Scheduling and Optimizing Stream Programs on Multicore Machines by Exploiting High-Level Abstractions

Dai Bui

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-184
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-184.html

November 7, 2013

Scheduling and Optimizing Stream Programs on Multicore Machines by Exploiting High-Level Abstractions

by

Dai Nguyen Bui

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering & Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Professor George Necula
Professor Raja Sengupta

Fall 2013

Scheduling and Optimizing Stream Programs on Multicore Machines by Exploiting High-Level Abstractions

Abstract

Scheduling and Optimizing Stream Programs on Multicore Machines by Exploiting High-Level Abstractions

by

Dai Nguyen Bui

Doctor of Philosophy in Engineering - Electrical Engineering & Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

Real-time streaming of HD movies and TV via YouTube, Netflix, Apple TV and Xbox Live is gaining popularity. Stream programs often consume considerable amounts of energy due to their compute-intensive nature. Making stream programs energy-efficient is important, especially for energy-constrained computing devices such as mobile phones and tablets. The first part of this thesis focuses on exploiting the popular Synchronous Dataflow (SDF) high-level abstraction of stream programs to design adaptive stream programs for energy reduction on multicore machines. Observing that IO rates of stream programs can vary at runtime, we seek to make stream programs adaptive by transforming their internal structures to adapt required occupied computing resources, e.g., cores and memory, to workload changes at runtime. Our experiments show that adapting stream programs to IO rate changes can lead to significant energy reduction. In addition, we also show that the modularity and static attributes of stream programs' abstraction not only help map stream programs on multicore machines more easily but also enable energy-efficient routing schemes of high-bandwidth stream traffic on the interconnection fabric, such as networks on-chip.

While SDF abstractions can help optimize stream programs on multicore machines, SDF is more suitable for describing stream data-intensive computations such as FFT, DCT, and FIR and so on. Modern stream operations such as MPEG2 or MP3 encoders/decoders are often more sophisticated and composed of multiple such computations. Enabling operation synchronization between different such computations with different semantics leads to the need for control messaging. We extend previous work on control messaging and give a formal definition for control message latency via the semantics of information wavefronts. This control-operation-integrated SDF (COSDF) is able to model sophisticated stream programs more precisely. However, the conventional scheduling method developed for SDF is not sufficient to schedule COSDF applications. To schedule COSDF applications, we develop a scheduling method using dependency graphs and applying a periodic graph theory, based on reduced dependency graphs (RDG). This RDG scheduling method also helps extract

parallelism of stream programs. The more precise abstraction of COSDF is expected to help synthesize and generate sophisticated stream programs more efficiently.

Although the SDF modularity property also improves programmability, it can come at a price of efficiency when SDF models are not compiled and run using model-based design environments. However, compiling large SDF models to mitigate the inefficiency can be prohibitive in the situations where even a small change in a model may lead to large recompilation overhead. We tackle the problem by proposing a method for incrementally compiling large SDF models that faithfully captures the executions of original SDF models to avoid potential artificial deadlocks of a naive compilation method.

To my parents, Bui Minh Quang and Nguyen Thi Dai Tu.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my parents, my brother and sister in-law for their love and constant support. They have been the source of motivation for me to get through the ups and downs of graduate student life. My dad, a high school teacher, has set an inspiring example for me. Though I am not "officially" his student for a day, I have learned a great deal from his hard work and his attitude toward life.

I would like to thank my advisor, Professor Edward A. Lee, for tolerating my ignorance during all these years. Edward has been a great advisor! He gave me lots of advice, time and space to develop myself during my PhD. His enthusiasm for beautiful knowledge and his passion for writing quality code are really inspiring.

A part of this thesis is from the work with Stavros Tripakis, Bert Rodiers and others. Thanks Stavros for his guidance during my first steps in research. Thanks Bill Thies for his work on Teleport messaging and his thoughtful feedback. A part of this thesis focuses on extending his Teleport messaging work. I also would like to thank Hiren Patel, Jan Reineke and Isaac Liu for their collaboration on a couple of papers.

Thanks Ben Lickly, Sayak Ray, Dan Holcomb, Chris Shaver, Christos Stergiou, and Marten Lohstroh for spending time discussing a variety of topics with me. Ben and I have taken a few classes together. It was quite fun, one time, we both only realized there had been a homework assignment after receiving an email announcing that homework deadline was extended! Thanks Mary Stewart and Christopher Brooks for their wonderful administration. Thanks Jia Zou, Shanna Forbes, David Broman, Patricia Derler, Eleftherios Matsikoudis, Michael Zimmer and Ben Zhang; interacting with you is my great pleasure. I also would like to thank other people in the Ptolemy group. I have had a great time being a member of the group.

I would like to thank Professors George Necula, Raja Sengupta, and Sanjit Seshia for being on my thesis committee and for their feedback and suggestions.

I would like to thank Vietnam Education Foundation (VEF) for financially supporting a part of my graduate study.

Last but not least, thanks my Vietnamese friends in Berkeley for socializing (including karaoke and soccer but not limited to) with me.

Chapter 1

Introduction

Real-time streaming of media data is growing in popularity. This includes both capture and processing of real-time video and audio, and delivery of video and audio from servers; recent usage number shows over 800 million unique users that visit YouTube to watch over 3 billion hours of video each month(`http://www.youtube.com/t/press_statistics/`). Stream programs often exhibit rich parallelism, as a result, they are very suitable for multicore machines. Most of the previous work focuses on parallelizing stream applications on multicore machines to speed up computation. However, as people use mobile devices more and data-centers consume huge amounts of electricity to process big amounts of stream data, energy-efficient stream processing becomes crucial. The first part of this thesis focuses on improving energy consumption of stream applications on multicore machines by exploiting the modularity and the static properties of Synchronous Dataflow (SDF), a popular stream model of computation. While conventional SDF is expressive enough to describe most of the stream programs [113], modern stream programs become more complicated and dynamic. Consequently, SDF is not convenient enough to express modern stream programs and the theory of SDF is not sufficient to schedule and check for deadlocks in the stream programs. Another issue arises when large SDF models require compiling incrementally to reduce compile time [116, 75, 77]. However, the naive monolithic code generation for sub-models can lead to deadlocks and rate inconsistency when linked with other sub-models later. In the second part of this thesis, we focus on those compilation issues of stream programs.

1.1 Parallelism-Energy Relationship

In this thesis, we focus on optimizing energy for stream programs by exploiting their rich parallelism and concurrency. The following equations from [59] capture the basic relation between energy consumption E, voltage V_{dd} and operating frequency f:

$$E = \beta \frac{1}{2} C f V_{dd}^2 U \tag{1.1}$$

$$f = \gamma \frac{(V_{dd} - V_{th})^{\alpha}}{V_{dd}} \tag{1.2}$$

where β is the activity constant, C is the total capacity. f, V_{dd} are the operating frequency and voltage respectively. α is a technology-dependent factor around 1.2 to 1.6. V_{th} is the threshold voltage. U is the utilization of the circuit component.

From (1.1) and (1.2), we can see that dynamic energy consumption is approximately proportional to the cube of frequency. Lowering frequency often leads to lower energy consumption at the cost of increasing running time. Especially, when CPU speed rises too high to keep up with the processing demand of applications, CPUs can become very hot and the heat can damage CPUs. As a result, to reduce energy consumption while maintaining the same running time, we often exploit parallel processing to speed up applications.

Stream programs are suitable for parallel processing because of their rich concurrency. Stream programs can be decomposed into multiple autonomous modules, called *actors*, connected using FIFO channels. As a result, we can map concurrent actors onto different cores to reduce running time. However, decomposing stream programs into concurrent actors in such a way that they interact with each other deterministically is non-trivial. In the next section, we summarize a number of programming models for stream programs that help determine how concurrent actors interact.

1.2 Stream Programming Models

Kahn Process Networks (KPN) [58] is a general programming model for stream programs. KPN applications are composed of several processes, called actors in this thesis, connected using FIFO channels. KPN actors block on read when there are not enough data on their input channels. KPN actors can always write data to their output channels (non-blocking write). These conditions lead to the deterministic execution of KPN programs [58]. The determinism of KPN programs can be understood as, regardless of possible execution orders of actors, the final outputs and the final data history on channels of one program remain unchanged.

Although KPN applications are deterministic [58, 79, 109], optimizing KPN applications is often more difficult [93, 39, 71] than less general programming models such as Synchronous Dataflow (SDF) [70]. Although SDF is less general than KPN, it turns out that SDF is still general enough to model many stream applications [113], and it is suitable for compiler optimization techniques for optimizing buffer sizes and static scheduling [14, 105, 13, 15, 16, 87, 86, 100, 47, 68, 37, 98, 50, 51, 116, 24]. SDF is mainly different from KPN in the sense that each SDF actor produces/consumes a fixed number of tokens from each of its output/input channel respectively. Cyclo Static Dataflow (CSDF) is a programming model generalized from SDF to enable more flexible executions. Each CSDF actor executes cyclically a fixed set of phases. At each phase, each CSDF actor produces/consumes a fixed number of tokens on its output/input channels. This generalization makes CSDF more flexible than SDF and enables CSDF to model the executions of certain stream programs that SDF cannot [94]

while maintaining the advantage of analysable buffers and static scheduling. However, this generalization is not necessary in practice because SDF is still flexible enough to model almost all normal stream programs [113, 112]. While SDF is suitable for modeling one-dimensional streams such as voice signals, it is not natural for modeling multi-dimensional streams such as videos frames. Modeling multi-dimensional streams using SDF requires converting those streams to one-dimensional ones. This conversion may render scheduling less efficient. Multi-dimensional SDF (MDSDF) [88] is a generalization of SDF for modeling multi-dimensional streams.

Petri nets [85], often used to model distributed systems, are more general than SDF. While Petri nets can model a wider range of applications than SDF, inferring properties, e.g., SDF properties, from Petri nets may be expensive and difficult.

1.3 Languages for Writing Stream Programs

StreamIt [114] is a programming language for writing stream programs. The underlying programming model of StreamIt is based on SDF with a number of extensions such as sliding windows [114], e.g., actors can peek (read) tokens on their input channels without consuming them, structured stream graphs [114] to ease stream graph manipulation processes such as clustering, partitioning and so on, and Teleport messaging [115] for expressing control operations. The static properties of SDF are useful not only in optimizing buffer space and statically scheduling actor executions but also in increasing parallelism, e.g., by duplicating stateless actors [47].

Brook [23] is a language suitable for writing high-performance image manipulation applications and compiling them to graphic processors. Spiral [123] is a high-level programming language for digital signal processing (DSP) applications. Programmers can write DSP mathematical operations in Spiral and the compiler will automatically transform the operations, generate code and tune the code to specific processor models.

Ptolemy (`http://ptolemy.eecs.berkeley.edu`), LabVIEW (`http://www.ni.com/labview/`) and Simulink (`http://www.mathworks.com/products/simulink/`) are model-based programming environments used for quick prototyping of signal processing applications.

1.4 Power of Stream Program Abstractions

In this thesis, we show the utilization of the stream program abstractions to design energy efficient programs as well as to improve verifying, scheduling and compilation processes.

Adaptive Stream Programs for Energy Efficiency

Previous work on compilation of stream programs for multicore machines [47, 68] often tries to maximize speed for specific processor models assuming that input data are always available and output devices can always take in processed data produced by stream programs.

However, when stream programs are embedded into external environment, this execution model can result in energy inefficiency. For example, when a stream program runs on four cores and its input rates are low, e.g., due to low sampling rates, the stream program will have to pause frequently to wait for input data. Pausing the program can waste energy due to processor energy leakage and synchronization between threads, which may use spin-locks to avoid performance degradation. To mitigate the problem, we can instead run the program on two cores when its input rates become low. This running scheme reduces leakage power through turning off two unused cores and lowers the inter-core synchronization energy through reducing the numbers of synchronization cores (two cores instead of four).

As a result, being able to *adapt*, e.g., adjusting the number of cores used, stream programs to IO rates at runtime would save energy. However, transforming stream programs on-the-fly to adapt to IO rate changes while maintaining consistent program states is not straightforward. The first part of this thesis will focus on designing a mechanism that exploits the high-level abstractions of stream programs, e.g., SDF, to transform stream programs on-the-fly to reduce energy [24]. The main idea is that programmers only need to write a single specification for one stream program; the compiler will exploit the high-level abstractions of stream programs to generate runtime transformable program automatically.

Energy-Efficient Routing of Stream Traffic on Networks on-Chip

Stream applications often demand high communication bandwidth. This property makes networks on-chip (NoC) a suitable communication paradigm for stream applications thanks to their path diversity [81]. As energy consumption is approximately proportional to the cube of frequency, exploiting NoC's path diversity property not only mitigates the scalability issue of the bus system but also can help reduce energy consumption. For example, instead of routing traffic through congested links, which would require the congested links running at high frequencies, we can route traffic through less congested links, which would enable links and routers to run at lower frequencies. As a result, we can reduce energy consumption of NoC carrying stream traffic.

Stream programs are often modeled by SDF. Because the amount of traffic between actors often remains stationary between iterations, when mapping actors of stream programs to multicore machines, the inter-core communication often remains stationary. Exploiting this stationary inter-core communication property of stream traffic, we can formulate a mixed integer linear programming (MILP) optimization problem that finds optimal frequencies for links and routers to minimize energy consumption. To make the problem more scalable, we also present a heuristic routing algorithm that approximates the optimal results of the MILP method. Our experiments show that, by exploiting NoC's path diversity to minimize links and routers' frequencies, we can significantly reduce energy consumption of NoC.

On the Semantics of Control-Operation-Integrated Synchronous Dataflow: Schedulability and Parallelism

SDF is suitable for modeling stream data-intensive computations, e.g., FIR, FFT, and DCT and so on. However, sophisticated stream operations, such as MPEG2 or MP3 encoders/decoders, are often composed of multiple such data-intensive computations. To synchronize the executions and configurations between such data-intensive computations with different semantics, they exchange control messages (CM) containing control information, e.g., frame types, filter parameters and so on. We first extend Thies et al.'s work [115] on control messaging and give a formal semantics for control operations (CO). Because integrating COs into SDF imposes additional control scheduling constraints, some SDF programs can become non-schedulable after integrating COs. The conventional SDF scheduling mechanism [70] does not account for such control constraints. To check whether such CO-integrated SDF (COSDF) programs are schedulable and find a schedule if one exists, we present a systematic scheduling method based on the reduced dependency graph (RDG) [61] theory. Our scheduling method can not only find possible schedules for COSDF programs but also extract parallelism from such programs.

To address the scalability issue of our initial scheduling method, we present a graph pruning technique that significantly reduces the size of RDGs of practical stream applications. We then present an algorithm that can make schedulability checking time for COSDF programs reasonable (within one minute for all the available benchmarks).

Composability of Modular SDF

In the last part of this thesis, we tackle the problem of incremental compilation of large SDF models. When large SDF models are executed within model-based environments such as Ptolemy (`http://ptolemy.eecs.berkeley.edu`) and LabVIEW (`http://www.ni.com/labview/`), to reduce large running time, it would be beneficial to compile the large models to mitigate the overhead of the execution environments. However, this approach is often undermined by large recompilation time even when just a small part of large models is edited. To reduce the large recompilation time, we can incrementally compile sub-parts of a large SDF model [75, 77, 116]. When a sub-part is edited, we only need to recompile the part and relink it with the other compiled parts. However, this incremental compilation technique can produce non-functional, e.g., deadlocked, buffer-unbounded compiled models; meanwhile the corresponding original models are functional. To avoid this problem, we present a technique for incrementally compiling sub-parts of large SDF models and composing them together that faithfully captures the executions of original models.

Chapter 2

Background

2.1 Synchronous Dataflow

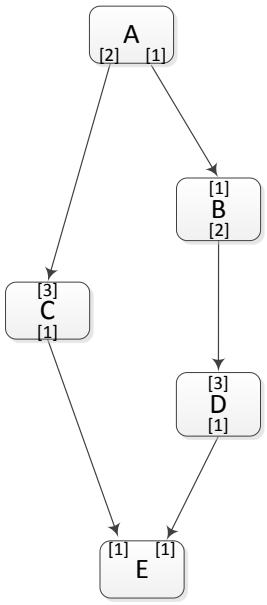


Figure 2.1: A Synchronous Dataflow model.

Lee and Messerschmitt propose SDF [70], which can model the executions of many stream programs [113]. This abstraction enables several compiler optimization techniques [14, 105, 13, 15, 16, 87, 86, 100, 47, 68, 98, 50, 51, 37, 116, 24] for buffer space, scheduling and mapping to underlying architecture as well as worst-case performance analysis [40].

In the SDF model of computation, a stream program is given as a graph composed of a set of actors communicating through FIFO channels. Each actor has a set of input and output ports. Each channel connects an output port of an actor to an input port of another actor.

Each actor consumes/produces a fixed number of tokens from/to each of its input/output port each time it executes. This model of computation is interesting as it allows static scheduling with bounded buffer space whenever possible.

Figure 2.1 shows an example of an SDF stream graph. Actor `A` has two output ports. Each time `A` executes, it produces 2 tokens on its left port and 1 token on its right port, actor `B` consumes 1 and produces 2 tokens each time it executes, and so on. The theory of the SDF programming model provides algorithms to compute the number of times each actor has to execute within one *iteration*, of the *entire* stream graph, so that the total number of tokens produced on each channel between two actors is equal to the total number of tokens consumed. In other words, the number of tokens on each channel between two actors remains unchanged after one iteration of the entire stream graph. For example, in one iteration of the stream graph in Figure 2.1, actors `A`, `B`, `C`, `D`, `E` have to execute 3, 3, 2, 2, 2 times respectively. Repeating this basic schedule makes the number of tokens on each channel remains the same after one iteration of the entire stream graph. For instance, for the channel between `B` and `D`, in one iteration, `B` produces 3×2 tokens while `D` consumes 2×3 tokens.

2.2 Basic SDF Scheduling Techniques

In this section, we present the scheduling technique for SDF stream graphs by Lee and Messerschmitt [70]. The SDF analysis methods proposed in [70] allow to check whether a given SDF graph has a *periodic admissible sequential schedule* (PASS). Existence of a PASS guarantees two things: first, that the actors in the graph can execute infinitely often without *deadlock*; and second, that only *bounded queues* are required to store intermediate tokens produced during the executions of these actors. The scheduling technique is composed of two steps: 1) Finding the number of times each actor executes within one iteration so that the number of tokens on each channel remain the same by solving a balance equation; 2) Using symbolic execution to find a concrete schedule, if one exists, for actors within one iteration based on the number of times each actor executes computed in the previous step.

Balance Equation for SDF Graphs

Let `A` and `B` be the upstream and downstream actors respectively of a channel. Each time `A` executes, it produces p tokens to the channel and `B` consumes c tokens from the channel. Let us assume within one iteration of the stream graph, `A` executes a times and `B` executes b times. Because after one iteration of an SDF graph, the number of tokens on each channel remains the same, for each channel, the number of tokens produced by the upstream actor must be equal to as the number of tokens consumed by the downstream actor. We have the following balance equation for the channel between `A` and `B`:

$$pa - cb = 0 \tag{2.1}$$

Now let Γ be the production/consumption matrix of an SDF graph where the entry at $(i,\ j)$ is the number of tokens that the j^{th} actor produces to the i^{th} channel each time it executes. Note that if the j^{th} actor consumes tokens from the i^{th} channel, the value of the entry at $(j,\ i)$ is equal to *negative* of the number of tokens that the j^{th} actor consumes each execution. Now let q be the vector denoting the numbers of times the actors execute within one iteration. By applying equation (2.1) for all the channels in the SDF graph, we reach:

$$\Gamma q = \vec{0} \tag{2.2}$$

As Γ is derived from the graph, solving (2.2) for q results in the number of times each actor has to execute within one iteration. Because we are only interested in non-trivial solutions, e.g., $q \neq \vec{0}$, as a result, if matrix Γ has rank $\alpha - 1$, then equation (2.2) has a non-trivial solution. A *connected* stream graph with matrix Γ of rank $\alpha - 1$ is called a *consistent* graph.

The following theorem from [70] proves that, for a consistent graph, there exists $q > \vec{0}$.

Theorem 1 *(Lee & Messerschmitt) Given a consistent connected SDF model with production/consumption matrix Γ, we can find an integer vector q whose every element is greater than zero such that $\Gamma q = \vec{0}$. Furthermore, there exists a unique least such vector q.*

Illustrative Example

We will use the example in Figure 2.1 to illustrate the method. The production/consumption matrix Γ for the stream graph is as follows:

$$\Gamma = \begin{array}{c} \\ (\texttt{A} \rightarrow \texttt{B}) \\ (\texttt{A} \rightarrow \texttt{C}) \\ (\texttt{B} \rightarrow \texttt{D}) \\ (\texttt{C} \rightarrow \texttt{E}) \\ (\texttt{D} \rightarrow \texttt{E}) \end{array} \begin{array}{c} \begin{array}{ccccc} \texttt{A} & \texttt{B} & \texttt{C} & \texttt{D} & \texttt{E} \end{array} \\ \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 2 & 0 & -3 & 0 & 0 \\ 0 & 2 & 0 & -3 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \end{pmatrix} \end{array} \tag{2.3}$$

Solving equation (2.2) for the smallest q, we obtain:

$$q = \begin{pmatrix} 3 \\ 3 \\ 2 \\ 2 \\ 2 \end{pmatrix} \tag{2.4}$$

We have $q_{\texttt{A}} = 3, q_{\texttt{B}} = 3, q_{\texttt{C}} = 2, q_{\texttt{D}} = 2, q_{\texttt{E}} = 2$ indicating that within each iteration, if `A, B, C, D` and `E` execute 3, 3, 2, 2, 2 times respectively, the number of tokens on each channel will remain the same.

