in U, \\ 0, & \text{otherwise.} \end{cases}$

(R4) If $G = (V, E, s, e)$ is a feed-back loop, i.e.,
$G = \circ G_1 = \circ(V_1, E_1, s_1, e_1)$, then

$$\begin{aligned} h(G,x,y,k) = \min\{ & h(G_1, p, q, k_1) + x \cdot t(s) + y \cdot t(e) \\ & + \frac{1}{2}(c(s, s_1) \cdot I(x \neq p) + c(e_1, e) \cdot I(q \neq y) + c(e, s) \cdot I(x \neq y)) \\ & : k = k_1 + \delta_s(x) + \delta_t(y), p, q \in \{0,1\}\} \end{aligned}$$

where $\delta_u(x) = \begin{cases} x, & \text{if } u \in U, \\ 0, & \text{otherwise.} \end{cases}$

For an actor u in (R1), the start node corresponds to the end node. The function h returns the execution time of u if $x = y = 1$ indicating that the actor is packed. If there is no feasible solution, i.e., $x \neq y$, the function returns ∞. The k parameter denotes the number of previously unassigned nodes; it is set to 1 if u has not been already packed.

In case of a pipeline (R2), the indicator function $I(\cdot)$ denotes whether the end node of G_1 and the start node of G_2 are assigned to the same processor (Figure 6.5(a)). We choose nodes such that the sum of costs for the substreams G_1 and G_2 and the communication cost between them is minimal.

For a split-join (R3), we similarly minimize the sum of costs between substreams G_1 and G_2 and the cost of communication with the splitter and joiner nodes s and e (Figure 6.5(c)). Note that unlike the pipeline case, there are additional execution costs associated with the splitter and joiner nodes represented by $t(s)$ and $t(e)$.

For a feedback loop (R4), we choose p and q so that the cost of G_1 with additional execution costs for the splitter and joiner nodes and the cost of communication between the pairs of actors (s, e), (s, s_1) and (e_1, e) is minimal (Figure 6.5(b)).

To prove the proposed dynamic program is equivalent to *PackP*, we first establish definitions as following.

Definition 8. *Given $\Pi \in \mathbb{R}$ and $0 \leq k \leq n$, let H_Π and F_k be the two sets formed as follows:*

$$H_\Pi = \{X \subseteq V : \hat{h}(X) \leq \Pi\} \tag{6.3}$$

$$F_k = \{X \subseteq V : f(X) = k\} \tag{6.4}$$

$$\begin{aligned} \tilde{h}(k) &= \min_{X \subseteq V} \{\hat{h}(X) : |X \cap U| = k\} \\ &= \min_{X \subseteq F_k} \hat{h}(X) \end{aligned} \tag{6.5}$$

where

$$\hat{h}(X) = t(X) + c(X, V \setminus X)$$

denoting the cost function for assigning actors of a stream graph to a processor of *PackP* in Equation 6.2, and function f that returns value of the objective function of *PackP*, i.e., $|X \cap U|$.

Note that the set H_Π corresponds to the feasible region of *PackP*. We introduce $\tilde{h}(k)$ that is computed efficiently via the proposed dynamic program.

Lemma 8. *The intersection of F_g and H_Π is non-empty iff* $\min_{X \in F_g}\{\hat{h}(X)\} \leq \Pi$.

Proof.

$$\begin{aligned}
\min_{X \in F_g}\{\hat{h}(X)\} \leq \Pi &\iff \forall X \in \arg\min_{X \in F_g}\{\hat{h}(X)\} : \hat{h}(X) \leq \Pi \\
&\iff \arg\min_{X \in F_g}\{\hat{h}(X)\} \cap H_\Pi \neq 0 \\
&\iff \arg\min_{X \in F_g \cap H_\Pi}\{\hat{h}(X)\} \\
&\iff F_g \cap H_\Pi \neq 0
\end{aligned}$$

□

Lemma 9. *The value of* $M_{f^*} = \max_g\{g : \tilde{h}(g) \leq \Pi\}$ *is the optimal solution of* PackP.

Proof. We claim that *PackP* maximizes the number of unallocated actors $U \subseteq V$ to be placed on a processor such that the total run-time does not exceed Π.

$$\begin{aligned}
M_{f^*} &= \max_g\{g : \min_{X \subseteq V}\{\hat{h}(X) : f(X) = g\} \leq \Pi\} \\
&= \max_g\{g : F_g \cap H_\Pi \neq 0\} \\
&= \max_{X \subseteq V}\{f(X) : X \in H_\Pi \neq 0\} \\
&= \max_{X \subseteq V}\{f(X) : \hat{h}(X) \leq \Pi\}
\end{aligned}$$

□

In Definition 6.5.2, we claimed that $\tilde{h}(k) = \min_{X\subseteq V}\{\hat{h}(X) : |X \cap U| = k\}$, which corresponds to $\min_{x,y} h(G, x, y, k)$. In order to establish the claim, we further show by structural induction, that for all graphs $G \in \mathbb{G}$, the following holds:

$$h(G, x, y, k) = \min_{X\subseteq V}\{\hat{h}(X) : |X \cap U| = k\}.$$

Proof. By structural induction

- If $G = (\{u\}, \emptyset, u, u)$ is an actor, then

$$h(G, x, y, k) = \min\{\hat{h}(X) : f(X) = k\}.$$

- If $G = (V, E, s, e)$ is a pipeline, i.e.,
  $G = G_1 * G_2 = (V_1, E_1, s_1, e_1)\ *\ (V_2, E_2, s_2, e_2)$, then

$$\begin{aligned}
h(G, x, y, k) = &\min\{\min_{X\subseteq V_1\setminus\{s_1,e_1\}}\{\hat{h}(X) : f(X) = k_1\} \\
&+ \min_{X\subseteq V_2\setminus\{s_2,e_2\}}\{\hat{h}(X) : f(X) = k_2\} \\
&+ \frac{1}{2}c(e_1, s_2)\cdot I(p \neq q) : k = k_1 + k_2,\ p, q \in \{0, 1\}\} \\
= &\min\{\min_{X\subseteq V_1\cup V_2}\{\hat{h}(X) : f(X) = k\} : k = k_1 + k_2\} \\
= &\min\{\hat{h}(X) : f(X) = k\}.
\end{aligned}$$