To vectorize SDF programs or to coarsen executions, we can scale the smallest q. Vectorizing SDF programs can help improve speed and reduce energy consumption by running SDF programs on vector processors [50]. Scaling q is not always possible for stream graphs with loops.

Symbolic Execution to Derive Concrete Sequential Schedules

Solving the balance equation as in the previous section gives the number of times each actor has to execute within one iteration so that the number of tokens on each channel remains the same. However, a concrete execution schedule is not immediately available after the procedure.

Definition 1 *An actor in a stream graph is runnable if there are enough tokens on all of its input channels for it to consume to execute at least once.*

From the vector of actor executions within one iteration, we then apply a symbolic execution procedure [70] to find concrete sequential schedules as in Algorithm 2.2.

Related Work

The static properties of SDF present several optimization opportunities [13]. In [14], Bhattacharyya and Lee present a *looped* scheduling technique, called pairwise grouping of adjacent nodes (PGAN), that hierarchically clusters SDF graphs to expose the successive reoccurrence of identical firing sub-patterns. For example, the technique may transform the schedule (3A)(3B)(2C)(2D)(2E) of the SDF graph in Figure 2.1 into a looped form as 3(AB)2(CDE). This is done by successively grouping two adjacent nodes in an SDF graph; e.g., A and B are grouped together. This PGAN technique can result in significant buffer and code space reduction. In [15], Bhattacharyya et al. present two heuristics that significantly reduce the complexity of PGAN for *acyclic* SDF graphs. Sermulins et al. [100] use a similar technique that fuses (groups) adjacent actors in such a way that the code and data space, required by fusing actors, can fit into caches of processors. Gordon et al. [47] evaluate a technique that increases the parallelism of SDF programs by replicating stateless actors. This is possible for SDF actors have static IO rates. Murthy et al. [87] present several techniques that minimize the memory requirement of chain-structured SDF graphs. Murthy and Bhattacharyya [86] present a scheduling technique that allows the channels in an SDF graph to share buffers. As a result, this buffer merging technique can help reduce memory requirements for SDF programs. Samadi et al. [98] show that programs that can be described in the form of SDF can be compiled to general-purpose graphics processing units (GPGPU) adaptively based on input sizes resulting in significant speed improvement even when compared to the hand-optimized corresponding programs. This is made possible because describing programs in the form of SDF can help compilers schedule and analyse the programs better. Hormati [50] et al. exploit the repetitive and static properties of SDF programs to compile the programs

```
1  findSchedule(StreamGraph, q)
2  L ← StreamGraph.actors()     ▷ Form an arbitrary ordered list of all the actors in a
   stream graph.
3  schedule ← new List()
4  while True do
5  |  areAllScheduled ← True
6  |  progress ← False
7  |  forall the actor ∈ L do
8  |  |  if isRunnable(actor) and q_actor > 0 then
9  |  |  |  schedule.add(actor)
10 |  |  |  updateChannelInfo()▷ Update the numbers of tokens on the input/output
   |  |  |  channels of actor changed by firing actor.
11 |  |  |  q_actor ← q_actor − 1
12 |  |  |  progress ← True
13 |  |  end
14 |  |  if q_actor > 0 then
15 |  |  |  areAllScheduled ←False
16 |  |  end
17 |  end
18 |  if not progress then
19 |  |  return ∅                                          ▷ No schedule exists.
20 |  end
21 |  if areAllScheduled then
22 |  |  return schedule                                   ▷ Found the schedule.
23 |  end
24 end
```

Figure 2.2: Symbolic execution for finding concrete sequential schedules.

to vector processors, resulting in significant speed improvement and energy reduction. Bui and Lee [24] show how to exploit the SDF abstraction to design stream programs that can adapt to IO rate changes for energy and resource reduction. Geilen and Stuijk present a method for estimating the worst-case performance of SDF programs when switching between different scenarios. Falk et al. [37] show how to cluster SDF actors together to form quasi-static schedules that co-ordinate actor firings when mapped to multiprocessor system-on-chip (MPSOC) resulting in significant latency and throughput improvement.

2.3 Stream Dependency Function and Dependency Graph

The method for finding concrete execution schedules in Section 2.2 only finds single processor schedules. However, SDF programs often expose rich concurrency suitable for running on multicore. In this thesis, we will present and develop more advanced scheduling techniques that are able to extract parallelism of SDF graphs as well as to schedule SDF graphs when integrated with control-operations used to synchronize executions between actors. The scheduling techniques are based on building dependency graphs that capture execution dependencies between actors. Lee and Messerschmitt [70] also present a parallel scheduler but it is for SDF without control-operations.

First, we will describe an abstract function called Stream Dependency Function, `SDEP`, presented by Thies et al. [115]. This function can be used to construct data dependency graphs.

Stream Dependency Function `SDEP`

`SDEP` represents the data dependency of the executions of one actor on the executions of another actor in a stream graph. $\texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n)$ returns the minimum number of times actor `A` must execute to allow actor `B` to execute n times. This is based on the intuition that an execution of a downstream actor requires data from some executions of its upstream actors; thus, each execution of a downstream actor depends on certain executions of its upstream actors. In other words, given an n^{th} execution of a downstream actor `B`, the `SDEP` function returns the latest execution of an upstream actor `A` that the data it produces, going through and being processed by intermediate actors, affects the input data consumed by the n^{th} execution of actor `B`.

Definition 2 *(SDEP)*

$$\texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n) = \min_{\phi\in\Phi,|\phi\wedge\texttt{B}|=n} |\phi \wedge \texttt{A}|$$

where Φ is the set of all legal sequences of executions, ϕ is a legal sequence of executions and $|\phi \wedge \texttt{B}|$ is the number of times actor `B` executes in the sequence ϕ.

SDEP Calculation

The StreamIt compiler computes the `SDEP` function using the pull schedule described in [115]. To intuitively illustrate the `SDEP` calculation using the pull schedule, we take a simple example of actors `B` and `D` in Figure 2.1. The $\texttt{SDEP}_{\texttt{B}\leftarrow\texttt{D}}(m)$ is as in Table 2.1. In the example, when `D` does not execute, it does not require any number of executions of `B`. In order for `D` to execute the first time, it requires three tokens on its input channel. Based on this requirement, the

pull schedule algorithm will try to pull three tokens from the supplier, actor B. To supply the three tokens, B has to execute at least two times, therefore we have $\texttt{SDEP}_{\texttt{B}\leftarrow\texttt{D}}(1) = 2$. Similarly, when D wants to execute one more time, it needs two more tokens so it will try to pull the two tokens from B. Again, B has to execute one more time to supply the two tokens and we have $\texttt{SDEP}_{\texttt{B}\leftarrow\texttt{D}}(2) = 3$. Readers can refer to [115] for further details on the algorithm.

Table 2.1: Stream dependency function example

m	$\texttt{SDEP}_{\texttt{B}\leftarrow\texttt{D}}(m)$
0	0
1	2
2	3

Periodicity of SDEP

As SDF is periodic, therefore SDEP is also periodic. This means that one does not need to compute SDEP for all executions, instead, one can compute SDEP for some executions then based on the periodic property of SDEP to query future dependency information. The following equation was adapted from [115]:

$$\texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n) = i * |\mathcal{S} \wedge \texttt{A}| + \texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n - i * |\mathcal{S} \wedge \texttt{B}|) \tag{2.5}$$

where $\mathcal{S}$ is the executions of actors within one iteration of a stream graph. Accordingly, $|\mathcal{S} \wedge \texttt{A}|$ is the number of executions of actor A within one iteration of its stream graph. i is some iteration such that $0 \leq i \leq p(n)$ where $p(n) = n \div |\mathcal{S} \wedge \texttt{B}|$ is the number of iterations that B has completed by its n^{th} execution.[1]

2.4 Sliding Windows

The StreamIt language [114] is a language for writing stream applications extending the SDF model of computation with a sliding window feature[2], in which actors can *peek* (read) tokens ahead without consuming those tokens. As a result, many states of programs are stored on channels in the form of data tokens instead of being stored internally within actors. Consequently, many actors become *stateless* and eligible for the actor replication technique to speed up computation [47]. Figure 2.3 shows how peeking can help eliminate stateless actors. Suppose that B consumes one token each execution. B also stores the two most recently consumed tokens in an internal queue, say t_1, t_2. As a consequence, B is stateful. Now suppose that B can peek, then tokens t_1 and t_2 can be stored at the input channel instead. As a result, B becomes stateless.

[1] We define $a \div b \equiv \lfloor \frac{a}{b} \rfloor$.

[2] This feature was presented in Gabriel [17].

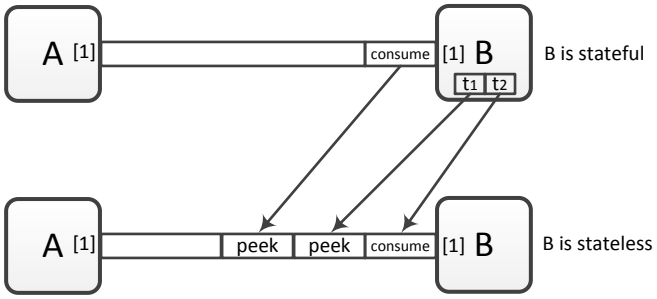


Figure 2.3: Simple peeking example.

Chapter 3

StreaMorph: Adaptive Stream Programs

This chapter presents the concept of *adaptive programs*, whose computation and communication structures can *morph* to adapt to environmental and demand changes to save energy and computing resources. In this approach, programmers write one single program using a language at a higher level of abstraction. The compiler will exploit the properties of the abstractions to generate an adaptive program that is able to adjust computation and communication structures to environmental and demand changes.

We develop a technique, called StreaMorph, that exploits the properties of stream programs' Synchronous Dataflow (SDF) programming model to enable runtime stream graph transformation. The StreaMorph technique can be used to optimize memory usage and to adjust core utilization leading to energy reduction by turning off idle cores or reducing operating frequencies. The main challenge for such a runtime transformation is to maintain consistent program states by copying states between different stream graph structures, because a stream program optimized for different numbers of cores often has different sets of actors and inter-actor channels. We propose an analysis that helps simplify program state copying processes by minimizing copying of states based on the properties of the SDF model.

Finally, we implement the StreaMorph method in the StreamIt compiler. Our experiments on the Intel Xeon E5450 show that using StreaMorph to minimize the number of cores used from eight cores to one core, e.g., when streaming rates become lower, can reduce energy consumption by 76.33% on average. Using StreaMorph to spread workload from four cores to six or seven cores, e.g., when more cores become available, to reduce operating frequencies, can lead to 10% energy reduction. In addition, StreaMorph can lead to a buffer size reduction of 82.58% in comparison with a straightforward inter-core actor migration technique when switching from using eight cores to one core.

3.1 Introduction

Given that traffic from mobile devices to YouTube(`http://www.youtube.com/t/press_statistics/`) tripled in 2011, energy-efficient stream computing is increasingly important, especially since battery life is a big concern for mobile devices. In addition, as computing devices are often now equipped with high-resolution displays and high-quality speakers, streaming quality is expected to rise accordingly. Processing high-quality streams requires proportionally higher computing energy. While several efforts [47, 68] aim at using parallel computing to meet the demand of stream computation, not much attention has been paid to energy-efficient parallel stream computing. These efforts mainly focus on devising scalable compilation techniques to speed up stream computation. While multicore can address the rising computing demand for stream computation, it also can result in an unnecessary waste of energy due to low core utilization when more cores are used than needed at low streaming rates.

In this chapter, we present StreaMorph [24], a programming methodology that exploits the high-level abstractions of stream programs to generate runtime adaptive ones for energy-efficient executions. Based on one single specification of a stream program, the compiler exploits domain-specific knowledge to generate an adaptive program that is able to adjust its occupied computing resources, processors, memory and so on, at runtime. Such adaptive programs help reduce energy consumption by adapting their computing power to demand changes at runtime. This programming methodology comes from our new perspective on stream programs. The previous work on stream compilation assumes stream programs are isolated entities. This assumption misses an important optimization opportunity. As isolated entities, stream programs are set to run as fast as possible by using as much resources as possible. Running stream programs aggressively may result in a considerable waste of energy. In contrast, we treat stream programs as integrated components within certain systems. For example, an MPEG2 encoder/decoder does not need to run faster than its camera/display frame rates. Digital signal processing (DSP) programs, such as Radar, FM radio, and Vocoders, do not need to run faster than their input signals' sampling rates. Because using more cores than needed for certain input rates to run these programs will waste energy due to static energy leakage and inter-core data communication overhead, it is beneficial to find the minimal number of cores required for a specific workload rate to reduce energy consumption.

Determining the optimal number of cores used for a stream program at compile-time is often not possible due to several factors. These factors include varied processor speed used to run stream programs, users' control on stream quality and speed, varied runtime signal rates, etc. As a result, compiled stream applications are necessary to be able to adapt computing speed accordingly to external changes at runtime to save energy. Expressing stream programs as synchronous dataflow (SDF) [70] models presents an opportunity for such an energy optimization. In SDF, a program is decomposed into multiple autonomous actors connected using FIFO channels. As a result, stream programs can dynamically relocate actors and channels between cores at runtime to minimize the number of cores used while maintaining required throughput demands. Using fewer cores reduces leakage energy as well

as inter-core communication energy overhead.

A prior inter-core actor migration technique, the one implemented in Flextream [49] by Hormati et al. in the context of changing processor availability, can adjust the number of cores used, but it is not optimal as it does not adjust memory usage accordingly. In cloud computing as well as in multicore embedded systems, processors are not the only resource that applications share. In these computing environments, it is also desirable to reduce the memory usage of each application to improve the overall utilization of a whole system [43, 42].

To improve inter-core actor migration technique, our StreaMorph technique not only migrates actors between cores but also *transforms* stream graphs, thereby adjusting the set of actors, inter-actor channels and the number of cores used. Runtime stream graph transformation entails mapping and transferring data on inter-actor channels of different stream graph configurations. This process is non-trivial because the inter-filer channel states become complicated when stream programs execute through several stages such as initialization and software pipelining [46].

We tackle the problem by proposing an analysis that helps reduce the number of tokens copied between optimized configurations of a single stream program. The main idea of the proposed analysis is to derive sequences of actor executions to drain tokens on channels as much as possible. This StreaMorph scheme helps optimize energy consumption in several ways. Either minimizing the number of cores used at a fixed frequency or lowering operating voltages and frequencies of processors by using more cores to handle the same workload rates can help reduce energy consumption. In addition, high processor temperatures due to running at high utilization for long periods can degrade processors' performance and lifespan [45, 11]. As a result, increasing the number of cores used to lower core utilization, thereby reducing processor temperatures, can mitigate the problem.

We apply the StreaMorph scheme to a set of streaming benchmarks [47] on the Intel Xeon E5450 processors. The results show that, on average, using StreaMorph to minimize the number of cores used whenever possible from eight cores to four reduces energy consumption by 29.90%, and from eight cores to one reduces energy consumption by 76.33% on average. Using StreaMorph to spread workload on more cores when they become available from four cores to six or seven cores and applying DVFS can reduce energy consumption by 10%. Our StreaMorph scheme also leads to an 82.58% buffer size reduction when switching from an eight-core configuration to one compared to Flextream [49].

This chapter makes the following contributions

- We present the concept of adaptive programs that are beneficial in cloud computing and embedded systems. Our work demonstrates a case for employing high-level abstractions to design programs that adapt to demand and resource changes at runtime.
- We identify an energy optimization opportunity by taking into account external settings, e.g., input data rates or the number of available cores, of stream programs with a task-level program transformation technique.
- We present an analysis that helps simplify program state copying processes when switching between configurations.

- We implement the method in the StreamIt compiler and demonstrate experimentally the effectiveness of the method using a set of StreamIt benchmarks.
- We also derive an approximate energy model analysis for stream programs on multicore and experimentally validate the model. The model can serve as a guidance for finding optimal energy-efficient configurations.

3.2 Adaptive Stream Programs

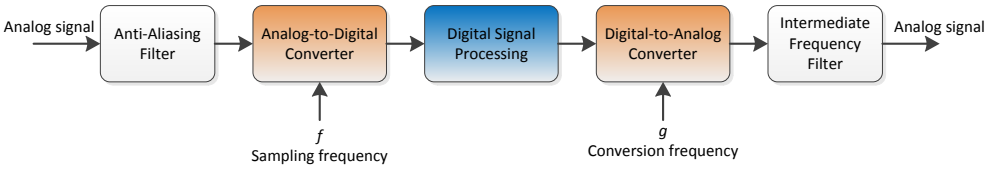


Figure 3.1: Basic digital signal processing structure.

Stream programs are often compiled to maximize speed [47, 68] assuming fixed numbers of cores. This approach implicitly assumes that sources/sinks of data for stream programs can always produce/consume data. This assumption is not often true in practice. For example, an audio source cannot produce data faster than its sampling rate. A sink, e.g., a monitor, does not consume faster than its designated frame rate. To understand this situation better, we discuss where stream programs fit into external settings.

Figure 3.1 shows a general structure of DSP systems. In the figure, the analog-to-digital converter (ADC) samples analog signals at a sampling frequency f and outputs digital signals to the DSP module, which executes stream programs. The DSP module manipulates digital signals and subsequently feeds processed digital signals to the digital-to-analog converter (DAC). The DAC converts input digital signals to analog signals at a conversion frequency g.

This general structure of DSP systems suggests DSP modules need not process more slowly than sampling f and/or conversion g frequencies. It is generally not possible to get ahead of the source of data, when that source is a real-time source, and it is generally not necessary to get ahead of the sink. Even with real-time sources and sinks, there may be some benefit to getting ahead, because results can be buffered, making interruptions less likely in the future when there is contention for computing resources. However, getting ahead produces no *evident* benefit unless such contention occurs, and it comes at an energy cost. When both the source and the sink are operating in real time, it also comes at a cost in latency. Having more cores than necessary can waste energy due to static energy leakage when idle. In addition, inter-core data transfer overhead when using more cores than needed can be another source of energy inefficiency. Even when buffering is used, in steady state, it is beneficial for processing speed to match IO rates, otherwise buffers will overflow or underflow.

We identify two scenarios that can lead to the mismatch between IO rates and processing speed.

- **Varied IO rates at runtime:** Users can use fast forward functionality to quickly browse through a video or audio file. Graphic applications need to render at faster/slower frame rates. When users want to increase/decrease songs' tempos in Karaoke systems, sound synthesis engines have to adjust accordingly. Sampling rates are increased/decreased to alter quality.
- **Different processor speeds:** Even though compilers can match between processing speed and certain IO rates at compile time, this optimization can only be done for specific processor models. Processors of the same instruction set architecture can vary in several dimensions such as clocking frequencies, microarchitecture, fabrication processes. These variations lead to wildly varied execution times of a single program on different processors with the same instruction set.

Energy Optimization

To get the energy benefits of matching processing speed to IO rates, two common techniques are: 1) To consolidate tasks by relocating tasks to fewer cores to improve core utilization and enable turning off idle cores [107, 4]; 2) To vary processing speed by applying dynamic voltage and frequency scaling(DVFS) [7, 30, 48, 120] or processor composition [25, 63] or heterogeneous processors of different speeds [69, 121, 78].

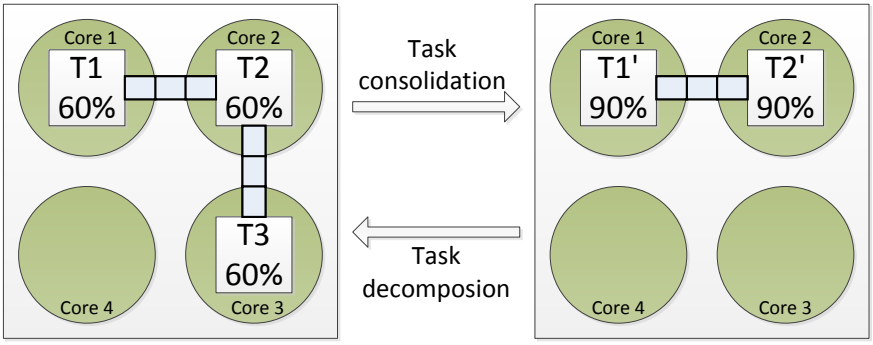


Figure 3.2: Task consolidation and decomposition.

Task Consolidation

Especially for stream programs, task consolidation can help reduce inter-core communication overhead. However, applying the task consolidation technique to stream programs optimized for speed is not straightforward. For example, in the left of Figure 3.2, a stream program is composed of three tasks `T1`, `T2` and `T3` connected through two FIFO channels. Suppose that each task resides on a separate core in a quad-core processor and utilizes 60% of its core

at current IO rates. Relocating normally one of the three tasks to another occupied core will overload the core. Consequently, the processing system cannot guarantee the program's required IO rates. Instead, the program should switch to a different configuration composed of two tasks `T1'` and `T2'`; each utilizes 90% of its residing core as in the right of Figure 3.2.

This configuration switching is complicated because the two configurations may be composed of different sets of actors and channels due to speed optimization techniques that depend on the number of cores used. For example, the configuration composed of `T1`, `T2` and `T3` is optimized for 3 cores, while the configuration composed of `T1'` and `T2'` is optimized for 2 cores.

Task Decomposition

Task consolidation is necessary as discussed in the previous section, however, there are a number of scenarios where it is useful to decompose tasks.

- Suppose that running `T1'` and `T2'` on cores 1 and 2 at 90% utilization drastically increases those cores' temperatures [74, 45]. At this point, we want to run 2 tasks on 3 cores to reduce utilization as in Figure 3.2, and thereby consequently reducing the cores' temperatures.
- If the processors posses a DVFS capability, and there are more cores available because some other applications terminated, decomposing two tasks `T1'` and `T2'` into three tasks `T1`, `T2` and `T3` and reducing the operating frequencies of the processors can lead to energy reduction; power is proportional to the cube of frequency.

Adjusting Shared Resources Proportionally

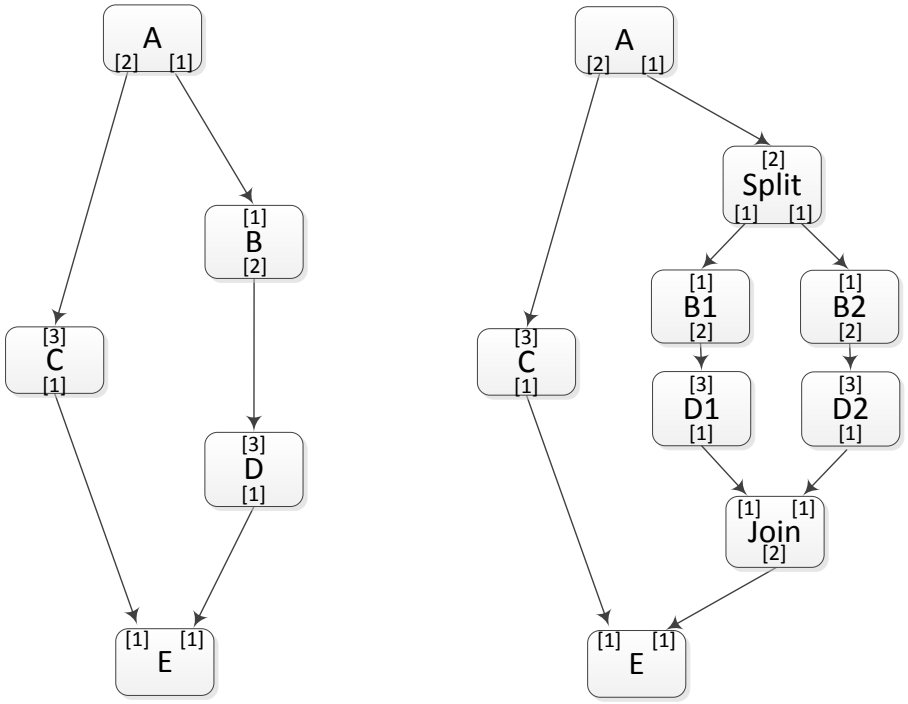
Section 3.2 suggests that inter-core actor migration to adjust processor utilization and speed can help reduce energy consumption. While the inter-core actor migration implemented in Flextream [49] can adjust the number of cores used at runtime, it is not optimal, because the technique does not reduce memory usage accordingly. This limitation reduces the value for cloud computing, where applications are allocated cores, memory, networking bandwidth, and so on, *proportionally* [43, 42]; for example, one application using two cores is allocated 4GB of memory while another application using four cores is allocated 8GB. As a consequence, just adjusting the number of cores used by relocating actors between cores without adjusting memory usage as in Flextream [49] is no longer sufficient. In particular, reducing memory usage is crucial in embedded systems where memory is scarce.

In contrast to Flextream [49], our StreaMorph technique adjusts both the number of cores used and memory usage simultaneously. We illustrate the difference using an example. To speed up stream computation, stateless actors are often replicated as in Figure 3.3(b) to utilize available cores, where actors $\mathtt{B}_1$, $\mathtt{B}_2$, $\mathtt{D}_1$ and $\mathtt{D}_2$ are duplicated from the respective ones in the original stream graph in Figure 3.3(a). Duplicating actors is necessary when stream IO rates are high and we need to use more cores to handle such high IO rates. When IO rates become low, a Flextream inter-core actor migration technique retains the

stream graphs in Figure 3.3(b) and relocates actors to reduce the number of core used. For example, originally the application in Figure 3.3(b) runs on two cores, but now the input rate is lowered and one core can handle the workload. In this case, the actors are relocated in Figure 3.3(b) and the stream graph is not modified. In contrast, our StreaMorph technique modifies the stream graph into the one in Figure 3.3(a). As a result, our technique would be more memory-efficient because it eliminates the buffers between actors `Split`, $\texttt{B}_2$, $\texttt{D}_2$, and `Join`. In addition, we will show that our technique also reduces the buffer sizes of the other buffers.

3.3 StreaMorph: Designing Adaptive Programs with High-Level Abstractions

Section 3.2 suggests that adapting stream programs to external changes can lead to energy and resource reduction. In this section, we will show how to attain such adaptivity by exploiting stream program abstractions.



(a) Original SDF model (b) Transformed SDF model

Figure 3.3: Stream graph transformation

Stream Abstractions and Data Parallelism

Model of Computation

The SDF [70] model of computation is often used to model executions of stream programs [113]. This abstraction enables several compiler optimization techniques [47, 68] for buffer space, scheduling and mapping to underlying architecture.

Sliding Windows

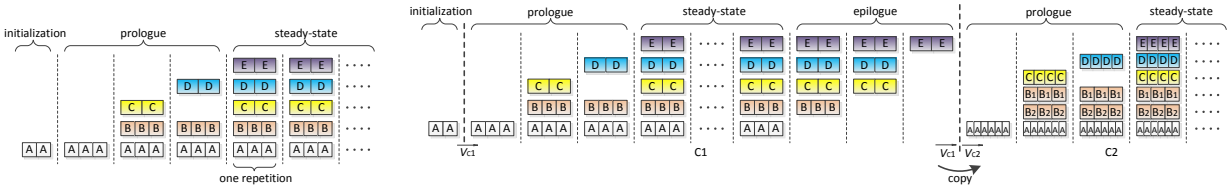
The StreamIt language [113] is a language for writing stream applications extending the SDF model of computation with a sliding window feature, in which actors can *peek* (read) tokens ahead without consuming those tokens. As a result, many states of programs are stored on channels in the form of data tokens instead of being stored internally within actors. Consequently, many actors become *stateless* and eligible for the actor replication technique to speed up computation as in Section 3.3. As many states of programs are stored on channels, it is easier for compilers to migrate states between different configurations during the morphing process.

Data Parallelism Exposed by Stream Abstractions

In this section, we will discuss how the stream abstractions in the previous section can help adjusting parallelism in stream programs, thereby adjusting computing speed. To speed up a stream application, we can replicate *stateless* actors so that multiple instances of one stateless actor execute in parallel. This actor replication technique is feasible because each actor in an SDF application consumes/produces a known numbers of tokens whenever it executes at each of its input/output port. As a result, after replicating actors, the compiler knows how to distribute/merge data tokens to/from each replicated actor.

Let us take the example in Figure 3.3(a) to illustrate the problem. Suppose that `B` and `D` are stateless, as a result, we can duplicate the actors to obtain the configuration in Figure 3.3(b). Because `B` consumes one token and `D` produces one token each time they execute, we can distribute data tokens to each duplicated `B` and collect data tokens from each `D` evenly in a round-robin fashion using `Split` and `Join` actors in Figure 3.3(b). With this duplication, we can get 2x speed-up for the computation of the two actors. Stateless actor replication is a popular technique that has proved to achieve significant speed-up for several StreamIt benchmarks [47, 68]. Stateless actors can be replicated many times to fill-up all available cores, consequently, this technique is dependent on the number of cores used.

Execution Scaling: The actor replication technique may require changing the number of times each actor executes within one iteration. For example, `A`, `C`, and `E` in Figure 3.3(b) now execute 6, 4, and 4 times respectively in one iteration. This means those actors now execute twice as often within one iteration. This effect is called execution scaling.



(a) Conventional execution stages. (b) Configuration switching strategy.

Figure 3.4: Execution strategies for stream programs

Implementing Adaptive Stream Programs

In the previous section, we discussed how the stream abstractions can help improving stream program speed by exploiting data parallelism. However, switching between configurations optimized for different numbers of processors is complicated by several compilation techniques applied to stream programs [46]. We will show how to mitigate the complication in this section.

Execution Stages

Initialization Stage: As downstream actors can peek tokens without consuming them, upstream actors have to execute a number of times initially in the *initialization* stage to supply more tokens to downstream peeking actors. For example, suppose that actor `B` in Figure 3.3(a), each time it executes, peeks ahead 3 tokens and consumes only the first one of the 3. Consequently, it requires at least 2 additional tokens on the channel between actors `A` and `B`. To satisfy this requirement, `A` has to execute twice in the initialization stage to load the channel with two tokens.

Software Pipelining: Gordon et al. use software pipelining to address a drawback of hardware pipelining [47]. In hardware pipelining, only contiguous actors should be mapped onto one core, e.g., in Figure 3.3(a), actors `A` and `D` should not be mapped into a core when actor `B` is mapped to another core. This mapping restriction can potentially lead to unbalanced allocation. Software pipelining allows mapping any actors to any core, e.g., actors `A` and `D` can be mapped into a core even when actor `B` is mapped to another core. This is done by constructing a *loop prologue* in which each actor executes a certain number of times to make data tokens available for downstream actors to run in one iteration. In other words, within one iteration, actors do not communicate directly to each other, instead, they output tokens into buffers that will be read by downstream actors in the next iteration. Figure 3.4(a) shows how the prologue stage is executed and its transition to the steady-state stage. In the steady-state stage, actors execute one stream graph iteration forming a steady-state *repetition*. Each actor is able to execute completely independently within one steady state repetition without the need for waiting for upstream actors to produce data

in that repetition. For example, for the model in Figure 3.3(a), in order for `E` to execute 2 times independently, `C` and `D` have to execute twice. Consequently, `B` and `A` have to execute 3 times. Similarly, for `C` and `D` to execute twice independently, `B` and `A` have to execute 3 times. Finally, for `B` to execute 3 times independently, `A` has to execute 3 times. Adding up, in the prologue stage, `A` executes (3+3+3) = 9 times, `B` executes (3+3)=6 times. `C` and `D` execute 2 times. After the loop prologue `A`, `B`, `C`, `D`, `E` can execute independently within one steady-state repetition.

Deriving Reverse Sequences

Normally, a compiled program repeatedly executes its steady-state repetitions regardless of external environment changes as in Figure 3.4(a). This conventional execution model may no longer be efficient in cloud computing, mobile computing and cyber-physical systems where applications run in various environments or have to interact with physical environments. For example, let us consider the situation when there are more cores becoming available in the system because some other applications terminated. One possible way to save energy is to utilize the newly available cores to process a part of the workload of the running application at lower frequencies to save energy. Suppose utilizing the available cores would require the application to switch from configuration $\mathcal{C}_1$ in Figure 3.3(a) to the configuration $\mathcal{C}_2$ in Figure 3.3(b). This transition takes place at the end of a repetition in the $\mathcal{C}_1$'s steady-state stage. Note that, in software pipelining, the prologue stage fills the channels of $\mathcal{C}_1$ with tokens, for example, the channel between `B` and `D` contains 6 tokens; the channel between `D` and `E` contains 2 tokens. As `B` and `D` are duplicated, determining how to copy and distribute those tokens on the corresponding channels in Figure 3.3(b) is complicated. It is even more problematic to derive from such a state of $\mathcal{C}_2$ a sequence of actor executions such that, after executing such a sequence, actors in Figure 3.3(b) can execute in a software pipelining fashion. Deriving such a switching procedure for each pair of configurations may be costly.

We simplify the switching process using the strategy in Figure 3.4(b). Instead of deriving complicated inter-configuration token copying procedures, we use an *epilogue* stage to reduce the number of copied tokens. The epilogue stage is derived to undo the effect of the prologue stage. As a result, we only need to copy the fixed and known number of tokens across the configurations produced by the initialization stage. If there is no peeking actor, only initial tokens and states of stateful actors may still require copying.

Definition 3 *Sequences of executions that undo the effect of the program's current configuration's prologue stage are called reverse sequences.*

Let $\vec{V_{\mathcal{C}}}$ be the vector of the number of tokens on each channel of configuration $\mathcal{C}$ right before its prologue stage.

Definition 4 *States of a configuration $\mathcal{C}$, whose vectors of the numbers of tokens on the channels are equal to $\vec{V_{\mathcal{C}}}$, are called $\mathcal{C}$'s pre-prologue states.*

Note that, for any configuration $\mathcal{C}$, from a pre-prologue state, if each actor executes the same number of iterations, $\mathcal{C}$ is again in a pre-prologue state. As a result, a reverse sequence can be derived by finding $d_{\mathtt{A}}, \forall \mathtt{A} \in \mathcal{C}$, the additional number of times each actor has to execute more so that all the actors will have executed the same number of iterations. Suppose that number of iterations is J, counting from $\mathcal{C}$'s first pre-prologue state (right before executing $\mathcal{C}$'s first prologue stage). Suppose that $\mathcal{C}$ has n actors $\mathtt{A}_i$ where $i = 1, \ldots, n$, and each actor $\mathtt{A}_i$ executes $p_{\mathtt{A}_i}$ and $s_{\mathtt{A}_i}$ times in its prologue stage and one iteration respectively. We have:

$$p_{\mathtt{A}_i} + XI\ s_{\mathtt{A}_i} + d_{\mathtt{A}_i} = Js_{\mathtt{A}_i}\ \forall \mathtt{A}_i \in \mathcal{C} \tag{3.1}$$

where I is the current repetition of the steady-state stage and X is the execution scaling factor. From equation (3.1), since X, I , and J are all integers, $p_{\mathtt{A}_i} + d_{\mathtt{A}_i}$ has to divide $s_{\mathtt{A}_i}$. Let $\alpha_{\mathtt{A}_i}$ be an integer such that $p_{\mathtt{A}_i} + d_{\mathtt{A}_i} = \alpha_{\mathtt{A}_i} s_{Ai}$. Equation (3.1) becomes:

$$\begin{aligned} \alpha_{\mathtt{A}_i} s_{\mathtt{A}_i} + XI\ s_{\mathtt{A}_i} &= Js_{\mathtt{A}_i} \\ \Leftrightarrow \alpha_{\mathtt{A}_i} &= J - XI \end{aligned} \tag{3.2}$$

Note that equation (3.2) holds $\forall \mathtt{A}_i \in \mathcal{C}$. Expanding the equation for all actors in the configuration $\mathcal{C}$, we arrive at: $\alpha_{\mathtt{A}_1} = \alpha_{\mathtt{A}_2} = \ldots = \alpha_{\mathtt{A}_n}$. Because $\alpha_{\mathtt{A}_i} = \frac{p_{\mathtt{A}_i} + d_{\mathtt{A}_i}}{s_{Ai}} = \frac{d_{\mathtt{A}_i} + (p_{\mathtt{A}_i} \bmod s_{Ai})}{s_{Ai}} + \lfloor \frac{p_{\mathtt{A}_i}}{s_{A1}} \rfloor$, then:

$$\frac{d_{\mathtt{A}_1} + (p_{\mathtt{A}_1} \bmod s_{A1})}{s_{A1}} + \lfloor \frac{p_{\mathtt{A}_1}}{s_{A1}} \rfloor = \ldots = \frac{d_{\mathtt{A}_n} + (p_{\mathtt{A}_n} \bmod s_{An})}{s_{An}} + \lfloor \frac{p_{\mathtt{A}_n}}{s_{An}} \rfloor \tag{3.3}$$

As it is desirable to switch between configurations as soon as possible, we find the smallest $d_{\mathtt{A}_i} \geq 0$ satisfying equation (3.3) by finding the j such that:

$$j = \underset{i \in [1..n]}{\operatorname{argmax}} \left(\frac{p_{\mathtt{A}_i} \bmod s_{Ai}}{s_{\mathtt{A}_i}} + \lfloor \frac{p_{\mathtt{A}_i}}{s_{Ai}} \rfloor \right) \tag{3.4}$$

Because $\alpha_{\mathtt{A}_i}$ is an integer and $\alpha_{\mathtt{A}_i} = \frac{d_{\mathtt{A}_1} + (p_{\mathtt{A}_1} \bmod s_{A1})}{s_{A1}} + \lfloor \frac{p_{\mathtt{A}_1}}{s_{A1}} \rfloor$, we can conclude that $(d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})) \bmod s_{Aj} \equiv 0$. Hence,

- If $p_{\mathtt{A}_j} \bmod s_{Aj} = 0$, we find the smallest $d_{\mathtt{A}_j} = 0$.
- If $p_{\mathtt{A}_j} \bmod s_{Aj} > 0$, we find the smallest $d_{\mathtt{A}_j} = s_{Aj} - (p_{\mathtt{A}_1} \bmod s_{A1})$. It is easy to prove that $d_{\mathtt{A}_j} \geq 0$.

Now we derive other $d_{\mathtt{A}_i}$ from equation (3.3) as follows:

$$d_{\mathtt{A}_i} = \left(\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}} + \lfloor \frac{p_{\mathtt{A}_j}}{s_{Aj}} \rfloor - \lfloor \frac{p_{\mathtt{A}_i}}{s_{Ai}} \rfloor \right) s_{Ai} - (p_{Ai} \bmod s_{Ai}) \tag{3.5}$$

$$= \left(\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}} + \lfloor \frac{p_{\mathtt{A}_j}}{s_{Aj}} \rfloor \right) s_{Ai} - p_{Ai} \tag{3.6}$$

As $\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}}$ is equal to either 0 or 1, from equation (3.6), it is easy to prove that $d_{\mathtt{A}_i}$ is an integer. Now we need to prove $d_{Ai} \geq 0\ \forall i = 1, \ldots, n$. From (3.3) and (3.4):

$$\lfloor \frac{p_{\mathtt{A}_j}}{s_{Aj}} \rfloor - \lfloor \frac{p_{\mathtt{A}_i}}{s_{Ai}} \rfloor \geq \frac{p_{\mathtt{A}_i} \bmod s_{Ai}}{s_{\mathtt{A}_i}} - \frac{p_{\mathtt{A}_j} \bmod s_{Aj}}{s_{\mathtt{A}_j}} \tag{3.7}$$

Plugging into equation (3.5), we arrive at:

$$d_{\mathtt{A}_i} \geq \left(\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}} + \frac{p_{\mathtt{A}_i} \bmod s_{Ai}}{s_{\mathtt{A}_i}} - \frac{p_{\mathtt{A}_j} \bmod s_{Aj}}{s_{\mathtt{A}_j}} \right) s_{Ai} - (p_{Ai} \bmod s_{Ai})$$
$$\geq \frac{d_{\mathtt{A}_j} s_{Ai}}{s_{Aj}} \geq 0 \tag{3.8}$$

Being able to find d_{Ai} does not necessarily mean that reverse sequences always exist; there may be additional data dependency constraints. Within the SDF literature, even if we can find numbers of times actors execute within one iteration, it is still possible that the stream graph is not schedulable, e.g., when the stream graph has loops forming circular dependencies. The following theorem implies that it is always possible to undo the effect of prologue stages.

Theorem 2 *A reverse sequence always exists for a configuration $\mathcal{C}$ if the configuration has prologue and steady-state schedules (an executed stream graph).*

Proof: We prove this by contradiction. Suppose that we cannot derive a concrete reverse sequence based on the values d_{Ai} found as above. The only reason would be because of data dependency between the executions of the actors. Let $\mathtt{A}_i^e$ denote the e^{th} execution of actor $\mathtt{A}_i$ from the beginning right before the prologue stage. There exist two possible cases:

1) There exists a data dependency loop between actor executions $\mathtt{A}_{j_1}^{e_{j_1}} \prec \mathtt{A}_{j_2}^{e_{j_2}} \prec \ldots \prec \mathtt{A}_{j_m}^{e_{j_m}} \prec \mathtt{A}_{j_1}^{e_{j_1}}$ of m actors. As the existence of this dependency loop depends solely on the property of $\mathcal{C}$, consequently, the execution of $\mathcal{C}$ will be stalled due to the dependency loop. This contradicts with the fact that $\mathcal{C}$ can be repeated in the steady-state stage forever.

2) Some actor B is at its b^{th} execution and has not completed d_B executions of its epilogue stage to reach its execution $(Js_B)^{\text{th}}$; in other words, $b < Js_B$. B cannot proceed further because it requires more data tokens from its upstream actor A, while A has completed d_A executions in its epilogue stage to reach its $(Js_A)^{\text{th}}$ execution. This implies that A, having executed J iterations, still does not produce enough tokens for B to execute J iterations. This contradicts the property of SDF that the number of tokens A produces in one iteration is equal to the number of tokens B consumes in one iteration. ■

We now need to derive concrete reverse sequences based on d_{Ai}. It is desirable to use current buffers without increasing buffer sizes for executing reverse sequences. As each buffer is large enough to store tokens from the upstream actor for the downstream actor to execute at least one iteration even if the producing actor has not produced any more tokens, it is safe to execute upstream actors for at most one iteration continuously. After that, downstream actors have to execute to free up buffers. Algorithm 3.5 derives reverse sequences when stream graphs do not contain loops. However, most of the stream benchmarks do not contain loops [113] and all the benchmarks used in [47, 68] are loop-free. When stream graphs have loops, the classical symbolic execution method of SDF [70] can be used. The classical method may yield more complicated sequences as it randomly selects actors to execute. This random

execution ordering can cause severe performance degradation during configuration switching due to losing cache locality. To mitigate the potential negative effect of the classical symbolic execution method, the following algorithm seeks to execute each actor several times in a row by traversing actors in dataflow order.

```
   Data: reverseMap: Map from actors to numbers of reverse executions
   schedule: containing prologue and steady-state schedules
   Result: reverseSeq: reverse sequence of actor executions
 1 reverseSeq ← new List()
 2 actors ← getActorsInDataflowOrder()
 3 while reverseMap.size() > 0 do
 4     thisStepMap ← new Map()
 5     for f ∈ actors do
 6         nReverseExes ← reverseMap.get(f)
 7         nIterExes ← schedule.getNumberExesInOneIteration()
 8         m = min(nReverseExes, nIterExes)
 9         thisStepMap.put(f, m)
10         nReverseExesLeft ← nReverseExes − m
11         if nReverseExesLeft > 0 then
12             reverseMap.put(f,nReverseExesLeft)
13         else
14             reverseMap.remove(f)
15         end
16     end
17     reverseSeq.add(thisStepMap)
18 end
19 return reverseSeq
```

Figure 3.5: Deriving reverse execution sequences when stream graphs are loop-free.