- If $G = (V, E, s, e)$ is a split-join, i.e.,
  $G = G_1 \parallel G_2 = (V_1, E_1, s_1, e_1) \parallel (V_2, E_2, s_2, e_2)$, then

$$\begin{aligned}
h(G, x, y, k) = &\min\{\min_{X\subseteq V_1}\{\hat{h}(X) : f(X) = k_1\} + \min_{X\subseteq V_2}\{\hat{h}(X) : f(X) = k_2\} \\
&+ x\cdot t(s) + y\cdot t(e) \\
&+ \frac{1}{2}(c(s, s_1)\cdot I(x \neq p_1) + c(s, s_2)\cdot I(x \neq p_2) \\
&+ c(e_2, e)\cdot I(q_1 \neq y) + c(e_2, e)\cdot I(q_2 \neq y)) \\
&: k = k_1 + k_2 + \delta_s(x) + \delta_e(y),\ p_1, q_1, p_2, q_2 \in \{0, 1\}\}.
\end{aligned}$$

- If $G = (V, E, s, e)$ is a feed-back loop, i.e., $G = \circ G_1 = \circ(V_1, E_1, s_1, e_1)$, then

$$\begin{aligned} h(G, x, y, k) = \min\{ & h(G_1, p, q, k_1) + x \cdot t(s) + y \cdot t(e) \\ & + \frac{1}{2}(c(s, s_1) \cdot I(x \neq p) + c(e_1, e) \cdot I(q \neq y) + c(e, s) \cdot I(x \neq y)) \\ & : k = k_1 + k_2 + \delta_s(x) + \delta_t(y), p, q \in \{0, 1\}\}. \end{aligned}$$

$\square$

6.5.3 Reduced Stream Graphs for LaminarIR

Communication cost aware orchestration algorithm is essential to determine best parallel schedule for LaminarIR. However, the proposed communication cost aware orchestration algorithm needs adjustments to be applied on LaminarIR. LaminarIR is designed for general graphs therefore not restricted to structured stream graphs. With structured stream programs, LaminarIR eliminates data-flows non-computational nodes such as nodes only for data distribution or merge, which effects as destructing of structured stream graphs. Thus, orchestration information based on structured stream graphs is necessary to be converted into general stream graphs forms. We call *reduced stream graph* for graphs generated by eliminating nodes for data distribution and merge from structured stream graphs.

Figure 6.6 shows an example of a reduced stream graph after eliminating splitter (node D) and joiner (node C) from a structured stream graph. As shown in the example, topology of reduced stream graph (Figure 6.6b) represents direct memory access pattern of LaminarIR which is not represented in the structured stream graph (Figure 6.6a). This LaminarIR-friendly topology of reduced stream graph enables to 1) profile data communication overheads of LaminarIR in higher precision and 2) compute actual data communication cost of LaminarIR when a parallel schedule is given.

To maintain semantics of structured stream programs in reduced stream programs, data rotation orders which has been implicated in the definition of splitters and joiners in structured stream programs have to be incorporates into scheduling information of

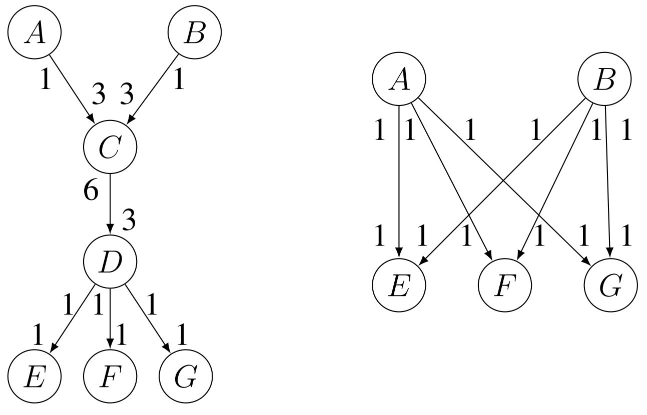


Figure 6.6: (a) An example structured stream graph and (b) corresponding reduced stream graph.

reduced stream program. For example, the defined data rate of the joiner (actor C) and the splitter (actor D) implies that the joiner consumes three incoming data once per edge in round-robin way and splitter produces one outgoing data once per edge in round-robin way. Therefore, one of the valid steady-state schedules of Figure 6.6a is

$$3A3BC2D3(EFG).$$

Implied distribution/merge information in splitters and joiners are eliminated as well when structured stream programs are transformed into reduced stream graphs. Thus, reduced stream program needs to extend scheduling information to retain the implication, for example,

$$(A, E, 1)(A, F, 1)(A, G, 1)(B, E, 1)(B, F, 1)(B, G, 1).$$

A schedule of reduced stream program is represented by a list of tuples. Each tuple consists of three elements, where the first and second element of the list indicate producer and consumer respectively followed by the third element which means data rate per transfer.

6.6 Time Complexity

Now let us analyze the running time of the proposed dynamic programming algorithm and the overall running time of the Algorithm 2. A stream graph has at most $2n - 1$ edges represented by actors and pipelines. Hence, the decomposition binary tree has at most $2n - 1$ leaf nodes and the total number of nodes is thus bounded by $4n - 3$. For leaf nodes and nodes that represent split-join and feed-back loop compositions in the binary tree, the number of values of the objective function that need to be computed is $\mathcal{O}(p^2)$, where $p = |P|$. For nodes that represent pipeline compositions, the number of computations is $\mathcal{O}(p^3)$. Since the number of nodes in the decomposition tree is linear, the overall running time of the *PackP* subroutine is $\mathcal{O}(np^3)$. Note that in case of split-join nodes with $d > 2$ streams, the decomposition tree is not binary but can be converted to a binary tree without affecting the solution. For every such split-join node, we replace it with a complete binary subtree T of height $\lceil \log_2 d - 1 \rceil$ rooted at r and attach d nodes of the split-join node as children of T at most 2 per each node. We assign $t(r)$ the local cost of the split-join node and for every other internal node u in this subtree we set $t(u) = 0$. The edge costs between the internal nodes are set to 0, while the costs on the edges to the attached children are set to the costs associated with the split-join node. By Lemma 7, Algorithm 2 terminates in at most $\log_2 n$ iterations. Thus, the overall running time is bounded by $\mathcal{O}(np^3 \log_2 n)$, where n is number of nodes in a graph and p is the number of processors to utilize.

6.7 Experimental Results

We evaluate our method as follows:

- Quality bounds of approximated solution compared to the optimal solution.
- Efficiency of approximation algorithm in problem solving time compared to the ILP program.
- Effectiveness of theoretically computed solutions in real applications.

In this experiment we have used an Intel Xeon E5-2697 running a standard build of the Centos 7.2 Linux distribution (kernel version 3.10.0). As a benchmark suite for our scheduling algorithm, we have used 6 applications from the StreamIt [77]. All benchmarks are a part of the StreamIt benchmark suite publicly available from [3] and vary from digital signal processing, video processing, and linear algebra, to sorting. The information about the actors in each benchmark is provided in Table 6.2. Besides parametric specifications of benchmarks in column 2, we provide two additional benchmark specific characteristics which we observe to have high correlation to the effectiveness of the proposed communication-cost-aware orchestration. Column 3 in Table 6.2 shows the number of nodes related to data distribution and collection (split-joins), which indicate complexity of stream-graph topology. Column 4 shows the proportional overhead of communication overhead over the computational overhead. Higher proportion of splitters and joiners indicates nodes in a benchmark share many data channels with each other, which means higher data dependencies among nodes. Higher communication cost ratio over computation cost means a benchmark has higher chance to be improved by communication-cost-aware orchestration than computation only orchestration.

Table 6.2: Benchmark characteristics

Benchmark	Parameters	SJs	$\frac{\text{comm.}}{\text{comp.}}$ ratio
RadixSort	number of values to sort (16)	0	33.81%
DCT	window size (16)	4	337.61%
FFT	window size (64)	2	121.49%
FMRadio	bands (7); window size (128); decimation (4)	10	63.76%
MergeSort	number of values to sort (16)	14	62.75%
MatrixMult	matrix dims NxM, MxP (12x12, 9x12)	10	21.60%
BeamFormer	channels (15)	4	23.61%

The approximate solutions of all instances are bounded by $\log_2 n$ (cf. Section 6.4), where n is the number of actors. However, this bound is a worst-case bound, and we are able to observe better bounds for concrete instances. Note that the approximation ratio of an instance is bounded by the number of iterations of the `while`-loop in Algorithm 2. Each loop iteration penalizes the solution by at most Π as shown in Lemma 6.

Table 6.3: Approximate vs. optimal results. The number of actors in each benchmark is indicated by n, while $|E|$ represents the number of communication channels in the stream graph. Note that all $\frac{APX}{OPT}$ values are bounded by $\log_2 n$.

Benchmark	$\lvert V\rvert$	$\lvert E\rvert$	APX/OPT		
			2 Cores	4 Cores	6 Cores
RadixSort	13	12	1.157	1.368	1.182
DCT	24	37	1.407	1.577	0.953
FFT	26	26	1.386	2.096	1.491
FMRadio	31	37	1.437	2.586	2.085
MergeSort	31	37	1.328	1.337	N/A
MatrixMult	52	88	1.000	0.984	1.000
BeamFormer	58	71	1.051	N/A	N/A
Average			1.252	1.658	1.342

In Table 6.3, we show the result of the optimal solution (OPT) compared to the approximation (APX) for 2, 4 and 6 processors. The optimal solution was computed using the ILP program given in the Appendix (Equation A.1). The ILP program is written in IBM ILOG AMPL [38] and IBM ILOG CPLEX Interactive Optimizer 12.6.3.0 [39] is used as a solver. On average, quality bound of our approximation algorithm were in 25.2% with 2 processors, in 65.8% with 4 processors, and in 34.2% with 6 processors. Our approximation solutions are closer to optimal solutions when benchmarks have equable execution times among nodes, e.g., RadixSort, MatrixMult, and BeamFormer. Our approximation algorithm solved better solution than the ILP in some cases such as DCT on 6 processors and MatrixMult on 4 processors. It is because the ILP uses branch&bound techniques compute optimal solutions with specific threshold to termi-