Illustrative Example

We illustrate the analysis method using the configuration in Figure 3.3(a). For the program: $p_{\mathtt{A}}$=9,$p_{\mathtt{B}}$=6, $p_{\mathtt{C}}$=$p_{\mathtt{D}}$=2, $p_{\mathtt{E}}$=0; $s_{\mathtt{A}}$=$s_{\mathtt{B}}$=3, $s_{\mathtt{C}}$=$s_{\mathtt{D}}$=$s_{\mathtt{E}}$=2; X= 1. As $\frac{p_{\mathtt{A}} \mod s_A}{s_A} + \lfloor \frac{p_A}{s_A} \rfloor$=3 is max and $p_A \mod s_A \equiv 0$, we set $d_{\mathtt{A}} = 0$. Finally, from (3.6), we find $d_{\mathtt{B}}$=3, $d_{\mathtt{C}}$=$d_{\mathtt{D}}$=4, $d_{\mathtt{E}}$=6. Readers can verify that condition (3.1) is satisfied. Now, applying Algorithm 3.5, we find that, in the first reverse step, `B`, `C`, `D`, `E` execute 3, 2, 2, 2 times respectively. `B` has executed all its reverse executions, so it is removed from the *reverseMap*. In the second step, `C`, `D`, `E` execute 2, 2, 2 times respectively, and `C`, `D` are removed from the *reverseMap*. In the last step, only `E` executes 2 times and is removed from the *reverseMap*. Now the *reverseMap* is empty so the algorithm terminates. Applying the reverse sequence will drain all the tokens on the channels from `B` to `D` and from `D` to `E`. Figure 3.4(b) displays how the reverse execution sequence is executed.

Peeking Token Copying

As reverse sequences can undo the effect of prologue stages, if stream programs do not contain peeking actors, then after epilogue stages, the effect of respective prologue states is undone.

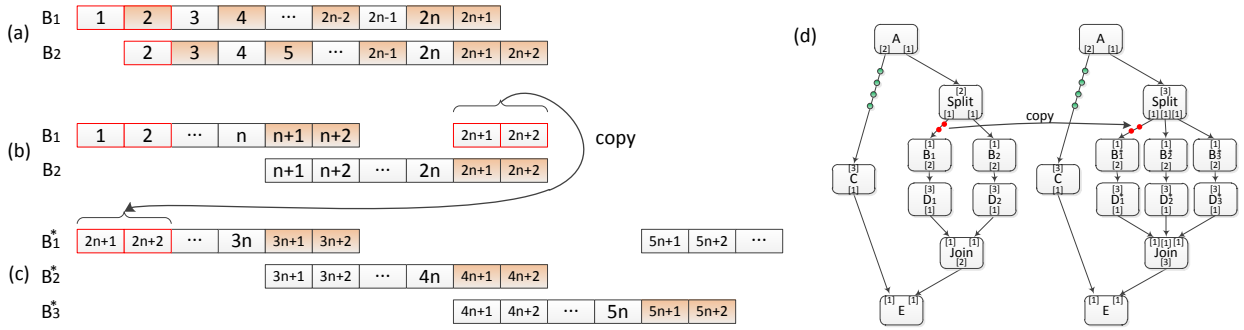


Figure 3.6: Peeking token copying in sliding window computation. Each cell denotes a token. Colored tokens are peeked only and never consumed. (a) Data token distribution in fined-grained interleaving execution when B is replicated by 2; (b) Data token distribution in coarse-grained interleaving execution when B is replicated by 2; (c) Data token distribution in coarse-grained interleaving when execution B is replicated by 3; (d) Copy peeking tokens when switching configurations.

As a result, we only need to copy actor states, which are small, and initial tokens, which often do not exist. When peeking actors exist, for example, B peeks 3 tokens and only consumes 1 each time it executes, the channel between A and B will always contain tokens after the initialization stage. Copying those peeking tokens requires further elaboration about how stream graphs with peeking actors are optimized.

Efficient Sliding Window Computation: Gordon, in his PhD thesis [46], presents a method to reduce inter-core communication that degrades performance in SMP and Tilera machines for applications with peeking actors. We will use an example to illustrate Gordon's method.

Consider the configuration in Figure 3.3(b), where actor B is duplicated into B_1 and B_2. Because B peeks tokens, it is necessary to send more tokens to the input channels of replicating actors B_1 and B_2 than the number of tokens B_1 and B_2 consume. A fine-grained data parallelism approach where B_1 and B_2 alternatively consume tokens from A at the step size of 1 will double the amount of communication between A and B as in Figure 3.6(a). For example, suppose that in one iteration A produces $2n$ tokens on its output channel, and there are 2 more tokens, numbered 1 and 2, produced by A in the initialization stage. B_1 consumes token 1 and reads tokens 2 and 3; B_2 consumes token 2 and reads tokens 3 and 4; B_1 consumes token 3 and reads tokens 4 and 5; and so on. At the end of the iteration in the steady-state stage, B_2 consumes token $2n$ and reads tokens $2n+1$ and $2n+2$. As a result, both B_1 and B_2 require $2n+1$ tokens out of $2n+2$. The total amount of traffic is therefore $4n+2$ tokens in comparison with $2n+2$ tokens for the original stream graph. This communication overhead may degrade performance significantly for applications with peeking actors in SMP and Tilera machines [46].

To reduce the communication overhead, a coarse-grained data parallelism approach [46] is used instead as in Figure 3.6(b). Now $\texttt{B}_1$ consumes the first n tokens and reads tokens $n+1$, $n+2$; $\texttt{B}_2$ consumes tokens from $n+1$ to $2n$ and reads tokens $2n+1$ and $2n+2$. As a result, `A` only needs to send $n+2$ tokens to both $\texttt{B}_1$ and $\texttt{B}_2$ within one iteration of the steady-state stage. The amount of communication is therefore $2n+4$ tokens in comparison with $2n+2$ tokens for the original graph. If n is large, the overhead becomes insignificant. As $2n$ is the number of tokens `A` produces within one iteration, we can scale up the number of times each actor executes within one iteration to increase n. Note that in Figure 3.6(a)(b) and (c), colored tokens are peeked only and never consumed.

Copying Peeking Tokens: The StreamIt compiler enforces an additional constraint [46] that the number of tokens on the input channel of a replicated actor right after the initialization stage has to be smaller than the number of tokens the actors will consume within one iteration. This can be achieved by scaling up the same number of times each actor executes in one iteration. This constraint implies that, in pre-prologue states, only the input channels of the first replicated peeking actors contain tokens regardless of configurations. As a result, after applying reverse sequences to bring a configuration to a pre-prologue state, we only need to copy tokens from the input channel of the first replicated actors in the current configuration to the input channel of the respective first replicated actors in the incoming configuration. For example, only the tokens in the input channel of $\texttt{B}_1$ need to be copied to the input channel of $\texttt{B}_1^*$ as in Figure 3.6(d). The tokens on the channel between `A` and `C`, due to the executions of `A` in the initialization stage, only need to be copied if the two configurations contain different versions of `A` and `C` or the channel between `A` and `C` needs to be reallocated for a larger buffer size. As the number of peeking actors and the number of peeking tokens are small [113], it would take an insignificant amount of time to copy.

3.4 Task Decomposition for Low-Power

As discussed in Section 3.2, when more cores become available because some other programs terminated, if processors possess DVFS capability, we can reduce energy consumption by decomposing tasks, thereby spreading workload on more cores, and lowering operating voltages and frequencies. We will derive an analysis that helps justify the hypothesis supported by the experimental results in Section 3.5.

We employ the energy model for one CPU from [92]:

$$P_{cpu} = P_{dynamic} + P_{static} = (C_e V_{cpu}^2 f_{cpu}) + (\alpha_1 V_{cpu} + \alpha_2) \tag{3.9}$$

Based on equation (3.9), the dynamic energy consumption P_n if computation is spread on n cores is:

$$P_n \approx \left(\sum_{i=1}^{n} C_i \right) V_n^2 f_n \tag{3.10}$$

If we assume that inter-core communication consumes very little energy in comparison with computational energy or the same as intra-core communication, spreading stream graphs on more cores enables lowering operating frequencies and voltages, although it increases $\sum_{i=1}^{n} C_i$. Suppose that we can run a stream application at a specific IO rate with two configurations of n and m cores, where $n > m$. From equation (3.10), we arrive at:

$$\frac{P_n}{P_m} \approx \frac{n}{m} \left(\frac{V_n}{V_m}\right)^2 \frac{f_n}{f_m} \tag{3.11}$$

Note that the n-core configuration is supposed to be as fast as the m-core configuration. As a result, it requires that the number of instructions delivered by n cores in one second be equal to that of m cores: $nf_nIPC_{avg} \approx mf_mIPC_{avg}$, where IPC_{avg} is the average number of instructions per cycle and it should be the same for the two configurations because both configurations run the same workload targeting the same IO rates. Equivalently, $\frac{n}{m} \approx \frac{f_m}{f_n}$. Plugging into (3.11), we arrive at:

$$\frac{P_n}{P_m} \approx \left(\frac{V_n}{V_m}\right)^2 \tag{3.12}$$

From [56], frequencies relate to voltages as follows:

$$f \propto \frac{(V_{cpu} - V_t)^\gamma}{V_{cpu}} \tag{3.13}$$

where V_t is the threshold voltage of transistors and γ depends on the carrier velocity saturation and lies between 1.2 to 1.6 for the current technologies. As a result, running on more cores with lower frequencies can reduce energy consumption. For example, $m < n \Rightarrow f_n < f_m \Rightarrow V_n < V_m \Rightarrow P_n < P_m$. Operating voltage reduction also reduces static energy consumption as in equation (3.9). From (3.12) and (3.13), we can also see that power is approximately proportional to the cube of frequency.

Note that this is a simplified energy model for stream applications on multicore machines. This model assumes that computation-communication ratios are large enough to approximately ignore inter-core communication energy or inter-core communication consumes the same amounts of energy as intra-core communication. The first assumption depends on benchmarks' characteristics while the second assumption is often not true in practice. This approximation analysis serves as a predictive model to explain the results in Section 3.5.

3.5 Evaluations

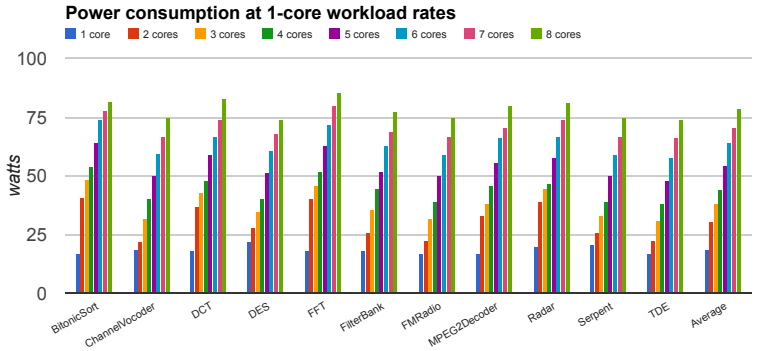
We implement the StreaMorph scheme in the StreamIt compiler [46]. To model input data token streams controlled externally at certain rates, we instrument code of source actors with the token-bucket mechanism [111]. Whenever a source actor wants to send out a number of data tokens, it has to acquire the same number of rate control tokens in a bucket. If the bucket does not have enough rate control tokens, the thread of the source actor will sleep

and wait until there are enough rate tokens in the bucket. A timer interrupt periodically fills the bucket with rate control tokens at rate r and wakes up the waiting thread. Our coarse-grain level implementation of this mechanism has a negligible effect on performance.

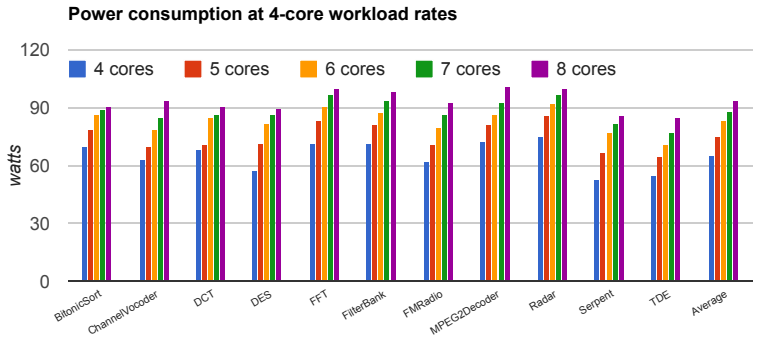
We run our experiments on a server with two Intel Xeon E5450 quad-core CPUs operating at two frequencies 2GHz and 3GHz. We use a power meter to measure the power consumption of the whole system. The system has a typical idle power consumption $P_{idle} \approx 228$ watts. For each benchmark, we measure the dynamic power consumption, P_{load}, computed using the following equation:

$$P_{load} = P_{measure} - P_{idle} \tag{3.14}$$

We use the same set of benchmarks in other papers [47, 68]. Most of the benchmarks are in the DSP domain. Each benchmark is compiled into a program composed of several configurations, where each configuration is specific to a number of cores.

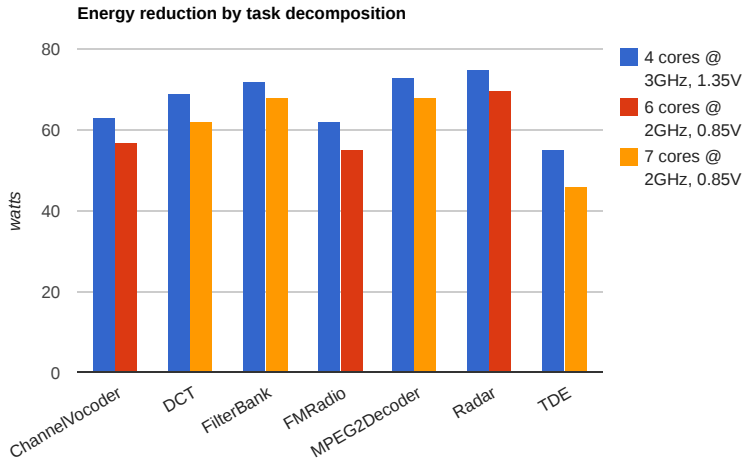


(a) Energy consumption when input rates saturate one core.

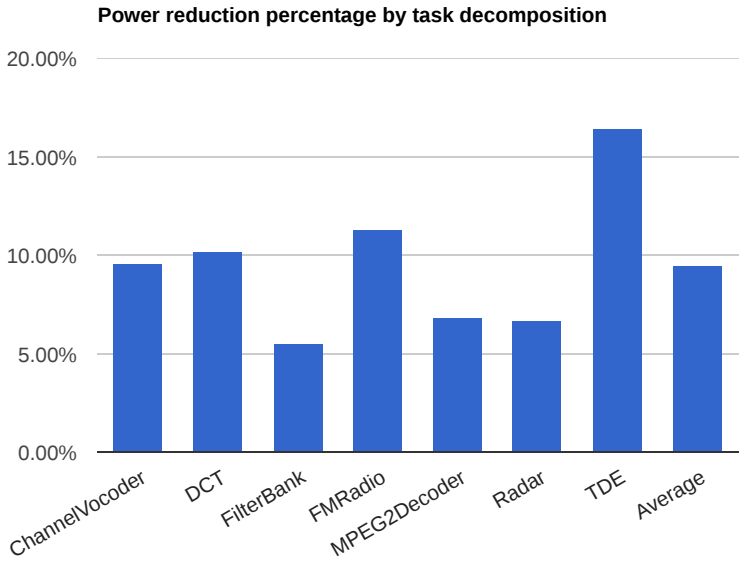


(b) Energy consumption when input rates saturate four cores.

Figure 3.7: Energy reduction by task consolidation.



(a) Measured energy consumption.



(b) Energy reduction percentage

Figure 3.8: Energy reduction by task decomposition.

Energy Reduction by Task Consolidation

In this section, we evaluate the effectiveness of the task consolidation scheme using the StreaMorph technique. Suppose that when the IO rates of a stream program are reduced, the processors become under-utilized. As a result, we can morph the stream graph of the application to minimize the number of cores used to reduce P_{load}; cores are run at 3GHz. Figure 3.7(a) and Figure 3.7(b) show the effectiveness of consolidating tasks using StreaMorph to minimize the number of cores used to one and four cores respectively when input rates become low enough. Concretely, morphing from eight cores to four cores reduces energy

consumption by 29.90% on average. Morphing from eight cores to one core reduces energy consumption by 76.33% on average.

Energy Reduction by Task Decomposition

This section demonstrates experimentally the effectiveness of the task decomposition scheme with StreaMorph. When there are more available cores in the system, because some other program terminates, the analysis in Section 3.4 suggests we should transfer a part of the workload to the newly available cores to lower operating voltages and frequencies of all the cores to save energy. For each benchmark, we use the workload rate that can be handled by 4 cores at 3GHz. We use the StreaMorph technique to switch each application to a new configuration using more cores, such that the new configuration can still handle the same workload rate at a lowered frequency, say 2GHz. Figure 3.8(a) shows the measured energy consumptions and Figure 3.8(b) shows the energy reduction percentage gained by task decomposition. On average, the energy reduction percentage is around 10%.

Our experiment also shows that this method is only effective for the benchmarks that have high computation-communication ratios [46]; in this experiment, the ratio is greater than 100. The reason for this result is that the benchmarks with small computation-communication ratios would incur significant inter-core communication energy overhead compared to computational energy.

Effect of Synchronization on Energy Consumption

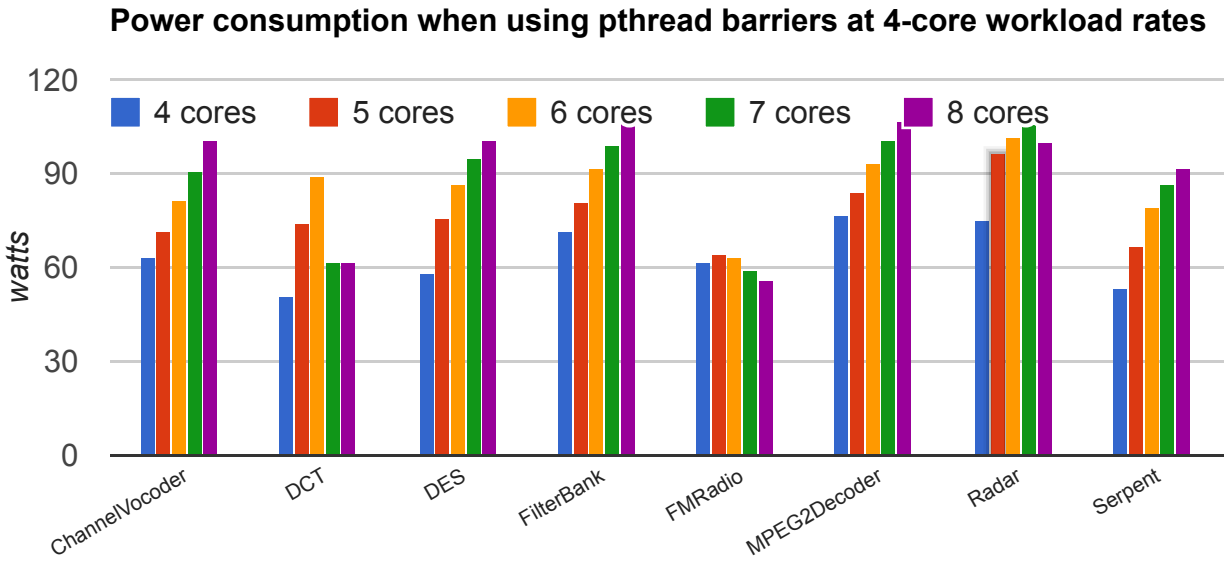


Figure 3.9: Energy consumption when using pthread barriers

Note that between steady-state repetitions, threads are synchronized by a barrier to avoid buffer overrun situations. As threads containing source actors will have to wait, by sleeping, if there are not enough rate control tokens, consequently, other threads have to wait at barriers. Our experiments in the previous sections use the original implementation of barriers in the StreamIt compiler that uses spin-locks. This scheme wastes energy because processors continuously check for conditions. To enable processors to sleep to save energy, we replace spin-lock barriers with pthread barriers. However, using pthread barriers degrades performance of a number of benchmarks such as `FFT`, `BitonicSort` and `TDE`. For `FMRadio`, using pthread barrier makes energy consumption approximately stay the same even when the number of cores used is increased. For `ChannelVocoder`, `DES`, `FilterBank`, `MPEG2Decoder`, `Radar` and `Serpent`, using pthread barriers incurs additional energy consumption in comparison with using original spin-lock barriers. This is because it would take a considerable amount of energy to wake processors up. Figure 3.5 shows the energy consumption when using pthread barriers for 4-core workload rates. On average, across the benchmarks whose performance is not degraded by using pthread barriers, using pthread barriers consumes approximately the same amount of energy as using spin-lock barriers, while spin-lock barriers are better for speed.

StreaMorph vs. Flextream Task Migration

In the previous sections, we demonstrated how StreaMorph can help save energy. However, how is StreaMorph compared to the straightforward actor migration scheme implemented in Flextream [49] by Hormati et al. Figure 3.10 shows the advantage of StreaMorph over a Flextream actor migration scheme in reducing buffer sizes when switching from multiple cores to one core. For example, when switching from eight cores to one core, because StreaMorph transforms the stream graph structures of applications, it reduces the buffer sizes by 82.58% on average over Flextream, which does not modify the stream graph structures. Even when switching from two cores to one core, StreaMorph can help reduce buffer sizes by 57.62%. Especially, for benchmarks `ChannelVocoder`, `FMRadio`, `FilterBank`, StreaMorph can help reduce buffer sizes more substantially, while it is not useful in the case of `TDE` because `TDE` does not require the actor-replication technique even for eight cores.