Table 6.4: AAP and ILP solving times in seconds for 2, 4, and 6 processors

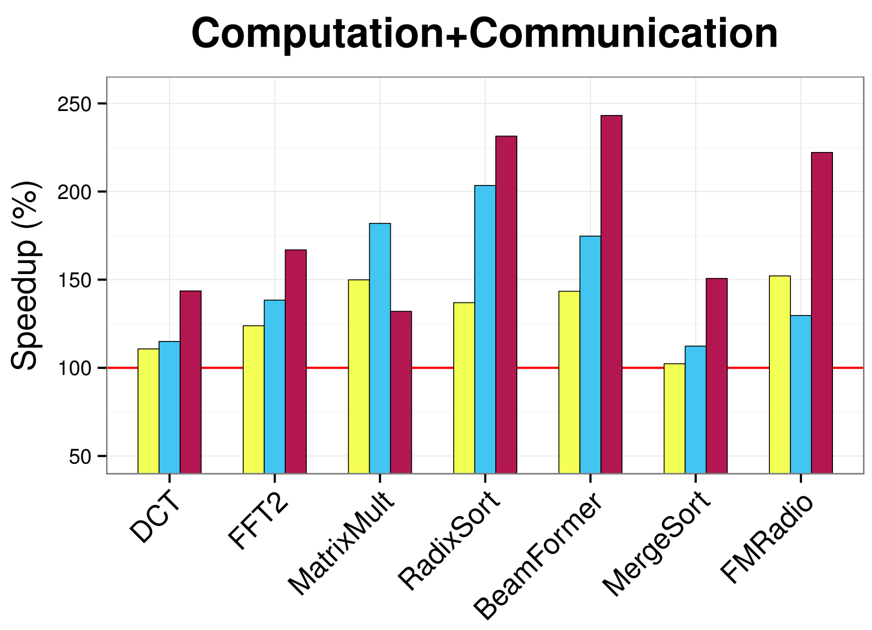
Benchmark	$\|V\|$	$\|E\|$	2 Cores		4 Cores		6 Cores	
			AAP	ILP	AAP	ILP	AAP	ILP
RadixSort	13	12	0.238	0.190	0.235	1.183	0.244	0.977
DCT	24	37	0.753	0.537	0.724	33.490	1.002	25.808
FFT	26	26	1.905	0.705	2.888	7.492	2.163	4.851
FMRadio	31	37	1.487	1.697	1.583	19.903	1.151	144.285
MergeSort	31	37	2.382	1.245	2.690	35.405	2.816	N/A
MatrixMult	52	88	3.178	7.923	3.580	50.721	8.773	86.340
BeamFormer	58	71	11.251	26.614	11.706	N/A	10.846	N/A

nate. With an increasing number of processors, the ILP program becomes intractable. The entries marked by 'N/A' in Table 6.3 indicate that the ILP did not provide *any* solution after a preset timeout, 10,000 seconds.

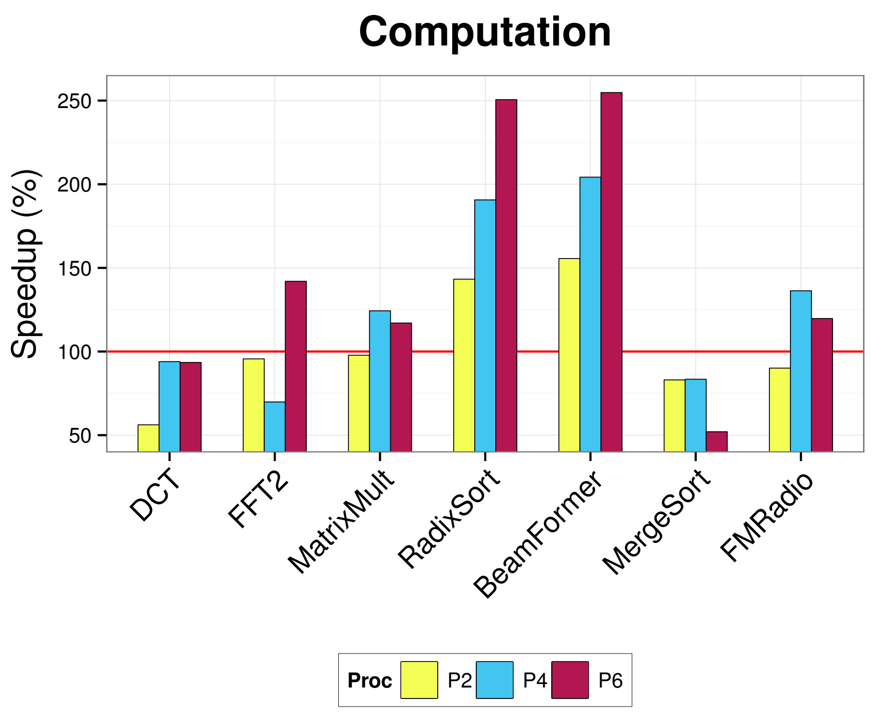
Table 6.4 shows actual running time of our approximation algorithm and the ILP program. Problem solving time of the ILP increases drastically as the number of processors grows. With 6 processors, two out of seven benchmarks failed to be solved by the ILP program in 10,000 seconds. In contrary, problem solving time of our approximation algorithm which does not increase by growing number of processors.

In Figure 6.7a, we present the speedups achieved by our communication-cost-aware approximation algorithm over single processor execution for 2, 4, and 6 processors. We obtain average speedups of 1.31x, 1.50x and 1.84x on 2, 4 and 6 processors respectively. Some benchmark instances do not scale well due to *bottleneck* actors, i.e., actors that constrain the throughput of a stream program. For example, MatrixMult achieves lower speedups than other benchmarks as the number of processors increases, even though the actor placement for these benchmarks is optimal for 6 cores (see Table 6.3).

Figure 6.7b shows experimental result of communication-cost-insensitive scheduling for comparison. For two benchmarks, communication-cost-insensitive scheduling show worse performance than sequential program, and communication-cost-insensitive



(a) Communication-cost-aware scheduling



(b) Computation-only scheduling

Figure 6.7: Speedup comparison of (a) communication-aware scheduling with (b) computation-only scheduling

scheduling does not show gradual performance improvement over increasing number of processors for four benchmarks. Such problem happens more frequently when execution times of actors in a program are similar and topology of the stream-graph is complex. The most critical problem of the communication-cost-insensitive scheduling is that the quality of schedules are highly dependent on the fluctuation rate of the performance measurement. If actor execution times are very much similar and fluctuation rate of a testing machine is very high e.g., memory structure of the machine is complex, then result of actor allocation can be arbitrary, and so is the schedule.