In addition, Flextream does not allow the optimizations dependent on the number of cores as described in Section 3.3. Furthermore, in Flextream, actors are not fused to allow fine-grain actor migration to avoid the situation in Figure 3.2, while actor fusion is beneficial to reduce synchronization and communication between actors [47]. As a consequence, Flextream suffers around 9% performance penalty from the optimal configurations [49]. Our experiment comparing the performances of StreaMorph and the Flextream actor migration techniques for switching from five cores to three cores also result in the same result. To save space, we do not present the result here.

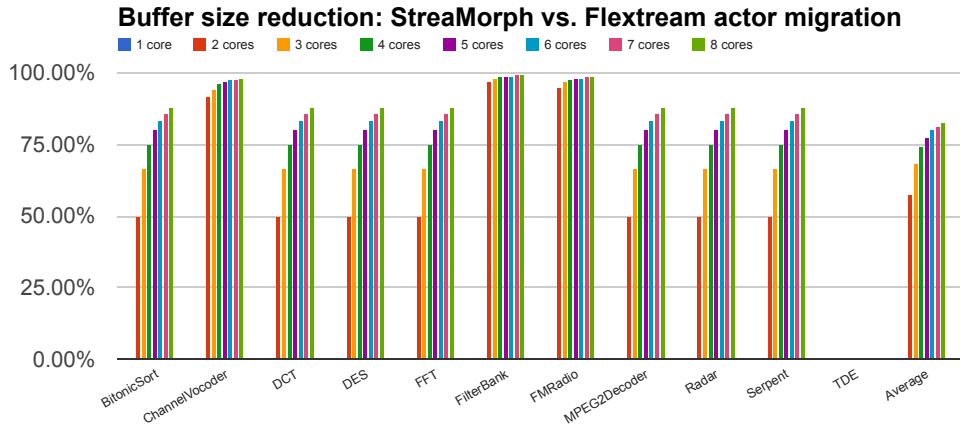


Figure 3.10: Buffer reduction.

Switching Time

We have shown that switching between configurations can help reduce energy consumption of multicore systems. However, if it takes too long to switch between configurations, QoS of stream programs may suffer. It is desirable to measure the switching time of the system. A configuration switch is composed of three steps: the epilogue stage of the current configuration, state copying, and the prologue stage of the incoming configuration. The following equation shows how switching time is broken down:

$$t_{switching} = t_{epilogue} + t_{state-copying} + t_{prologue} \tag{3.15}$$

We run each benchmark and measure $t_{epilogue}$ and $t_{prologue}$ on 1 to 8 cores. To save space, we only report the maximum and average measured values for each benchmark in Table 3.1. As, $t_{state-copying} \approx 0$, it is often too small to measure exactly, we instead report the number of bytes to copy for each benchmark in Table 3.1. We can use the data from the table to compute maximum amounts of time to switch from one configuration to another configuration. For example, for the `DES` to switch from 3 cores to 7 cores, $t_{epilogue}$ for 3 cores is bounded by 148μs and $t_{prologue}$ for 7 cores is bounded by 210μs, and $t_{state-copying} = 0$ because there is no tokens that need to copy. The total switching time is smaller than or equal to $148 + 0 + 210 = 358\mu$s for `DES`. We can see that switching times are small enough not to degrade user experience.

In addition, note that during the configuration switching process, processors still do useful work such as running epilogue and prologue stages, and as a result, the overall performance would not be affected too much.

	Prologue time (μs)		Epilogue time (μs)		Copy size (bytes)	
Benchmarks	Max	Average	Max	Average	Token	State
BitonicSort	716	126	27	22	0	4
ChannelVocoder	2999	2477	45703	39271	8820	252
DCT	28	21	104	68	0	4
DES	210	153	148	115	0	4
FFT	52	38	37	31	0	4
FilterBank	1442	1121	12385	7217	1984	508
FMRadio	1582	988	56575	26974	492	508
MPEG2Decoder	116	75	176	125	0	4
Radar	123	104	347	305	0	1032
Serpent	2303	853	648	548	0	0
TDE	5	3	770	361	0	4

Table 3.1: Switching time statistics.

3.6 Lessons Learned

Designing the StreaMorph scheme, we realized a number of principles for designing adaptive programs:

- *Modularity:* Applications should be decomposed into subprocesses to allow task migration between cores.
- *Functional subprocesses:* Ideally, the subprocesses should be designed to be stateless to expose parallelism, e.g., by replicating stateless processes.
- *Internal state exposure:* States of programs should be exposed by storing on external queues instead of within subprocesses to facilitate state migration while morphing programs.
- *Predictable inter-process communication rates:* This property eases program state transferring processes, e.g., by reducing the amount of copied state.

These principles point to other domains for adaptive programs. We find we can apply the adaptive program concept to programming models such as Cilk [38], PetaBricks [8] and SEDA [119].

3.7 Related Work

The Hormati et al.'s Flextream [49] work is closely related to our work. Hormati et al. present a method that can efficiently repartition stream programs to adapt dynamically to environment changes such as the number of available cores. The major drawback of the Flextream is that it does not reduce memory usage proportionally when reducing the number of cores used as shown in Section 3.5 as well as 9% performance degradation. In addition,

we not only present a method enabling the above adaptations but also show how to use the method to reduced energy consumption of stream programs by putting stream programs into external settings. Aleen et al. [4] propose a method to dynamically predict running times of portions of streaming programs based on input values. Portions of programs are dynamically relocated across cores based on prediction results to balance workload on multicore. Task consolidation has been deployed in the Linux kernel to reduce energy consumption by relocating workloads to fewer cores to allows other cores to turn into deep sleep states [107]. Besides the task consolidation technique, adjusting processors' speed using the DVFS capability to computation demands is another popular technique. Choi et al. [30] present a method to reduce energy consumption of a MPEG decoder program by adapting processor frequencies to MPEG video frame rates. The dynamic knobs framework [48] dynamically adjusts processor frequencies based on QoS requirements to save energy. In [10], Baek and Chilimbi present a framework that compile programs into adaptive ones that can return approximate results based on QoS requirements to reduce energy consumption.

Hardware energy-efficient research has been focusing designing processors that can adapt themselves to QoS requirements. Executing code regions of certain characteristics to suitable cores in heterogeneous multicore systems composed of cores with different points of energy/performance sharing the same ISA to save energy has been explored in [69, 121, 78]. Another approach to energy-efficient computing is to exploit the DVFS capability of modern processors [120, 7]. Burger et al. proposed an Explicit Data Graph Execution (EDGE) architecture [25] to reduce energy consumption by getting rid of complicated speculation circuits inside modern processors and using compiler techniques instead.

Our work is also related to the fair multiple resource sharing problem [43, 42] in cloud computing. Our StreaMorph scheme, when reducing the number of cores used, also reduces memory usages accordingly. This feature makes our StreaMorph scheme more suitable for cloud computing than the Flextream approach [49].

In [91], Parhi and Messerschmitt present a method for finding multiprocessor rate-optimal schedules for data flow programs. Rate-optimal schedules help programs achieve minimal periods for iterations given an infinite number of cores. Renfors and Neuvo's framework [96] focuses on determining the minimal sampling periods for a given digital actor structure assuming the speed of arithmetic operations is known and the number of processing units is infinite. This line of work is different from our work in the sense that, they assume static streaming rates and applications are optimized for specific hardware platforms while in cloud computing, mobile computing and cyber-physical systems, applications run on a wide variety of underlying hardware.

Finally, our work derives from the work by the StreamIt compiler group [47, 46, 113]. However, we focus on the energy-efficient aspect instead of speed optimization [47, 68]. Our analysis depends on the static properties of the SDF to derive sequences of actor executions that drain tokens on channels. Deriving such reverse sequences for more expressive stream models of computations such as Kahn process networks [58] is problematic due to unknown traffic patterns between processes. Although this work is within the SDF domain, the process migration technique to improve core utilization and reduce energy consumption is applicable

to other streaming languages as well [23, 60, 82].

3.8 Conclusion

We have made a case for exploiting high-level abstractions to design adaptive programs by presenting our StreaMorph technique for stream programs. We have shown that high-level abstractions can help design *adaptive programs*. The concept of adaptive programs proposed in this chapter is important in cloud computing, mobile computing, and cyber-physical systems when applications can be dynamically deployed and migrated on a wide variety of environments. Morphing programs can also help isolate performance of applications, thereby improving QoS of applications.

This work can be extended in several directions. Our next step would be predicting the number of cores necessary for a given IO rate. We plan to apply the adaptive program concept to applications with implicit parallelism with multiple algorithm choices, e.g., in PetarBricks [8], so that applications can execute adaptively under fair-multiple-resource constraints [43, 42].

Our evaluations use Intel Xeon processors, it would be more interesting if our evaluations are done using multimedia processors, however, at the time the chapter is written, we do not have a multicore multimedia processor platform at hand. We also have not explored the idea of using configuration switching to lower core utilization to protect processors from overheating.

Chapter 4

Exploiting Networks on-Chip Route Diversity for Energy-Efficient DVFS-Aware Routing of Streaming Traffic

Stream programs often demand high communication bandwidth. Networks on-Chip (NoC) is an interconnection paradigm to address the scalability issue of the conventional bus systems. In this chapter, we focus on a streaming traffic routing scheme for NoC to reduce energy consumption. Our routing technique combines the dynamic voltage frequency scaling (DVFS) capability and the route diversity property of NoC for energy reduction. Based on estimated communication demands of stream programs' flows, our routing algorithm finds energy-efficient DVFS-aware routes for the flows that minimize link and router frequencies to save energy. We evaluate our algorithm on a set of stream applications written in the StreamIt language. Our experimental results show that exploiting both the DVFS capability and the route diversity of NoC can significantly reduce links and routers' energy in comparison with a number of previous approaches, which utilize only either DVFS or route diversity. Our routing technique reduces router energy by 25% and 10% over the default XY routing and a state-of-the-art maximal throughput routing technique respectively when applying the same DVFS policy for the set of stream benchmarks. For links, our routing technique reduces energy by 26% and 29% over the two routing techniques respectively when applying DVFS. Our router frequency tuning technique results in 8.7X router energy reduction over several current policies that do not tune router frequencies. Applying DVFS to both links and routers comes at the cost of 8% increase in average latency. To the best of our knowledge, our work is the first to present a case for applying DVFS to routers to save energy.

4.1 Introduction

Networks on-chip is a scalable interconnection fabric for multicore machines that can address the limited scalability of the bus paradigm. Thanks to their high throughput, NoC are especially suitable for stream applications, which often demand high communication bandwidth [81]. The route diversity of NoC that makes the communication paradigm scalable can also become a source of energy-inefficiency. NoC energy consumption has been becoming increasingly substantial and important in multicore machines [102] along with the rising communication bandwidth demands, e.g., from high-quality stream applications. This chapter focuses on devising NoC energy reduction scheme while still meeting the communication bandwidth requirement of stream applications.

The advent of underlying hardware capabilities such as DVFS enables new energy saving schemes [106, 72, 102, 27, 122, 54, 83, 89]. The previous approaches often assume the default XY routing scheme in NoC before applying DVFS to reduce energy consumption. This routing assumption misses an important opportunity to balance network traffic appropriately to further reduce energy consumption. This limitation may come from the difficulty in predicting and estimating precisely the random traffic patterns and time-varied communication bandwidth of general applications. In contrast, stream applications often exhibit fixed traffic patterns with predictably periodic bandwidth [113, 47, 70]. As a result, exploiting those stationary properties of stream traffic can lead to better throughput-optimized routing schemes [1, 64, 84]. In the related work, routing is used as a mechanism to reduce communication bottlenecks at congested links. Differently, we focus on reducing NoC energy from both the DVFS and application-aware perspectives. Our routing scheme exploits the stationary properties of stream traffic to route stream flows to appropriate routes while tuning operating voltages and frequencies of both *links* and *routers* simultaneously on the routes to achieve better energy saving.

Tuning router frequencies for energy is difficult and has not been considered in [102, 106, 27, 72] due to the randomness of general applications' traffic. Tuning router frequencies for general traffic may lead to drastic performance degradation because routers are often more complicated than links and located at the intersections of several links and flows. However, the predictably stationary property of stream traffic makes router frequency tuning more feasible. Experimentally, we show that our router frequency tuning scheme results in substantial router energy reduction without drastic performance hits.

To realize the above points, we formulate the stream traffic DVFS-aware routing as a Mixed Integer Linear Program (MILP) problem using the Orion 2 NoC [59] power model of links and routers. As the routing problem is NP-hard, we present a heuristic DVFS-aware routing scheme to handle large networks with large numbers of traffic flows. When applications no longer execute within one single environment, e.g., in cloud computing, but rather are dynamically relocated between cores [31, 49], e.g., to reduce peak power consumption or to save energy by consolidating tasks and turning off unused cores, this heuristic routing scheme would be beneficial because it can address the scalability issue of the optimal MILP routing schemes [1, 64, 84]. While frequent traffic rerouting using MILP

can be expensive, our heuristic DVFS-aware routing scheme enables fast and inexpensive traffic rerouting.

Our experiments in 2-D mesh networks show that our routing scheme leads to 26% and 29% link energy reduction compared to the default XY routing approach employed in [106, 72, 102, 27, 106, 122] and the Transcom-Fission [1], a state-of-the-art maximal throughput routing technique, respectively while applying the same DVFS policy. Our router frequency tuning scheme, which, to the best of our knowledge, has not been explored for energy reduction, leads to 8.6X energy reduction over not tuning router frequencies. When applying our router frequency tuning scheme, our DVFS-aware routing scheme leads to 26% and 10% energy router reduction over the XY and Transcom-Fission routing respectively. The paper's contributions include:

- An energy-efficient DVFS-aware routing method for NoC by exploiting the DVFS capability and NoC route diversity to additionally reduce energy consumption when compared to the state-of-the art routing techniques.
- Given that, applications, e.g., in cloud computing, need to quickly adapt to external changes, e.g., resources [49], IO rates, it is necessary to be able to reroute traffic dynamically at runtime. Our heuristic DVFS-aware routing algorithm helps facilitate such runtime reconfiguration with significantly less routing time compared to MILP methods.
- We present a case and a technique to tune router frequencies leading to substantial router energy reduction for stream traffic.
- Through our experiments with a set of stream benchmarks written in the StreamIt language, we explain a traffic distinction between the software pipelining and hardware pipelining compilation techniques for stream programs. This compilation-oriented traffic characterization provides hints for future NoC-aware stream program compilation techniques.

4.2 Motivating Example

Specifically, for a link, we can estimate its utilization as follows: $U_{link} = \frac{b}{f_{link} w}$ where b is the total amount of traffic flowing through the link during one second and w is the width of the link in numbers of bits. Plugging into (1.1), we arrive at:

$$E = \beta \frac{1}{2} C V_{dd}^2 \frac{b}{w} \tag{4.1}$$

We will use equation (4.1) to analytically illustrate the motivating example.

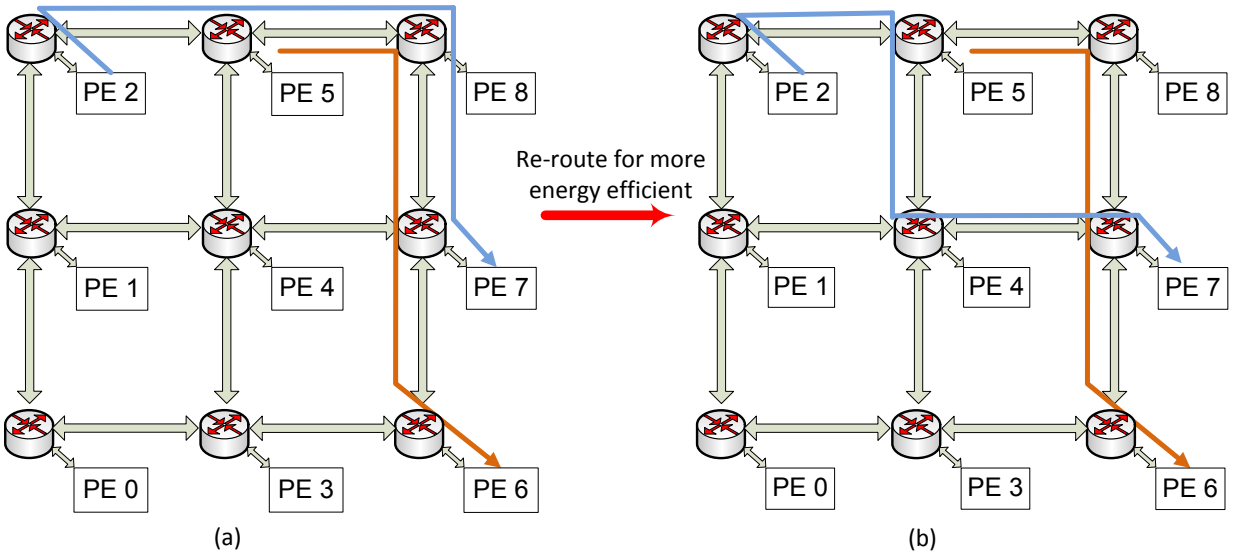


Figure 4.1: Energy aware routing.

DVFS-Aware Energy Efficient Routing

In a NoC, traffic is sent between nodes through a network of routers connected by links as in Figure 4.1. Exploiting the route diversity of NoC can help reduce not only traffic contention but also energy consumed in interconnection networks. For example, Figure 4.1(a) shows a small NoC of two flows $(2 \rightarrow 7)$ and $(5 \rightarrow 6)$ with bandwidth $b_{(2\rightarrow7)}$ and $b_{(5\rightarrow6)}$ respectively. If the conventional XY routing method [44, 32] is used, the two flows share links $(5 \rightarrow 8)$ and $(8 \rightarrow 7)$. Suppose that this setting requires these links to run at a frequency-voltage pair (f_3, V_3) to be able to carry the traffic of the two flows. Now, consider the case when flow $(2 \rightarrow 7)$ is routed through route $(2 \rightarrow 5 \rightarrow 4 \rightarrow 7)$ while the other flow remains in the same route as in Figure 4.1(b). In this setup, each link only carries the traffic of one flow. The required frequency-voltage pair for links $(5 \rightarrow 8)$ and $(8 \rightarrow 7)$ becomes (f_2, V_2), and the required frequency-voltage pair for links $(2 \rightarrow 5)$ and $(4 \rightarrow 7)$ becomes (f_1, V_1). In XY routing, links $(5 \rightarrow 8)$ and $(8 \rightarrow 7)$ have to carry the traffic of both of the flows, as a result, it is probable that $f_3 > f_2$ and $f_3 > f_1$. Consequently, by equation (1.2) we have $V_3 > V_2$ and $V_3 > V_1$. Plugging in equation (4.1), we have:

$$E_{XY} = 2\beta\frac{1}{2}C\left({V_3}^2\frac{b_{(2\rightarrow7)} + b_{(5\rightarrow6)}}{w}\right) > E_{DVFS-aware} = 2\beta\frac{1}{2}C\left({V_1}^2\frac{b_{(2\rightarrow7)}}{w} + {V_2}^2\frac{b_{(5\rightarrow6)}}{w}\right) \quad (4.2)$$

Equation (4.2) suggests that balancing flows in the NoC can lead to link energy saving. Moreover, because the routing setting in Figure 4.1(b) allows links $(5 \rightarrow 8)$ and $(8 \rightarrow 7)$ to

run at a lower frequency, as a result, routers 5, 7, and 8 would be able to operate at a lower frequency while still meeting the traffic demand of flows $(5 \rightarrow 6)$ and $(2 \rightarrow 7)$. Therefore, in addition to link energy reduction, the routing setting in Figure 4.1(b) can also help reduce the energy of routers 5, 7 and 8.

The above DVFS-aware routing example suggests that balancing network traffic can lead to energy saving. However, there are cases in which balancing network traffic naively can lead to energy inefficiency. We illustrate the observation by a more complicated example. Let us assume that $b_{(2\rightarrow7)} < b_{(5\rightarrow6)}$. As a result, $f_1 < f_2$ and $V_1 < V_2$. Now, another new flow $(5 \rightarrow 7)$ arrives with bandwidth demand $b_{(5\rightarrow7)}$. A routing mechanism that seeks to balance network traffic would route the new flow through route $(5 \rightarrow 4 \rightarrow 7)$ because these links are less congested. However, routing the new flow through the route may require links $(5 \rightarrow 4)$ and $(4 \rightarrow 7)$ to raise their frequency from f_1 to f_2 in order to meet the total demand of the two flows. Let us consider the situation when routing the new flow through route $(5 \rightarrow 8 \rightarrow 7)$ does not make the links on the route raise their current f_2 operating frequency to meet the total demand. Consequently, the total energy consumed by flow $(2 \rightarrow 7)$ on links $(5 \rightarrow 4)$ and $(5 \rightarrow 7)$ rises from $2\beta\frac{1}{2}CV_1{}^2\frac{b_{(2\rightarrow7)}}{w}$ to $2\beta\frac{1}{2}CV_2{}^2\frac{b_{(2\rightarrow7)}}{w}$. To avoid such a situation, a DVFS-aware routing scheme would choose route $(5 \rightarrow 8 \rightarrow 7)$ for the new flow as it saves more energy for the traffic of flow $(2 \rightarrow 7)$ on links $(5 \rightarrow 4)$ and $(4 \rightarrow 7)$. Therefore, the network balancing routing scheme may not be as energy-optimal as the latter DVFS-aware routing scheme. This is the major difference between a DVFS-aware routing scheme and throughput-optimized routing schemes that seek to minimize the congested links' traffic load [64, 1] to reduce communication bottlenecks. While routing to reduce communication bottlenecks on congested links can improve throughput, resulting in energy reduction by lowering the frequency of an entire network, it often does not pay attention to reducing energy consumption for non-bottleneck parts. As a consequence, throughput-optimized routing may not be as energy-optimal as our DVFS-aware routing scheme.

4.3 Background

Streaming

As the SDF semantics can model most of stream programs [113], stream applications often bear the static properties of SDF such as the amount of traffic sent between actors within one iteration is fixed and bounded. As a result, after mapping actors of a stream program to a multicore chip, the communication pattern between nodes in the chip within one iteration [1] is known and fixed. Abdel-Gawad and Thottethodi, in their Transcom [1] work, exploit this property to improve the communication throughput of stream programs.

Differently from Transcom [1] that uses the traffic of programs compiled with the hardware pipelining technique, we focus on the traffic of stream programs compiled using the software pipelining technique [47, 68] as it has better throughput. The major advantage

[1] For the formal definition of iteration, readers can refer to [70].

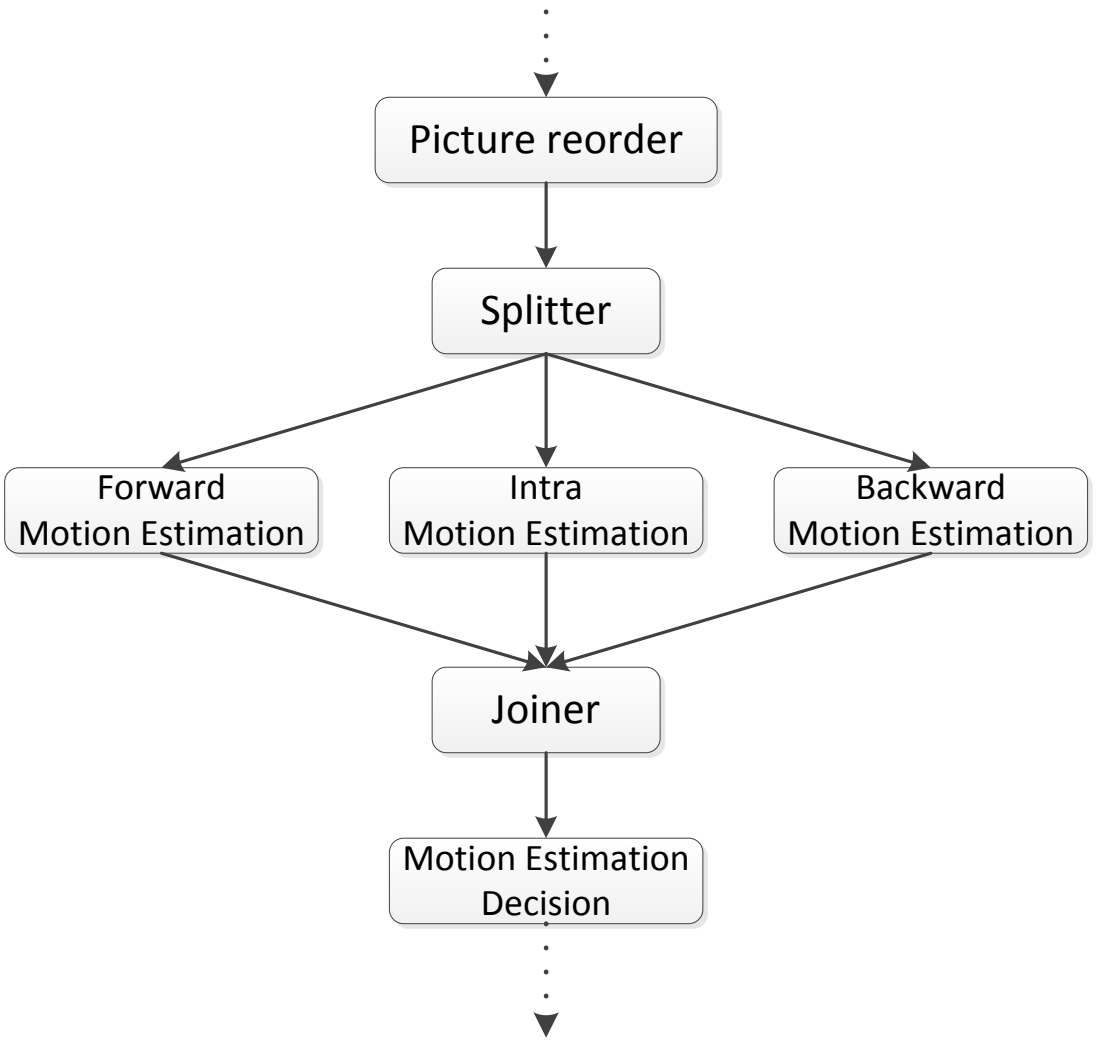
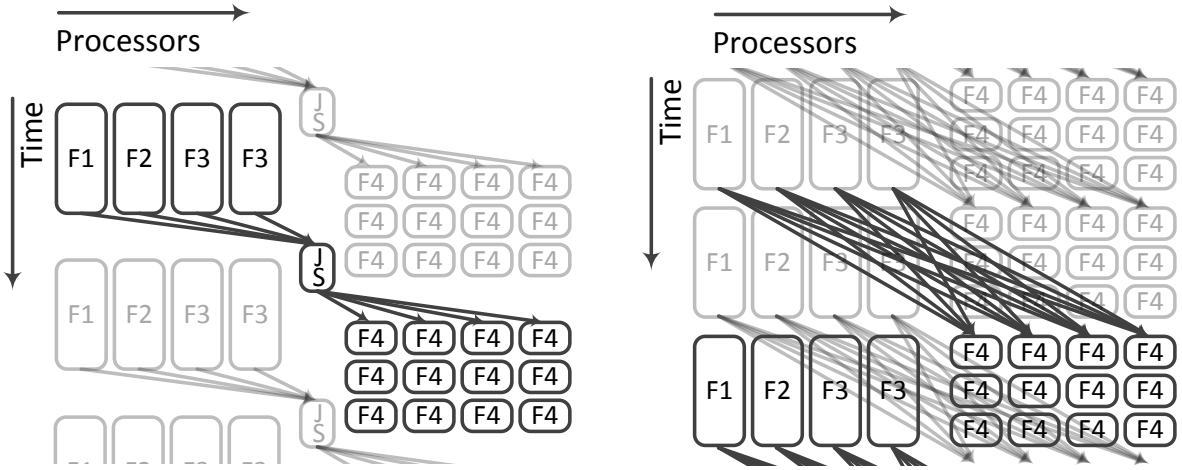


Figure 4.2: A portion of the MPEG2 decoder stream graph in StreamIt.

of software pipelining over hardware pipelining is that it allows mapping any actors to any core, while in hardware pipelining, non-contiguous actors should not be mapped to one core [47]. For example, in Figure 4.2, mapping actors`Picture Reorder` and `Motion Estimation Decision` to one core and actors `Forward Motion Estimation`, `Intra Motion Estimation`, `Backward Motion Estimation` to another core is not recommended in hardware pipelining because it may result in inter-actor synchronization difficulty and inefficiency. As a consequence, hardware pipelining may lead to unbalanced mapping of stream programs to multicore. In contrast, this mapping is normal in software pipelining. There is also a distinction between the traffic patterns of software pipelining and hardware pipelining that makes the Transcom traffic fission (splitting) technique not very effective for software pipelin-

ing traffic. We defer elaborating on this observation to the evaluation Section 4.7 because experimental results can help explain the phenomenon better.



(a) Hardware Pipelining (b) Software Pipelining

Figure 4.3: Pipelining compilation techniques for stream programs.

In software pipelining, actors do not communicate directly within one iteration. Rather, they produce results into buffers that can be consumed by downstream actors in some next iteration. Figure 4.3(b) illustrates the execution mechanism of a stream program compiled with the software pipelining technique adapted from Gordon et al. [47]. In the figure, the communication between actors `F1`, `F2`, `F3`, and actor `F4` is overlapped with the computation within one iteration. This implies that the inter-core traffic is not very delay-sensitive meaning that outputs of actors do not need to be sent immediately with stringent delay bounds. Rather, actors' outputs can be delayed, orchestrated and transmitted at appropriate times as long as they are delivered by the start of the designated iteration so that downstream actors can start using the outputs. Kudlur and Mahlke [68] implement this computation-communication overlapping technique for stream programs for the Cell processor. In addition, due to the static and analyzable properties of stream programs [113], the amount of inter-core traffic carried by interconnection networks is often static and predictable across iterations of a stream program.

We use the StreamIt compiler [47] to map and derive the communication patterns and the traffic load for each stream program in 2-D mesh multicore chips similar to the RAW processor within one iteration. The constraint for our routing procedure is that, within one iteration period determined by computation time, NoC must deliver all inter-core traffic load. As the computation interval of one iteration can vary due to different processor speed and architecture, estimating this interval using a specific processor model is often not adequate. Instead, we evaluate the effectiveness of our routing mechanism with respect to different interval lengths.

Hardware Support for DVFS and Routing

Deploying DVFS-aware routes in NoC requires an additional feature in routers to guide packets through designated routes [73, 27, 64]. We assume a similar router architecture as presented in [64] that includes a programmable routing table at each router. Routing tables only require initializing once when an application starts running, thus the energy for initiating routing table is negligible. We also assume that links [102, 27, 106] and routers [83, 89] can operate at multiple frequencies.

The rationale of our router DVFS capability assumption derives from the works by Mishra et al. [83] and Ogras et al. [89]. Mishra et al. show that a hardware DVFS NoC architecture is practical and use that DVFS capability to boost and throttle router operations to mitigate NoC congestion. The architecture employs an asynchronous communication mechanism between routers with multiple on-chip voltage regulators. Meanwhile, Ogras et al. [89] present a design methodology for NoC with voltage-frequency island partitions.

4.4 Energy-Optimal Routing

In this section, we present a method for finding energy efficient routes that is aware of DVFS settings. We first give the formal model for the routing problem.

Definition 5 *An NoC is formulated as a flow graph $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices (routers) and $\mathcal{E}$ is the set of edges (links). Routers and links can operate at m operating points $O = \{(f_1, V_1), \ldots, (f_m, V_m)\}$. Each edge $e \in \mathcal{E}$ has a width of w bits. One stream program creates a set of flows $\mathcal{W} = \{W_1, \ldots, W_{|\mathcal{W}|}\}$. For each flow $W_i = (s_i, d_i, b_i)$, s_i and d_i are the source and the destination of the flow respectively, b_i is the average load of the flow. $t_{i,e}$ is a boolean variable denoting if flow i goes through edge e. $o_{j,e}$ denotes that edge e uses the j^{th} operating point of O. f_e and f_v denote the frequencies of edge (link) e and vertex (router) v respectively.*

Now, we formulate the DVFS-aware routing problem as a MILP problem that captured the essence of the routing scheme.

Energy-Optimal Integer Linear Programming Routing

While reducing link operating frequencies, we still need to meet stream programs' required throughput. Equation (4.3) indicates that the frequency of each edge must be greater than a certain value to sustain the total traffic demand of the flows going through the edge. Equations (4.4) and (4.5) enforce that the frequency of an edge or a vertex is only selected from one of the operating point options O. Equation (4.6) denotes that, at each vertex, a flow is not splittable. Equation (4.8) indicates that if a flow enters a vertex, which is not the flow's source or destination, the flow must exit in some direction. Equation (4.9) tells that

a flow must depart its source and arrive at its destination. Equation (4.10) forbids a flow from going back to the direction that it comes in.

Equation (4.11) is more complicated than it looks and requires a more detailed explanation. As routers are more complicated than links, routers often do not achieve 100% throughput due to the non-maximal virtual channel (VC) and switch arbitration processes [32]. As a consequence, routers running at the same minimal frequencies of links can cause performance degradation as shown in Figure 4.9. To mitigate this problem, we use equation (4.11) to constrain that the frequency of a router must not be smaller than the frequency of any of its connected links. The reason behind this constraint is that, if the frequency of the router is smaller than the frequency of one of its connected links, because the frequency of the link is already minimal, the router would not be able to sustain the traffic load through the link. In fact, the frequency of a router should be higher than the maximum optimal frequency of its connected links to avoid performance degradation. Therefore, we enforce that router frequencies have to be multiplied by a factor $X \leq 1$ as in equation (4.11) to avoid significant performance degradation.

Our objective is to optimize the total network energy consumption. Equation (4.15) computes the energy consumed by links based on the Orion 2 model [59]. Note that, equation (4.15) is shorten from equations (4.13) and (4.14). We interpolate the function:

$$\texttt{routerPower}((f_v, V_v), input\_bandwidth) = a(f_v, V_v) \times input\_bandwidth + c(f_v, V_v)$$

where $a(f_v, V_v)$ and $c(f_v, V_v)$ are fixed at a fixed operating point(f_v, V_v). We use the energy/throughput pairs returned by the Orion 2 power model [59] at each operating point to estimate these constants[2]. While using a linear function to estimate router energy based on input rates does not reflect the exact energy behavior of routers, it can capture precisely the read/write operation energy of buffers and crossbars. Virtual channel and switch allocation arbitration energy is difficult to estimate solely from input rates. However, the arbitration energy often only makes up for 7% of NoC energy consumption [59] so the estimation error would be supposedly small. Readers can refer to [59] for more details.

Capacity constraint:

$$\forall e \in \mathcal{E}, \quad \sum_{i \in \mathcal{W}} b_i t_{i,e} \leq f_e w \tag{4.3}$$

Frequency level selection:

$$\forall u \in \mathcal{E} \bigcup \mathcal{V}, \quad f_u = \sum_{j=1}^{m} o_{j,u} f_j \tag{4.4}$$

$$\forall u \in \mathcal{E} \bigcup \mathcal{V}, \quad \sum_{j=1}^{m} o_{j,u} = 1 \tag{4.5}$$

[2]The router power estimation function in Orion 2 actually returns energy estimation outputs as a linear function of input rates at a fixed operating point.

Unsplittable flow:

$$\forall i \in \mathcal{W}, \forall u \in \mathcal{V} \sum_{(u,v)\in\mathcal{E}} t_{i,(u,v)} \leq 1 \tag{4.6}$$

Bounded route length:

$$\forall i \in \mathcal{W}, \sum_{e\in\mathcal{E}} t_{i,e} \leq hop_i \tag{4.7}$$

Flow conservation:

$$\forall i \in \mathcal{W}, \forall u \neq s_i, d_i \in \mathcal{V}, \sum_{(w,u)\in\mathcal{E}} t_{i,(w,u)} = \sum_{(u,v)\in\mathcal{E}} t_{i,(u,v)} \tag{4.8}$$

Flow origination and termination:

$$\forall i \in \mathcal{W}, \sum_{(s_i,v)\in\mathcal{E}} t_{i,(s_i,v)} = 1, \sum_{(u,d_i)\in\mathcal{E}} t_{i,(u,d_i)} = 1 \tag{4.9}$$

180-turn restriction:

$$\forall i \in \mathcal{W}, \forall (u,v),(v,u) \in \mathcal{E}, t_{i,(u,v)} + t_{i,(v,u)} \leq 1 \tag{4.10}$$

Router frequency:

$$\forall u \in \mathcal{V} \begin{cases} X f_u \geq \min(\max_{(u,v)\in\mathcal{E}} \sum_{i\in\mathcal{W}} b_i t_{i,(u,v)}, X \max_{j=1}^{m} f_j) \\ X f_u \geq \min(\max_{(w,u)\in\mathcal{E}} \sum_{i\in\mathcal{W}} b_i t_{i,(w,u)}, X \max_{j=1}^{m} f_j) \end{cases} \tag{4.11}$$

Objective function:

$$\min \sum_{e\in\mathcal{E}} E_e + \sum_{v\in\mathcal{V}} E_v \tag{4.12}$$

where:

$$E_e = \sum_{j=1}^{m} \beta f_e {V_{dd_j}}^2 Co_{j,e} U_e \tag{4.13}$$

$$U_e = \frac{\sum_{i\in\mathcal{W}} b_i t_{i,e}}{f_e w} \tag{4.14}$$

$$E_e = \frac{\left(\sum_{j=1}^{m} \beta {V_{dd_j}}^2 Co_{j,e}\right)\left(\sum_{i\in\mathcal{W}} b_i t_{i,e}\right)}{w} \tag{4.15}$$

$$E_v = \texttt{routerPower}\left((f_v, V_v), \sum_{(u,v)\in\mathcal{E}} \sum_{i\in\mathcal{W}} b_i t_{i,(u,v)}\right) \tag{4.16}$$

Heuristic DVFS-Aware Routing

The above optimal unsplittable flow routing is NP-hard [65]. As a consequence, it does not scale well for large networks and stream graphs. However, the formulated MILP problem captures the constraints for our DVFS-aware routing scheme. In this section, we present a scalable heuristic routing method based on Dijkstra's algorithm that gradually routes flows through a network. The method finds a route that minimizes the energy consumption increment of the network. Algorithm 4.4 shows the distance function for Dijkstra's algorithm.

The idea behind the distance function is as follows. Whenever a new flow is considered to go through an edge (link), its traffic would be added on top of the edge's current traffic. As a result, the prospective traffic on the edge can make the current frequency of the edge no longer sufficient. If the prospective traffic requires a higher frequency level, we will not only need to compute the additional energy for the new flow on the edge, but also the additional energy for transporting the existing traffic on the edge if a higher frequency is required. If the edge needs to use a higher frequency, as routers should not run at a frequency lower than the frequency of any of its connected edges, the routers connected by the edge may need to use a higher frequency. As a consequence, routing the new flow through the edge may incur an additional energy increase due to transporting the routers' existing traffic at a higher frequency.

By plugging the distance function into Dijkstra's algorithm, we can find the route with the smallest amount of energy increase whenever routing a flow. We also make a small modification to conventional Dijkstra's algorithm. Note that conventional Dijkstra's algorithm terminates after routing a flow, however, in our case, we need to route flows successively through a network. As a result, we should balance the network's traffic whenever possible. Therefore, whenever there exist two directions (edges) with the same energy increase, it would be beneficial to choose the edge with lower total traffic.

In lines 35, 38, 41, and 44, function `energy` computes the estimated power consumption of a router given the amount of traffic that flows through it at a frequency-voltage pair (an operating point). This function is the same as function `routerPower` in the previous section.

After all the flows are routed through a network, we select the minimal frequency for each edge that satisfies the capacity constraint of the edge.

Traffic Splitting to Improve Energy Saving

It has been shown that splitting flow traffic to route through multiple different routes can improve throughput [84, 1]. However, multipath routing schemes often increase network traffic and route lengths as in the Transcom work [1]. In addition, the multipath routing techniques' primary purpose is to reduce the maximum traffic load on congested links to resolve communication bottlenecks without paying attention to non-bottleneck parts. As a consequence, although multipath routing can improve throughput, it may not be energy-optimal.

```
1  distance(u , v, flow_demand, G)
2  e←G.getEdge(u,v)
3  link_energy_increase←linkEnergy(e.currentTraffic() + flow_demand, e) - linkEnergy(e.currentTraffic(), e)
4  router_energy_increase←routerEnergy(flow_demand, e, G) - routerEnergy(0, e, G)
5  return link_energy_increase + router_energy_increase
6                          ▷ ************************************************************
7  linkEnergy(demand, edge)
8  for f in freq_options.minToMax() do
9  |   if demand ≤ f * w then
10 |   |   return powerConsume(f, demand, edge)
11 |   end
12 end
13 return ∞
14                         ▷ ************************************************************
15 routerEnergy(added_traffic, edge, G)
16 old_traffic←edge.currentTraffic()
17 new_traffic←added_traffic + old_traffic
18 edge.setTraffic(new_traffic)                    ▷ temporarily set the new traffic to compute energy consumption
19 new_freq← ∞
20 new_vol← ∞
21 for (f,V) in operating_points.minToMax() do
22 |   if new_traffic ≤ Xfw then
23 |   |   new_freq←f
24 |   |   new_vol←V
25 |   |   break
26 |   end
27 end
28 if new_freq > operating_points.maxFreq() then
29 |   (new_freq, new_vol)←operating_points.maxOperatingPoint()
30 end
31             ▷ compute estimated energy for the routers connected to the edge with the new frequency
32 u←edge.getSource()
33 v←edge.getDestination()
34 if new_freq > u.currentFreq() then
35 |   energy←u.energy((new_freq, new_vol), G)                    ▷ the router needs a higher frequency
36 end
37 else
38 |   energy←u.energy(u.currentFreqVol(), G)                     ▷ the current frequency is good enough
39 end
40 if new_freq > v.currentFreq() then
41 |   energy←energy+v.energy((new_freq, new_vol), G)              ▷ the router needs a higher frequency
42 end
43 else
44 |   energy←energyv.energy(v.currentFreqVol(), G)               ▷ the current frequency is good enough
45 end
46 edge.setTraffic(old_traffic)                                   ▷ restore the previous traffic
47 return energy
```

Figure 4.4: Computing cost function.

Our DVFS-aware routing method aims at tuning the workload on each link given different operating voltage-frequency levels to reduce energy consumption. However, when traffic demands of flows are too *coarse*, it may not utilize the remaining *slacks* between the frequency levels of links well. Let us come back to the example in Section 4.2. Suppose that a portion of flow $(5 \rightarrow 6)$'s traffic can be sent through route $(5 \rightarrow 4 \rightarrow 7)$ without making the links on the route higher their frequency from f_1 to f_2. The remaining of flow $(5 \rightarrow 6)$'s traffic is still sent through route $(5 \rightarrow 8 \rightarrow 7)$. The fraction of flow $(5 \rightarrow 6)$'s traffic sent through $(5 \rightarrow 4 \rightarrow 7)$ is now carried by links at frequency f_1, which is lower than f_2 of the links on route $(5 \rightarrow 8 \rightarrow 7)$. As a result, we would save more energy for the fraction of flow $(5 \rightarrow 6)$'s traffic sent through $(5 \rightarrow 4 \rightarrow 7)$.