Another noticeable observation is that RadixSort and BeamFormer show better performance with communication-cost-insensitive scheduler. As shown in Table 6.2, RadixSort and BeamFormer have very simple stream-graph topology, and communication cost overhead is relatively small compared to the computation overhead as shown in Table 6.2. In such case, benefit of communication-cost-aware scheduling become less effective and communication-cost-insensitive scheduler results more load-balanced schedule as it has less objectives to meet. Considering complexity of the stream-graph topology as a factor of scheduler would enhance scheduling quality, and this is one of the future works.

Chapter 7

Related Work

Unlike classical data-flow [24], SDF [51] graph is a data-flow graph which fixes the number of tokens produced and consumed by an actor at compile-time. The static nature of SDF facilitates compile-time optimizations wrt. streamgraph transformations, partitioning and scheduling. A number of contemporary stream languages have been adapted SDF as their computation model for aforementioned properties.

There exists a large body of work on stream programming languages and related compilation techniques, including StreamIt [77], Baker [19], Brook [17], Cg [55], CQL [7], Lime [8] and StreamFlex [71].

7.1 Compiler Optimizations to Overcome FIFO Queue Overhead

All modern stream programming languages except Cg implement data channels based on queues. Systems based on the streaming model such as Borealis [4], Flextream [37] and DANBI [58] 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. [12] 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. [13] introduced Gabriel, a design environment for digital signal processing, which employs symbolic names for tokens across actors. Gabriel does not model global data-flow, and data is flushed to memory after each actor invocation. Sermulins et al. [69] 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. Soulé et al. [70] and Bosboom et al. [15] described actor fusion techniques to remove inter-buffering overhead, but they did not remove queues as such as the communication mechanism between actors.

7.2 Static Analysis of Stream Programs

Majority of research on synchronous data-flow program optimizations are in the area of graph topology [35]. Hirzel et al. surveys topological optimizations that can enhance utilization of traditional compiler optimizations, and also stresses the necessity of new compiler analysis for stream programs. Our approach is distinctive from the proposed topological optimizations as any actor code is eligible to our approach.

The latest StreamIt system [69, 77] supports program-wise scalar replacement by fusing actors, only if the buffer in between two actors does not contain delay tokens, as well as tokens to peek exceeding consumption rate, and pre-loaded tokens to execute stream graph cycles. Our method separates actor-wise and program-wise scalar replacement, thus provides a modularized view of data token flows. Hence, we can handle SDF graphs with cycles and can treat them, soundly. StreamIt's scalar replacement is not sound when queue operations are involved with unbalanced conditional branches, and actors with static stream parameters in form of constant arrays.

Various researches have been devoted to exploit abstract interpretation for static program analysis on SDF program optimizations [14, 34, 83]. In [14], Blanchet et al. introduced an abstract interpretation-based static program analysis to theoretically verify large safety-critical software including periodic synchronous safety critical embedded software. The work exploits loop unrolling and trace partitioning for better analysis quality for program transformation. Halbwachs [34] utilizes an application of abstract interpretation, approximate reachability analysis to verify critical properties of synchronous programs.

Trace partitioning is proposed by [10, 56, 68] to improve the quality of static analysis by obtaining path sensitivity. In [9], Balakrishnan et al. proposes a refinement technique to simplify control-flow sequences especially for loops to improve the precision of abstract interpretation in the presence of widening.

7.3 SDF Scheduling Algorithms for Parallelization

A wide range of static scheduling algorithms for the SDF model exist [26, 31, 44, 47, 50, 67, 78, 79]. The StreamIt compiler [31] targets the Raw Microprocessor [57], shared-memory multicore architectures and clusters of workstations. Adjacent actors are often fused to increase the computation-to-communication ratio as long as the result is stateless. A heuristic for actor fission is then applied to increase data-parallelism to the extent that a communication-efficient balance between task and data parallelism is maintained. Coarse-grained software-pipelining of actors increases flexibility of the program partitioning and scheduling phases due to the eliminated actor dependencies within the same steady-state iteration. A greedy partitioning heuristic that minimizes the makespan is applied to load-balance actors among processor cores.

Actor placement of stream programs on multicore architectures with accelerators such as GPGPUs is examined in [79]. A communication-aware ILP formulation is presented for partitioning computations between CPU cores and GPGPU streaming multiprocessors (SMs) that minimizes makespan. Partitioned computations are then software-pipelined to execute on CPU cores and on the SMs of the GPU. Profiling is used to determine an optimal execution configuration of a stream program in terms of the number of registers per thread and the number of data-parallel actor instances. A buffer layout technique for GPUs that coalesces accesses to device memory is presented. Udupa et al. proposed a heuristic algorithm for the actor placement problem and experimentally demonstrated that the produced solutions are within 9.05% of the optimal solution obtained with the ILP program across a range of benchmark.

An iterative heuristic algorithm for partitioning and allocation that maps Kahn process networks with optional SDF-parts onto heterogeneous multiprocessors is presented in [18]. Partitions with short software pipelines are favored to reduce memory, latency and startup overheads that increase with the number of pipeline stages. In order to shorten software pipelines, generated partitions are convex, i.e., they are connected and do not contain circular dependencies. Partitioning is parameterizable through the provision of basic connected sets, which constitute collections of actors that the compiler is allowed to merge in a pairwise fashion. However, the assumption that partitions need to be convex is not shown to be code optimal.

Kudlur and Mahlke's stream graph modulo scheduling in [47] employs an ILP formulation to evenly distribute StreamIt actors among the synergistic processing elements of the Cell processor [36]. An integrated unfolding and partitioning technique that spreads data-parallel actors and maximally packs actors onto cores is incorporated into the ILP formulation. Stage assignment algorithm is then applied to overlap communication overheads with computations. In contrast to our model, an assumption was made about actor execution times that always dominate communication overheads. Hence, it is not applicable to multicore architectures where this assumption does not hold.