The above observation suggests that it would be beneficial to split traffic of flows into smaller trunks to better exploit the slacks between different frequency levels. We augment our routing scheme in the previous section with an additional preprocessing step. We split each flow into a number of sibling flows such that the number is at most the number of minimal routes between the source and the destination of the original flow. We enforce another constraint which requires the amount of traffic for each sibling flow must be greater than a certain amount to avoid increasing conflicts when there are too many flows. After splitting the flows, we apply the routing algorithm in Section 4.4 to the split flows. When scheduling the traffic of sibling flows, we avoid sending packets of sibling flows simultaneously, rather, we send sporadically as if sending packets of their parent flow, but through different routes.

Deadlock-Free Routing

We employ a similar VC assignment technique [104, 1] to resolve deadlocks in case there exist loops. Whenever route segments form a loop, we assign VCs to flows to avoid the deadlock by the loop. For further details, readers can refer to the papers.

4.5 Experiment Setup

Benchmarks and Compilation

We use a set of stream benchmarks written in the StreamIt language as in [47]. StreamIt compiler framework [46] compiles the stream programs in a fashion that overlaps communication with computation [117, 68, 47]. We first use the StreamIt compiler to compile and map the benchmarks to 2-D mesh 4×4 and 8×8 chips similar to the RAW processor [47]. After mapping, we use the compiler to derive the inter-core communication patterns and the traffic loads for flows. For different network sizes, the StreamIt compiler generates different stream graphs even for one benchmark.

Simulator

We use a C++ simulator modeled after the GARNET simulator [2] integrated with the Orion 2 power model [59] to extract the energy consumption of routers and links. Each router has five input ports and five output ports including one input port and one output port connected to the links to the local processing elements. Each port has 4 VCs with buffers of 12 flits deep. Links and flits are both 64 bits wide. An entire router runs at the same frequency, while different routers can operate at different frequencies. We assume the 65nm technology for the routers, similar to [83] [3]. We also assume that the energy overhead for the table routing logic is negligible [64]. Each router, each link can operate at one of the 10 frequency-voltage pairs as in Table 4.1 from [102].

Frequency (MHz)	125	240	300	370	450	540	640	750	870	1000
Voltage (vol)	0.90	1.10	1.19	1.3	1.43	1.58	1.75	1.95	2.20	2.50

Table 4.1: Frequency-voltage pairs

4.6 Results

As the previous work on exploiting DVFS to reduce NoC power consumption [102, 106, 83] assumes the default XY deterministic routing, we compare our energy-aware routing scheme against the XY routing when applying the same DVFS scheme. We also compare our routing scheme against the Transcom [1] traffic *fission* (splitting) technique (including the free-routing technique), a state-of-the art maximal throughput routing technique.

Link and Router Energy Reduction over Other Routing Techniques

Because routers and links are scheduled to operate at fixed frequencies once a program is initialized, we can ignore the transition energy. Figures 4.5 and 4.6 show the results of different routing techniques in 4×4 and 8×8 mesh networks at different traffic load levels.

As discussed in Section 4.3, we assume that the NoC has to deliver predictably known amounts of traffic between cores within one iteration interval of a stream program. The iteration interval of a stream program can depend on the computation time of processors, on different speed, and/or varied IO rates of streams, such as video framerates, sampling rates, and so on. Therefore, using profiled architecture-specific packet timing traces may not be adequate. Instead, we vary the interval length of one iteration to the point that our routing technique can guarantee the throughput. This operating point is denoted as 100%

[3] We chose this 65nm technology for the compatability reason with [83]. In fact, we can use more modern technolgy such as 32nm or 48nm.

traffic load. We then reduce the maximum traffic load by a half, denoting as 50% traffic load. We apply the same DVFS scheme to all the routing schemes to minimize the voltages and frequencies of links and routers without hurting the performance too much. Readers may say that this evaluation scheme is not fair for the Transcom-Fission technique because it may be better at higher traffic load. We argue that the effectiveness of the Transcom-Fission technique is not limited at high bandwidth traffic load. Rather, the Transcom-Fission technique seeks to minimize the maximum link load at *any* traffic load level, thereby reducing energy consumption by lowering operating frequencies. To be fair, we use the energy delay product (EDP) metric to compare between the routing schemes. We normalize the results to the results of XY routing.

The results in Figures 4.5 and 4.6 show that our DVFS-aware routing technique performs better than the XY and the Transcom-Fission routing techniques, resulting in 25% and 10% router mean energy reduction over the XY and Transcom-Fission schemes respectively. For link energy, our DVFS-aware routing scheme improves by 26% and 29% respectively over the XY and Transcom-Fission schemes. The XY routing scheme improves performance at lower traffic load. The application-aware schemes, such as Transcom-Fission and DVFS-aware, improve performance in larger networks, perhaps because the route diversity degree increases along with network sizes.

Surprisingly, the Transcom-Fission routing technique performs worst in the 4×4 network while it becomes better than XY routing in the larger 8×8 network. This implies that the traffic-fission scheme, which minimizes the maximal link load is not very effective for our software pipelining traffic. Note that, for the 8×8 network, there are some missing performance values of the Transcom-Fission scheme because our implementation of the Transcom-Fission scheme does not find a valid solution after one hour for those cases, while the Transcom [1] paper mentions that their procedure can find a valid solution after 10 minutes. Our DVFS-aware routing technique performs best in most of the cases. The Traffic-Splitting technique in Section 47 does not really improve the results of the DVFS-aware routing, while in theory, it should. We believe that there is some fundamental difference between the hardware pipelining traffic and the software pipelining traffic that makes the Transcom-Fission scheme perform not very well with the software pipelining traffic, while it performs very well with the hardware pipelining traffic [1], and the Traffic-Splitting scheme is not effective. We use Section 4.7 to explain the distinction between those traffic types that may help explain the surprising results.

DVFS Performance Penalty

Applying DVFS to both links and routers often leads to performance degradation, especially for routers. Figure 4.7 shows the performance degradation when applying DVFS. We use the amount of time to deliver the traffic load of one iteration, corresponding to the interval length of one iteration, to estimate the performance penalty when using DVFS. We compare the performance of DVFS against the non-DVFS performance. In the figure, 100% 4×4 denotes 100% (maximum) traffic load in a 4×4 network. Other notations are similar. Sev-

eral benchmarks such as `FFT, MPEG2, TDE, Bitonic-Sort` suffer almost no performance penalty when applying DVFS to both links and routers. While the `ChannelVocder` benchmark performance suffers most seriously from DVFS for the large 8×8 network. This result perhaps comes from heavy traffic conflict at shared VCs when applying the deadlock-free routing [1]. A similar phenomenon is mentioned in Transcom [1]. Overall, applying DVFS to both routers and links causes 8% of latency degradation. In return, applying DVFS leads to 8.7x routers and 4.4x links energy reduction over not using DVFS as shown in Figure 4.8.

Note that, when applying DVFS to routers, we enforce that the frequency of a router has to be scaled from the maximum of the minimum estimated frequency of its connected links as in equation (4.11) to avoid performance degradation. Figure 4.9 shows the trade-off between performance and energy when scaling router frequencies. The results are normalized to the performance at the smallest factor of 0.65. The figures suggest that a scaling factor smaller than 0.8 does not improve the performance while it causes 15% energy lost. A scaling factor greater than 0.8 degrades performance significantly while energy reduction is not substantial. We chose a scaling factor of 0.8 for our evaluations[4].

Heuristic DVFS-Aware Routing Time

To evaluate the feasibility of applying our heuristic DVFS-aware routing scheme to runtime program reconfiguration, e.g., as in [24, 49, 31], we measure the running time of our heuristic routing scheme on an eight-core Intel Xeon machine for an 8×8 NoC. Figure 4.10 shows the results of our evaluation. Although the routing routine is written in Python, none of the benchmarks has routing time exceeding 1 seconds. The mean routing times are less than half a second. We expect much faster routing times if the routing routine is written in C. In contrast, the MILP routing scheme often does not terminate after one hour for most of the benchmarks. These results demonstrate that our DVFS-aware routing scheme would be suitable for runtime traffic rerouting of runtime program reconfiguration [24, 49, 31].

4.7 Traffic Distinction between Software and Hardware Pipelining

The Transcom-Fission technique [1] proves to be very effective for the stream traffic compiled by the hardware pipelining technique by fissing (splitting) traffic of flows into smaller ones. From our insight, this is because the hardware pipelining traffic tends to lump up more than the software pipelining stream traffic. We will use an example to elaborate on this observation. Figure 4.3 illustrates the mechanisms of hardware and software pipelining techniques used in the StreamIt compiler [47]. In hardware pipelining, actors residing at different processors proceed at different rates. Actors interact with one another by blocking

[4]Scaling factor may be a misnomer. Note that we use the constraint $X \times f_{router} \geq f_{connected-links}$ to enforce this requirement. As a result the scaling factor X is actually smaller or equal to 1.

on FIFO communication. While in software pipelining in Figure 4.3(b), actors on different processors proceed at the same rate and synchronized by barriers after each iteration. There is also an inter-actor traffic pattern distinction between software and hardware pipelining used in the StreamIt compiler. In the hardware pipelining example in Figure 4.3(a), actors `F1`, `F2` and `F3` send their outputs to a split-join `JS`. This split-join merges the results and forwards to multiple parallel instances of actor `F4`. As a consequence, the links to/from the node containing the split-join `JS` would be more congested than the other links and can become serious communication *bottlenecks* if traffic is not distributed appropriately. While in software pipelining, actors `F1`, `F2` and `F3` split and send their outputs *directly* to `F4` actors without going through a split-join traffic-centralization node. As a result, software pipelining would naturally balance network traffic better. In addition, note that, actors `F1`, `F2` and `F3` use totally $4 \times 4 = 16$ communication flows to send outputs to `F4` actors in software pipelining, therefore, each flow would carry $\frac{1}{16}$ of the total traffic. While in hardware pipelining, each of the four flows to/from the split-join `JS` would carry $\frac{1}{4}$ of the total traffic. Consequently, the bandwidth of each of the flows in software pipelining would be equal to $\frac{1}{4}$ of the bandwidth of each of the flows in hardware pipelining. In Transcom, the authors observe that splitting one hardware pipelining traffic flow into more than 4 flows does not improve throughput. Therefore, splitting software pipelining traffic flows is not very effective because the software pipelining technique already splits inter-core traffic well enough. In our experiments, the effectiveness of the Transcom-Fission technique is limited. For each benchmark, only one or two flows are split into two child flows. As a result, the Transcom-Fission technique does not improve throughput over our DVFS-aware technique by a large margin for software pipelining traffic because the only advantage of Transcom-Fission over our DVFS-aware scheme comes from splitting flows.

In addition, as the software pipelining technique does not require to map contiguous actors to a core, software pipelining traffic often contains more inter-core flows than hardware pipelining traffic. For example, in Figure 4.2, hardware pipelining should map actor `Picture Reorder` to a first core and actor `Motion Estimation Decision` to a second core while mapping `Forward Motion Estimation`, `Intra Motion Estimation`, `Backward Motion Estimation` as well as `Splitter`, and `Joiner` to another core resulting in one inter-core flow. On the other hand, software pipelining may map actors `Picture Reorder` and `Motion Estimation Decision` to a core and actors `Forward Motion Estimation`, `Intra Motion Estimation`, `Backward Motion Estimation` to another core, which can balance workload better yet resulting in 6 inter-core flows. Because the unsplittable flow routing problem is NP-hard [65], Transcom-Fission MILP formulation for software pipelining traffic often does not scale as well as for hardware pipelining traffic. Note that, although software pipelining splits and balances traffic better, it often results in more traffic load than hardware pipelining.

4.8 Related Work

The work in [102, 106, 72, 83, 20] exploits the DVFS capability to reduce power consumption without exploiting the NoC route diversity to find energy-optimal routes. Our work combines and exploits both the DVFS capability and the NoC route diversity for link energy reduction. In addition, we also tune router energy consumption when deriving DVFS-aware routes, which results in substantial router energy reduction. The work on mapping applications with static communication bandwidth to NoC [64, 104, 1, 84] concentrates on reducing communication bottlenecks at congested links to maximize throughput. While these approaches can lead to energy reduction by lowering the operating voltage and frequency of an entire network, they neglect to tune operating voltages and frequencies of non-congested parts in the network. Our approach looks for energy reduction opportunities by tuning the frequencies and voltages at all the parts of a network. Shang el al. present Powerherd [103], a dynamic routing technique to manage peak power constraints in interconnection networks by avoiding routing packets to hotspot routers. Our work is different from Powerherd because we combine routing with DVFS as a mechanism to reduce power consumption. Li et al. [73] present an energy aware routing technique that aggregates traffic on networks on-chip to reduce leakage energy by turning off unused links. This is an interesting direction to explore in future work. The work in [53, 83] shows that it is feasible to design NoC in which each router and each link can adjust operating voltages and frequencies. In [83], Mishra et al. propose to scale up routers' frequencies to mitigate temporal network congestion, while our work presents a technique to reduce routers' frequencies to save energy. Orgras et al. [89] propose a design methodology for voltage frequency island partitions.

Our work exploits the predictably stationary properties of stream programs derived from the SDF stream programming model by Lee and Messerschmitt [70] and the recent implementation of the programming model in the StreamIt language and compiler [113, 47]. The static properties of stream programs enable offline routing and frequency tuning analyses. We also exploit the advantage of the recent software pipelining compilation technique for stream programs [47, 68] to delay and schedule traffic sent through networks. This work is also related to the ActiveMessages [117] programming model that overlaps between computation and communication.

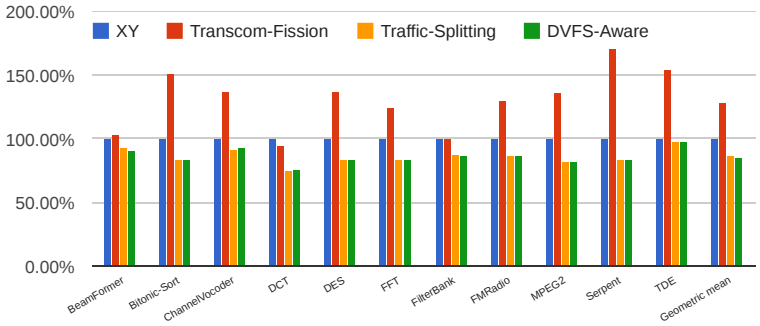
4.9 Conclusion

We have presented an energy-efficient DVFS-aware routing scheme of streaming program traffic written in the StreamIt language on NoC. The scheme exploits both the NoC route diversity property and the DVFS architecture capability. We estimate frequencies of both links and routers simultaneously to save energy. To the best of our knowledge, our work is the first to combine routing with DVFS. Our technique results in significant energy reduction in comparison with applying DVFS to the default routing scheme and a state-of-the-art throughput optimal routing scheme.

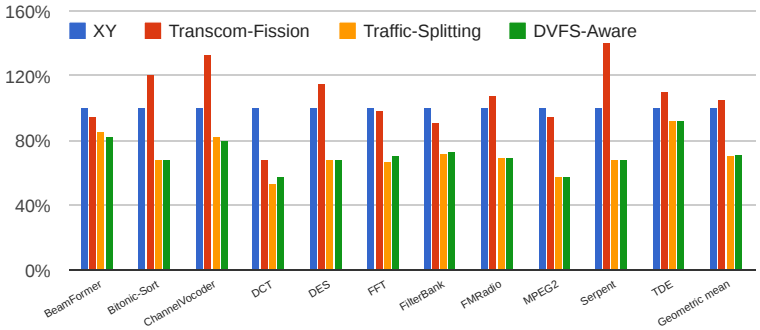
We also present a case for tuning router frequencies leading to substantial energy reduction. Our experiments demonstrate that it is possible to estimate the minimal router frequencies based on the traffic characteristics of an important class of applications. To the best of our knowledge, our work is the first to explore the possibility of reducing router frequencies to save energy.

Our heuristics DVFS-aware routing algorithm reduces routing time significantly, thus it is suitable for runtime reconfiguration of applications on multicore. Application runtime reconfiguration would be important in cloud computing and cyber-physical systems.

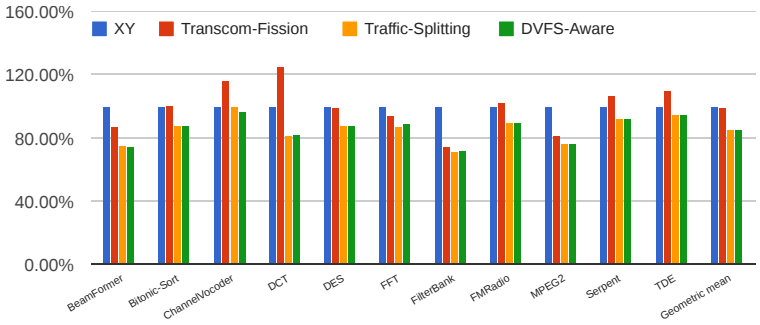
In addition, our work helps uncover a distinction between the traffic characteristics of software pipelining and hardware pipelining. It also suggests that software pipelining traffic balances networks better, thus, it can be one reason leading to the performance edge of software pipelining over hardware pipelining.



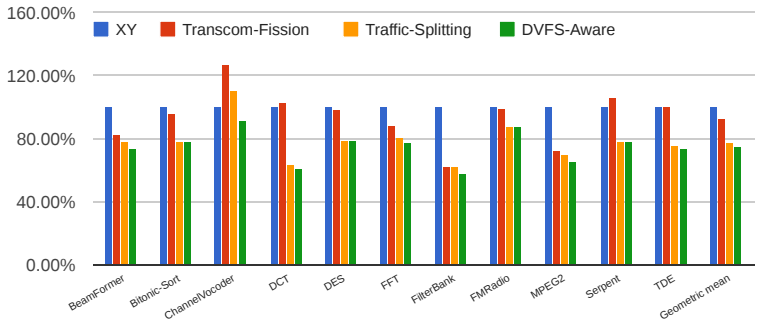
(a) Link energy improvement at 50% traffic load.



(b) Link energy improvement at 100% traffic load.

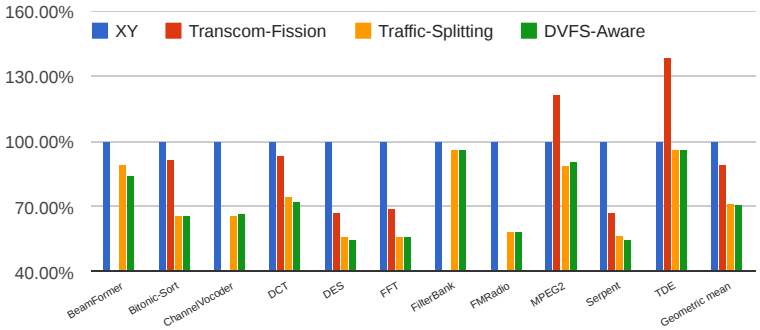


(c) Router energy improvement at 50% traffic load.

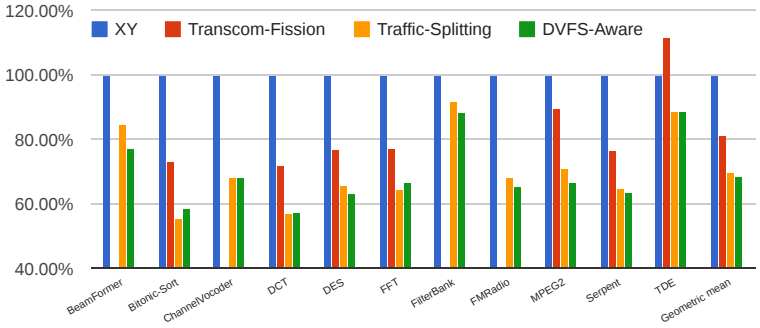


(d) Router energy improvement at 100% traffic load.

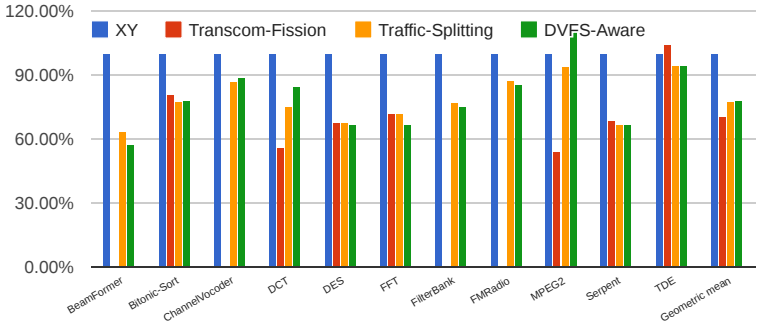
Figure 4.5: 4×4 network energy results.



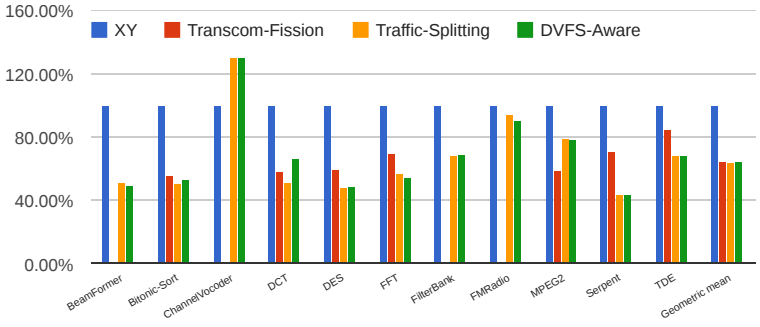
(a) Link energy improvement at 50% traffic load.



(b) Link energy improvement at 100% traffic load.



(c) Router energy improvement at 50% traffic load.



(d) Router energy improvement at 100% traffic load.

Figure 4.6: 8×8 network energy results.