Another ILP formulation, which combines the requirements for both rate-optimal software pipelining and the minimization of inter-processor communication overhead on a communication exposed architecture is illustrated in [80]. In this multi-objective optimization problem, the primary objective is to maximize the computational rate, while the secondary objective is to minimize the communication overhead. A binary search is used to find the maximal rate where the communication is minimized. In contrast to our model, the approach is devised specifically for the communication exposed architecture.

A 2-approximation algorithm for mapping stream programs onto a multicore architecture has been proposed in [26]. A data rate transfer model has been presented for stream graphs which expresses the data rate of each actor depending on a single parameter which is referred to as a *closed form*. The approximation algorithm has been implemented based on the closed form expression. The quality of the solutions achieved is near-optimal. However, the approach did not consider the communication overhead in allocating actors to processor cores.

In [27], a unified ILP formulation is presented that does consider communication costs of data channels on cache-coherent multicore architectures. The objective is to minimize the maximum workload over the set of processor cores taking into account the incurred communication overhead. However, as the number of processor cores increases, the ILP program quickly becomes intractable and does not present a practical solution for the actor placement problem.

Chapter 8

Conclusion

In this thesis, we introduced LaminarIR compiler framework which comprises optimization techniques to tackle performance obstacles of stream programs which reside in data communication implementation. We designed a new intermediate representation for structured stream programs, LaminarIR, and its theoretical foundation to ensure semantically correct program transformation from structured stream programs into LaminarIR. The backbone of the compiler consists of a parser which accepts StreamIt and transforms into LaminarIR, and a backend which generates C code from LaminarIR. To expand applicability of LaminarIR, we introduced a static analysis based actor transformation method. A new communication-cost-aware orchestration algorithm for structured stream programs is devised to convey benefits of LaminarIR onto multicore architectures.

This chapter summarizes contributions of the thesis in Section 8.1, and suggests prominent future works in Section 8.2.

8.1 Summary

We designed LaminarIR and its compiler transformation that shift FIFO buffer management from run-time to compile-time. Underlying theory of the LaminarIR is established to guarantee semantic preserving transformation. Effectiveness of our approach is verified in microarchitectural level by evaluating the LaminarIR on four representative processor architectures from three different processor vendors and achieving between

6.64x and 7.43x performance improvement on average against FIFO queue implementation. We also proved the new data-flow representation enhances effectiveness of standard and contemporary compiler techniques, by showing accelerated SSA form promotions by the LaminarIR. Our approach eliminates 35.9% data-communication resulting in 60% less memory accesses on the Intel i7-2600K.

For further sound and effective LaminarIR transformation, we proposed a sound static program analysis technique which employs static actor specifications to analyze dynamic control-flow of the actors in stream programs during compile-time. Local direct token access is an essential prerequisite to further resolve program-wise direct token flow which improves performance of the program drastically once applied on SDF programs. The proposed static analysis technique is distinguished from previous approaches by guaranteed soundness of result, and the analysis utilizes SDF specific properties to improve analysis quality. Evaluation of our static analysis technique is conducted actor-wise, meaning that only local direct access transformation is applied without global direct access transformation of the LaminarIR compiler framework. This implies that clarified data-flow improves performance of programs even if the visibility of the data-flow is restricted to an actor not through a whole program. On average, a program composed with actors that uses the LaminarIR for internal data-flow representation achieved from 2.2x to 2.5x speedups on the three difference processor architectures from the Intel, AMD, and ARM each. But as few cases in the experimental result also shows that increased code size by the proposed technique can overwhelm benefits from the analysis. Extension of the analysis technique which takes hardware specifications such as instruction cache size and protocols into accounts to evaluate the quality of the analysis result.

By devising an approximation algorithm for communication-cost-aware orchestration, we stretched practical application scope of the LaminarIR to parallel architectures. The proposed approximation algorithm balances workload of stream programs on cache-coherent multicore architectures and produces solutions in polynomial time $\log_2 n$, where n is number of actors in a stream program and the worst-case run-time of the algorithm is $\mathcal{O}(np^3 \log_2 n)$. To the best of our knowledge, this is the first approximation algorithm for solving actor placement problem considering communication costs of data channels. Current state-of-the-art approaches address the actor placement problem considering communication costs from heuristic measurements or rely heavily on optimal exhaustive search techniques including integer linear programming. Heuristics are either fast but do not provide performance guarantees of their solutions. Integer linear program approaches provide optimal solutions, but feasible input sizes are very limited for commercial ILP solvers.

We evaluated our approximation algorithm on a range of benchmarks from various scientific domains and demonstrated its near-optimal experimental bound for 2, 4, and 6 processors. We further presented two instance bounds for the approximate solution: one is based on linear relaxation and the other utilizes properties of the approximation scheme. The linear relaxation demonstrates tight bounds for a low number of processors. As the number of processors increases, the instance bounds on the approximation ratio obtained using the linear program deteriorate quickly. On the other hand, the latter bounding technique achieves tight bounds providing an invaluable insight into the problem structure. For our benchmark, we observe instance bounds of 2 in the worst case.

8.2 Opportunities for Future Work

Expansion of target stream languages Stream programs are expressed by independent computing units that are connected by data channels, and intended to process continuous input data stream. Different stream programming languages retain their own distinguishable characteristics as well such as different language properties to convey own purposes of the language, different implementations of conceptual structures, and different optimization techniques [35]. Thus, a framework that supports various stream programming language design has the prevailing academic meaning, because such framework enables researchers to apply and evaluate their approaches on programs with different characteristics. Examining an approach from various angles will enhance quality and broaden applicability of the approaches. One way to implement the generic stream programming language framework would be to develop a profound parser that accepts programs in various stream programming languages and generates a unified AST.

Scala [25] is a multi-paradigm programming language that embraces functional programming and object-oriented programming. We suppose the functionality of Scala language design is fairly close to the design of stream programming language. Scala also provide a strong library for programming language parsing, called Scala parser combinator, which facilitate development of parsers in Scala. In addition, we assume the parser will enable researches on statement level optimizations of stream programs, such as code level prefetching of data by rearrangement of data access patterns in stream programs to minimize data access latency.

Synchronization cost aware orchestration From in-depth investigations on performance of stream programs, we recognized that synchronization overhead accompanied by each steady-state iteration is another significant performance bottleneck of stream programming languages.

One of standard ways to synchronize a steady-state iteration running on parallel processors is to use global synchronization methods such as barrier, causing constant synchronization overhead per steady-state iteration. Even worse, the synchronization overhead increases in proportion to the number of processors to utilize, which becomes a huge scalability drawback. More fine-grained synchronization methods such as mutex can be potential alternatives to global synchronization methods. But implementation of fine-grained synchronization methods requires not only a sound understanding of underlying hardware and operating system but also a scheduling algorithm which is aware of the synchronization overhead.

Deploying on many-core computing infrastructure Stream computing is becoming the most efficient way of obtaining useful knowledge from Big Data [63]. Big Data input streams arise in external environments such as sensors, web-browsers and the mobile Internet. Streams of events are pushed to servers to be processed in real-time. Because a large volume of data is arriving at such systems, the information cannot be processed anymore in real-time by a centralized solution. A new computing paradigm, called Big Data Stream Computing (BDSC), has emerged to facilitate such large-scale real-time analytics computations on Big Data streams [52, 73]. Few resesarch bodies have explored BDSC already. Google's Map-Reduce [23] is established as a programming model for batch processing on large data-sets. Batch-processing does not match an on-line streaming setting, because streams change frequently over time and the latest data is the most valuable one. There is no single well-established processing model yet for BDSC. Given the exponential growth of devices connected to the Internet, the demand for large-scale stream processing can be expected to grow significantly in the coming years.

Appendices

Appendix A

An Optimal, ILP-based Solution for the Min-MAX AP Problem

The following integer program corresponds to the min-max APP problem in Section 6.3. Let $G(V, E)$ be a stream graph where V is the set of actors and $E \subseteq V \times V$ is the set of channels. Let, further, P be the set of processors. The objective is to partition V into $i = |P|$ disjoint sets such that the maximum makespan across all i sets is minimized, i.e., min max$_i r_i$ (Equation 6.1):

$$\begin{aligned}
&\text{min. } \Pi \\
&\quad \text{s.t.} \sum_{(u,v)\in E} \frac{1}{2} c(u,v) \cdot x_{up} \cdot (1 - x_{vp}) + \sum_{u \in V} t(u) \cdot x_{up} \le \Pi && \forall p \in P \quad \text{(A.1a)} \\
&\quad \sum_{p \in P} x_{up} = 1 && \forall u \in V \quad \text{(A.1b)} \\
&\quad x_{up} \in \{0, 1\} && \forall u \in V, p \in P \quad \text{(A.1c)}
\end{aligned}$$

In the above model, we introduce 0-1 *decision* variables x_{up} that model a function $V \to P$ for mapping actors to processing elements, where:

$$x_{up} = \begin{cases} 1, & \text{if actor } u \text{ is assigned to processor } p \\ 0, & \text{otherwise} \end{cases}$$

The communication costs between actors $u, v \in V$ are represented by parameter

Table A.1: Comparison of the objective function values of the ILP (OPT) and the linear relaxation (LR) for two processors.

Benchmark	LR	OPT
MergeSort	2.277	2.529
ChanVocoder7	155.185	157.495
MatrixMult	81.64	87.289
FFT2	33.872	34.284
FMRadio	8.272	N/A
BeamFormer	13.499	13.723
DCT	8.294	8.633
TDE	3183.437	3183.749
RadixSort	7.952	7.952
BitonicSort	3.056	3.25
DES	137.461	N/A
Serpent	384.727	N/A
MPEG	68.793	70.172
SAR	43290.162	45438.718
FilterBankNew	22.519	22.889

$c(u, v)$. We consider both directions for communication; if there is a communication channel between u and v, then we incur a double communication cost since there is also a channel between v and u. Thus we take a half of the objective function value. Parameter $t(u),\ \forall u \in V$ denotes the execution time of actor u and Π is the makespan variable.

Constraint (A.1b) ensures that variables x_{up} represent a sound mapping, such that a single operation $u \in V$ is mapped to exactly one processor, while the integral constraint (A.1c) guarantees that there are no fractional mappings. Note that we relax the uniformity of processors; an actor can be scheduled on any available processor and it takes exactly the same time to execute independently of the chosen processor.

The linear relaxation of Equation A.1 that was used to derive a lower bound on the objective function value is given below:

$$\begin{aligned}
&\text{min. } \Pi \\
&\text{s.t.} \sum_{(u,v)\in E} \frac{1}{2} c(u,v) \cdot y_{uvp} + \sum_{u \in V} t(u) \cdot x_{up} \leq \Pi && \forall p \in P \\
&\sum_{p \in P} x_{up} = 1 && \forall u \in V
\end{aligned}$$

where we replace the quadratic term $x_{up} \cdot (1 - x_{vp})$ by y_{uvp} which corresponds to the absolute value $|x_{up} - x_{vp}|$:

$$y_{uvp} \geq x_{up} - x_{vp}, \forall p \in P$$
$$y_{uvp} \geq x_{vp} - x_{up}, \forall p \in P$$

Table A.1 provides a measure of the quality of relaxed linear program compared to the optimal solution obtained with the ILP for two processors. We only present the two-processor case for the comparison due to the ILP program becoming intractable with increased number of processors (see Section 6.7).

Bibliography

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

[2] Lightweight performance counter tools (LIKWID) website. `https://code.google.com/p/likwid/`. Accessed: 2014-09-30.

[3] StreamIt website. `http://groups.csail.mit.edu/cag/streamit/index.shtml`. Accessed: 2014-09-30.

[4] 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.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[6] 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.

[7] 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.

[8] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the*

ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[9] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Refining the control structure of loops using static analysis. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 49–58, New York, NY, USA, 2009. ACM.

[10] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, and A. Gupta. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *Proceedings of the 15th International Symposium on Static Analysis*, SAS '08, pages 238–254, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[12] 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.

[13] 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.

[14] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 196–207, New York, NY, USA, 2003. ACM.

[15] 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.

[16] 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.

[17] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[18] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09: Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 57–66. ACM, 2009.

[19] 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.

[20] A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Comput. Lang. Syst. Struct.*, 37(1):24–42, Apr. 2011.

[21] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2-3):103–179, July 1992.

[22] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[23] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[24] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376. Springer-Verlag, 1974.

[25] M. O. et. al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[26] 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.

[27] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Profile-guided deployment of stream programs on multicores. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 79–88, New York, NY, USA, 2012. ACM.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

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

[31] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 151–162, New York, NY, USA, 2006. ACM.

[32] T. Goubier, R. Sirdey, S. Louise, and V. David. *Algorithms and Architectures for Parallel Processing: 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I*, chapter ΣC: A Programming Model and Language for Embedded Manycores, pages 385–394. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[33] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354. IEEE Computer Society, 2005.

[34] N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming*, 31(1):75 – 89, 1998. Selected Papers of the First International Static Analysis Symposium.

[35] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, Mar. 2014.

[36] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA '05: Proceedings of the 2005 International Symposium on High-Performance Computer Architecture*, volume 0, pages 258–262. IEEE Computer Society, 2005.

[37] 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.

[38] IBM Corporation. *IBM ILOG AMPL Version 12.2 USer's Guide*. May 2010.

[39] IBM Corporation. *IBM ILOG CPLEX Optimization Studio CPLEX USer's Manual*. 2015.

[40] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*. January 2015.

[41] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. January 2016.

[42] W. M. Johnston, J. R. Pual, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.

[43] G. Kahn. The semantics of a simple language for parallel processing. *Proc IFIP Congress*, pages 471–475, 1974.

[44] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. *LCTES '03: Proceedings of the 2003 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 38(7):1235–1245, 2003.

[45] Y. Ko, B. Burgstaller, and B. Scholz. Laminarir: Compile-time queues for structured streams. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 121–130, New York, NY, USA, 2015. ACM.

[46] 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.

[47] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008.

[48] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 2015.

[49] 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.

[50] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36:24–35, 1987.

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

[52] K.-C. Li, H. Jiang, L. T. Yang, and A. Cuzzocrea. *Big Data: Algorithms, Analytics, and Applications*. Chapman & Hall/CRC, 1st edition, 2015.

[53] B. M. Maggs, G. L. Miller, O. Parekh, R. Ravi, and S. L. M. Woo. Solving symmetric diagonally-dominant systems by preconditioning. Technical report, IN PROCEEDINGS. 38TH ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 2002.

[54] A. Malik and D. Gregg. Heuristics on reachability trees for bicriteria scheduling of stream graphs on heterogeneous multiprocessor architectures. *ACM Trans. Embed. Comput. Syst.*, 14(2):23:1–23:26, Feb. 2015.

[55] 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.

[56] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, pages 5–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[57] E. W. Michael, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: The raw machine. *IEEE Computer*, 30:86–93, 1997.

[58] 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.

[59] P. K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. PhD thesis, EECS Department, University of California, Berkeley, 1996.

[60] P. K. Murthy and S. S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Trans. Des. Autom. Electron. Syst.*, 9(2):212–237, Apr. 2004.

[61] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Form. Methods Syst. Des.*, 11(1):41–70, July 1997.

[62] D. Nguyen and J. Lee. Communication-aware mapping of stream graphs for multi-GPU platforms. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 94–104. ACM, 2016.

[63] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14, New York, NY, USA, 2013. ACM.

[64] R. Reiter. Scheduling parallel computations. *J. ACM*, 15(4):590–599, 1968.

[65] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pages 285–296, 1993.

[66] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Acoustics, Speech, and*

Signal Processing, 1995. ICASSP-95., 1995 International Conference on, volume 4, pages 2651–2654, 1995.

[67] S. Robert. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[68] S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. *Static Analysis: 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings*, chapter Static Analysis in Disjunctive Numerical Domains, pages 3–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[69] 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.

[70] 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.

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

[72] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12, 2010.

[73] D. Sun, G. Zhang, W. Zheng, and K. Li. Key technologies for Big Data stream computing. In K.-C. Li, H. Jiang, L. T. Yang, and A. Cuzzocrea, editors, *Big Data: Algorithms, Analytics, and Applications*. Chapman and Hall/CRC, 2015.

[74] W. Sung, J. Kim, and S. Ha. Memory efficient software synthesis from dataflow graph. In *Proceedings of the 11th International Symposium on System Synthesis*, pages 137–144. IEEE, 1998.

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

[76] 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.

[77] 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.

[78] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *CGO '09: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2009.

[79] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Synergistic execution of stream programs on multicores with accelerators. *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 44(7), 2009.

[80] H. Wei, J. Yu, H. Yu, and G. R. Gao. Minimizing communication in rate-optimal software pipelining for stream programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 210–217, New York, NY, USA, 2010. ACM.

[81] R. Wilhelm and D. Maurer. *Compiler design*. Addison-Wesley, 1995.

[82] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, Apr. 2011.

[83] M. Wipliez and M. Raulet. Classification of dataflow actors with satisfiability and abstract interpretation. *Int. J. Embed. Real-Time Commun. Syst.*, 3(1):49–69, Jan. 2012.

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

[85] D. Zhang, Z.-Z. Li, H. Song, and L. Liu. A programming model for an embedded media processing architecture. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 3553 of *Lecture Notes in Computer Science*, pages 251–261. Springer Berlin Heidelberg, 2005.