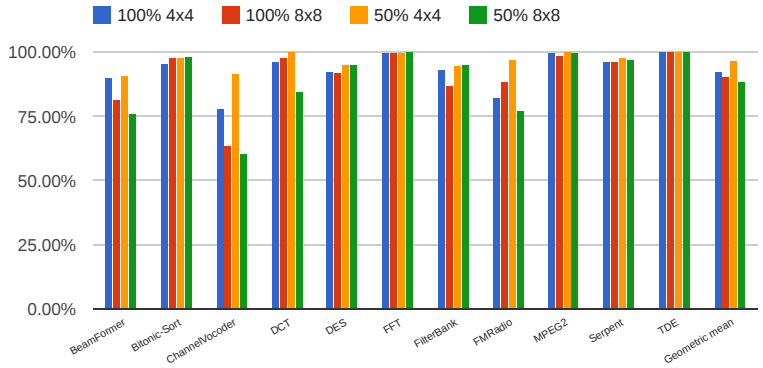
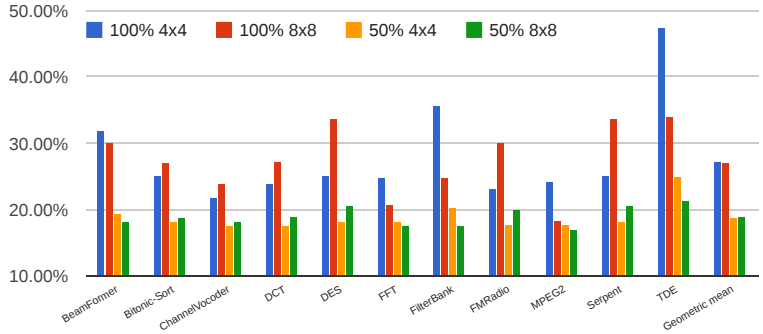
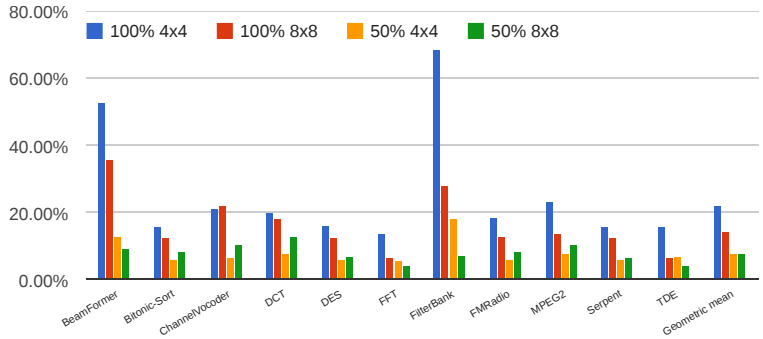


Figure 4.7: Performance penalty when applying the DVFS technique to both links and routers (normalized to non-DVFS).

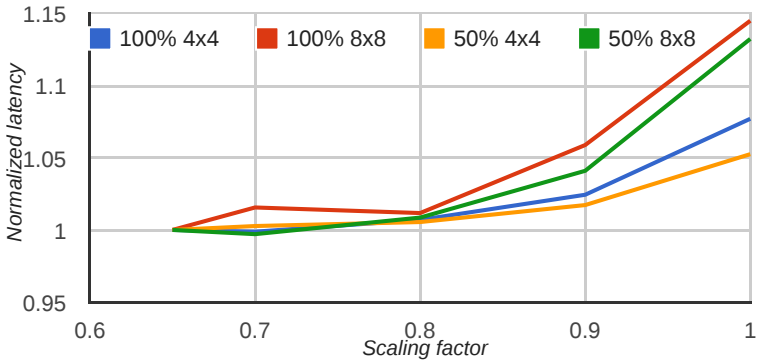


(a) Link energy improvement of DVFS over not using DVFS.

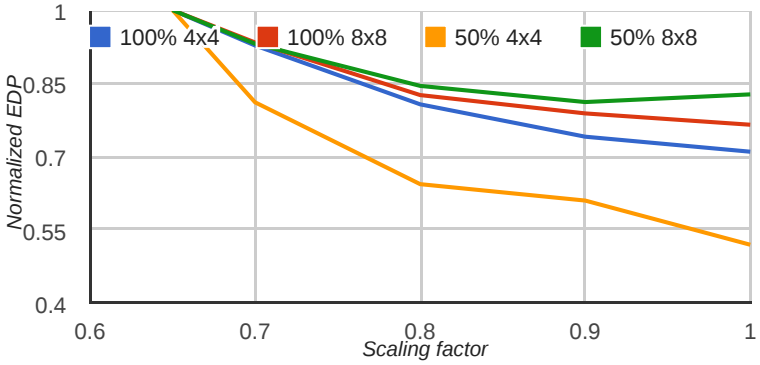


(b) Router energy improvement of DVFS over not using DVFS.

Figure 4.8: Energy effectiveness of DVFS over not using DVFS.



(a) Scaling factor vs. latency.



(b) Scaling factor vs. EDP.

Figure 4.9: Effectiveness of the router scaling factor.

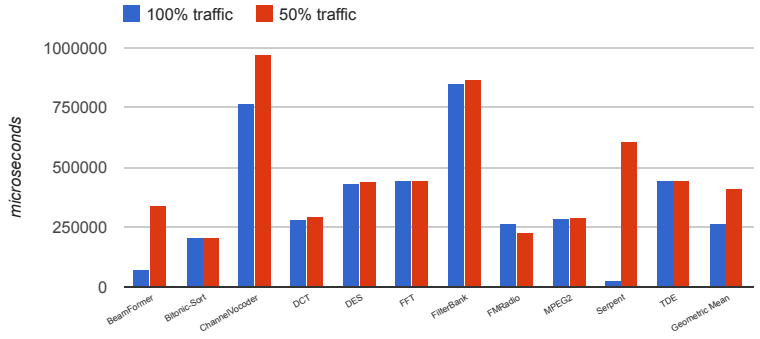


Figure 4.10: Heuristic DVFS-aware routing times on an 8×8 NoC.

Chapter 5

On the Semantics of Control-Operation-Integrated Synchronous Dataflow: Schedulability and Parallelism

We first extend and give a formal semantics for the control-operations-integrated synchronous dataflow (COSDF) programming model conceived by Thies et al. [115]. We present an analysis that determines the schedulability and parallelism of synchronous dataflow (SDF) models integrated with control operations (CO). SDF is a popular programming model for expressing the data-intensive computations, e.g., DCT, FFT, and so on, within stream programs such as MPEG2 or MP3 encoders/decoders. Within such sophisticated stream programs, the data-intensive computations need to synchronize their executions and configurations with each other by exchanging control messages (CM) containing control information, e.g., frame types in the case of MPEG2. Integrating COs into SDF imposes additional control constraints to the actor scheduling problem of SDF models. In particular, these CO constraints may render a schedulable SDF model no longer schedulable. Consequently, the existing SDF scheduling theory is no longer sufficient to handle SDF models integrated with COs. To use such models, we extend the existing SDF scheduling theory to account for the additional control constraints. Our scheduling method can determine the schedulability of SDF models with COs and produces schedules if the models are schedulable using a systematic method with dependency graphs. We implement our method in the StreamIt compiler. Our experiments indicate that our initial straightforward schedulability checking implementation may not scale well on certain large benchmarks like MPEG2 decoder. We tackle the scalability problem by first proposing a graph pruning technique that reduces dependency graph sizes by a factor up to 0.6 million times. We then come up with a circular checking technique that significantly reduces average running time by a factor up to 60000 times compared to existing methods in literature. For all available benchmarks, the schedulability checking times are less than a minute on an Intel Xeon 3GHz CPU. We show how to utilize the infrastructure

implemented for the schedulability checking problem to formally study parallelism and find schedules for SDF programs.

5.1 Introduction

SDF has been proved to be a suitable model of computation for describing the data-intensive computations such as DCT, FFT, and so on. These primitive computations are often integral parts of more sophisticated stream programs such as MP3, MPEG2 encoders/decoders. When these sophisticated stream programs are embedded into external environments, they need to not only process signals but also react to events from environments and commands from users. Adjusting internal operating configurations is one possible reaction of stream programs. These adjustments are relative to data signals, and thus require synchronization between their autonomous computations, e.g., FFT, DCT, and so on. Actors within the computations exchange CMs to synchronize their executions and configurations. COs appear in five StreamIt benchmarks [113, 112], including important benchmarks such as MPEG2 encoder/decoder. Each MPEG2 encoder/decoder benchmark uses six CMs for their COs. We expect that COs will become more prevalent in the near future when stream programs become more sophisticated in order to satisfy the increasing entertainment quality and wireless devices become more ubiquitous. In addition, if the control messaging mechanism is deployed in design environments such as Ptolemy(`http://ptolemy.eecs.berkeley.edu`) or LabVIEW(`http://www.ni.com/labview/`), users will be enabled to create their own many more applications with COs. Section 5.9 discusses in more detail about the usage scenarios of COs.

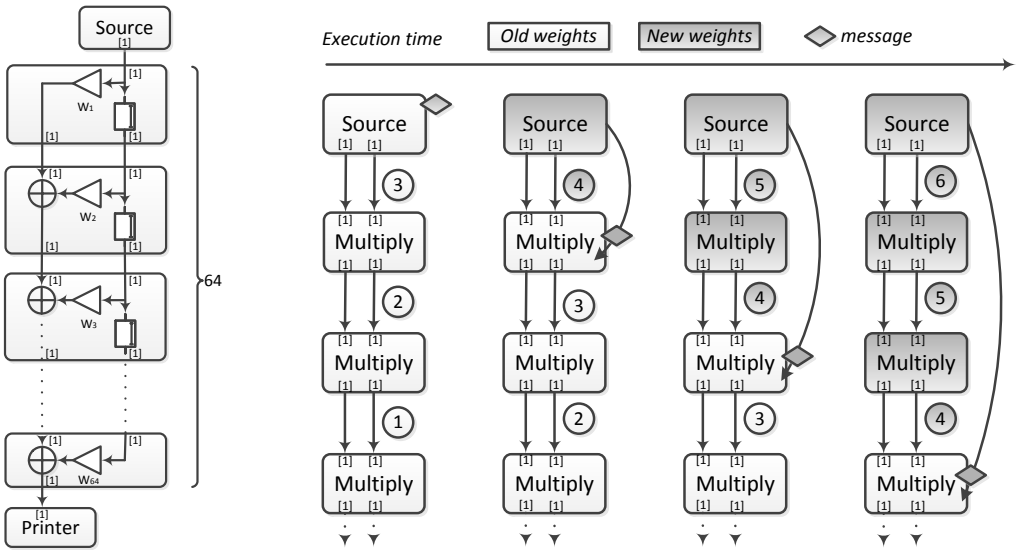
The additional CO semantics poses a new challenge for scheduling actors of stream programs. The added CO semantics makes the existing SDF scheduling theory developed in [70] no longer sufficient because the theory assumes that the dependency between executions of the actors in an SDF model only exists within one iteration of the model, while COs often impose additional dependency constraints between the executions of actors in different iterations. Even the extended scheduling method developed in [115] does not account for the COs that render CMs overlapped. The StreamIt compiler only allows non-overlapping CMs, as on the right side of Figure 5.2. The StreamIt compiler catches one specific non-schedulable case in which upstream CMs have negative latencies [115]. However, this specific case was reasoned by the compiler designers; it is not a result of a systematic approach. This conservative approach reflects an underdeveloped theory of execution dependency in the StreamIt compiler. However, in the full implementation of MPEG2 encoder/decoder benchmarks [35], e.g., in Figure 5.8, overlapping of CMs (dash arrows) does happen. These benchmarks are only able to run using a StreamIt simulation library that lets actors automatically interact to discover schedules using simulation. This approach fails to prove that the programs will not deadlock. Thies, in his PhD thesis [112], proposes a scheduling algorithm based on the auto-discovery mechanism that can potentially solve the scheduling problem of overlapping constraints. However, we did not find a proof of correctness for the scheduling method. In

particular, the semantics of CMs in [115] only account for SDF graphs without loops.

The contributions of this chapter include:

- A formal semantics for COs via the notion of information wavefronts. The formal semantics is extended to support SDF graphs with loops.
- A systematic approach for checking the schedulability of SDF graphs integrated with COs using execution dependency graphs.
- We propose two heuristic algorithms to make the schedulability checking problem feasible for large stream programs.
- We show how to exploit the formal execution dependency framework to study the formal parallelism of stream programs.

5.2 Integrating Control Operations into SDF



(a) FIR example

(b) New weights are sent to actors via control messages before the arrival of data.

Figure 5.1: Control messages example.

Control Operations

The SDF's periodic and static properties have proved to be suitable for expressing regular digital signal processing computations. However, stream applications do not only regularly

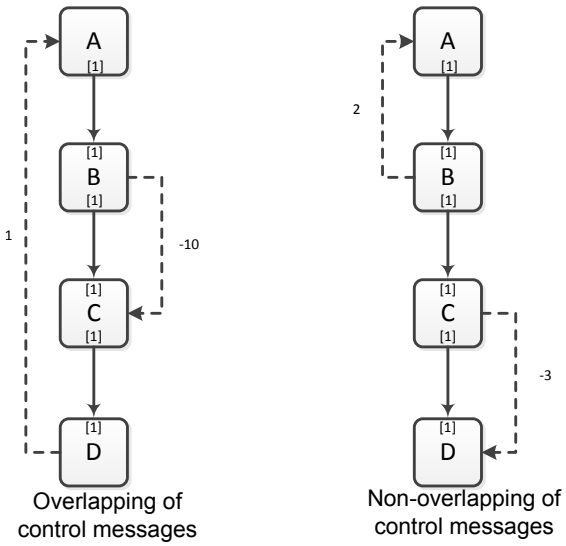


Figure 5.2: Overlapping and non-overlapping of control messages. Dashed arrows represent control communication.

compute signal data but also switch between operating configurations, e.g., to adjust amplifying volumes, feedback parameters, filter weights, employed protocols, compression rates, and so on. As SDF programs are partitioned into autonomous actors, to synchronize operating configurations, actors exchange CMs. The CM exchanging process is governed by control logic separated from data processing engines.

Let us take the Finite Impulse Response (FIR) example from [115] in Figure 5.1(a) to illustrate the notion of COs. FIR is a common kind of filters in digital signal processing where the output of the filter is a convolution of the input sequence with a finite sequence of coefficients. The FIR example is composed of one `Source`, one `Sink`, and 64 `Multiply` actors. Each `Multiply` actor has a single coefficient w, called *tap weight*. Now suppose that, during the execution, at some iteration, actor `Source` detects some condition, and decides to change the tap weights of the `Multiply` actors. The new set of weights should only be used to compute with data produced by the `Source` actor during and after its current execution. This means that the updates to the `Multiply` actors have to be precisely applied. Note that the FIR example is purely used to demonstrate the notion of CMs, real applications will have more complicated data processing stream graph structures and control logic.

Semantics of Control Information Latency

In this section, we formally give and extend the definition of CM latency informally presented by Thies et al. [115]. Our CM semantics extension accounts for SDF graphs with loops, which are not handled in [115]. Because operating configurations are specific to certain data tokens, CMs must propagate relatively to data tokens. As a result, receiving actors must process

CMs in sync with processing of data tokens. As SDF actors consume fixed numbers of data tokens in each execution, we can time CM processing by the execution numbers of receiving actors. Each CM is associated with a *latency* k that determines how much a message is delayed relatively to the data stream. To define the notion of latency, we first define the notion of *information wavefronts* [1]. Let A_m denote the m^{th} execution of actor `A`.

Definition 6 *Given* `A` *is connected to* `B` *and* `A` *is upstream of* `B`*, if execution* B_m *consumes any data produced by execution* A_n*, we say that* B_m *is data-dependent on* A_n*, denoted by* $B_m \xrightarrow{\delta} A_n$*. The data-dependent set of* A_n*, denoted as* $\texttt{DependsOn}(A_n)$*, is then defined using the following fixed point computation:*

$$\texttt{DependsOn}_0(A_n) = \{B_m | B_m \xrightarrow{\delta} A_n\}$$

$$\texttt{DependsOn}_r(A_n) = \texttt{DependsOn}_{r-1}(A_n) \bigcup \left(\bigcup_{e \in \texttt{DependsOn}_{r-1}(A_n)} \texttt{DependsOn}_0(e) \right)$$

$$\texttt{DependsOn}(A_n) = \bigsqcup \texttt{DependsOn}_r(A_n)$$

Now, let us define a set of executions that affect an execution A_n as follows:

$$\texttt{Affects}(A_n) = \{B_k | A_n \in \texttt{DependsOn}(B_k)\} \tag{5.1}$$

Using $\texttt{DependsOn}(A_n)$ and $\texttt{Affects}(A_n)$, we will define the information wavefront set $\texttt{WF}(A_n)$ of A_n, which captures the executions that propagate information to/from A_n.

Definition 7 *An execution* A_n *is said to trigger execution* B_m *iff* B_m *can only happen after* A_n *and no further execution of* `A` *is needed to enable* B_m*. Concretely,* A_n *triggers* B_m *iff* $B_m \in \texttt{DependsOn}(A_n)$ *and* $B_m \notin \texttt{DependsOn}(A_{n+1})$*.*

Intuitively, B_m is in the wave of executions made possible by data produced by A_n.

For an execution A_n, we define an upstream information wavefront set $\texttt{WF}^{\uparrow}(A_n)$ of upstream executions that propagate information to A_n. $\texttt{WF}^{\uparrow}(A_n)$ is a set of executions *triggering* A_n and not A_{n-1}, so information can flow directly from the executions to A_n. As a result:

$$\texttt{WF}^{\uparrow}(A_n) = \texttt{Affects}(A_n) \setminus \texttt{Affects}(A_{n-1}) \tag{5.2}$$

Similarly, the downstream wavefront set $\texttt{WF}^{\downarrow}(A_n)$ of A_n is composed of the executions that are directly triggered by A_n but not A_{n+1} so that information can flow directly to the executions from A_n without looping through `A` again, e.g., to A_{n+1}:

$$\texttt{WF}^{\downarrow}(A_n) = \texttt{DependsOn}(A_n) \setminus \texttt{DependsOn}(A_{n+1}) \tag{5.3}$$

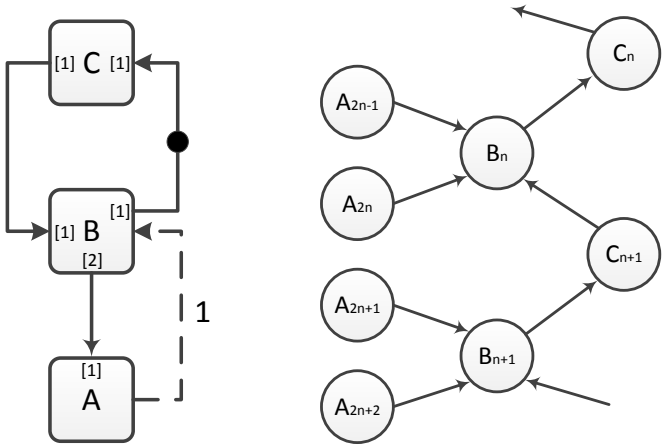
[1] The notion of information wavefronts was conceived informally by Thies et al. [114].

Definition 8 *The information wavefront set,* $\texttt{WF}(A_n)$*, of an execution* A_n *is defined as follows:*

$$\texttt{WF}(A_n) = \texttt{WF}^{\uparrow}(A_n) \cup \texttt{WF}^{\downarrow}(A_n) \tag{5.4}$$

Definition 9 *The latency* k *of a CM sent from execution* S_n *of actor* `S` *is intended to be processed by the receiving actor* `R` *in its execution* R_m *where* $R_m \in \texttt{WF}(S_{n+k})$*. Ideally,* R_m *is the most recent in* $\texttt{WF}(S_{n+k})$*, e.g.,* m *is smallest. If there is no* m *such that* $R_m \in \texttt{WF}(S_{n+k})$*, we then find such* R_m *in* $\texttt{WF}(S_{n+l})$ *where* $l > k$ *and* l *is minimal.*

Intuitively, the latency of a CM denotes the number of information wavefronts of the sender of the CM that the control information is designated to *traverse*. The reason for choosing $\texttt{WF}(S_{n+l})$ where $l > k$ and l is minimal when R_m does not exist in $\texttt{WF}(S_{n+k})$ is that, while S_{n+k} does not directly trigger R_m if `S` is upstream of `R` or is directly triggered by R_m otherwise, it contributes to data eventually consumed by R_m (if `S` is upstream of `R`) or consumes data derived from R_m (if `S` is downstream of `R`).



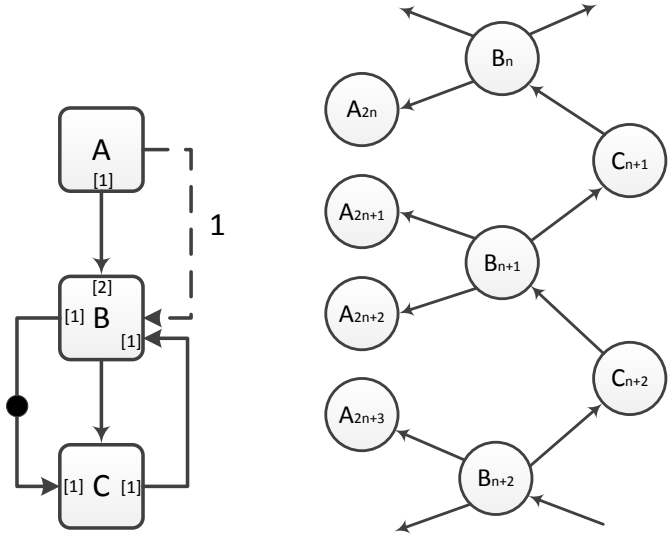
(a) Stream graph. (b) Dependency graph.

Figure 5.3: Upstream information wavefront example.

Let us take the stream graph in Figure 5.3(a) to illustrate the upstream latency concept. Figure 5.3(b) shows the corresponding dependency graph. Based on the dependency graph, we have:

$$\texttt{WF}^{\uparrow}(A_{2n}) = \texttt{Affects}(A_{2n}) \setminus \texttt{Affects}(A_{2n-1}) = \emptyset$$
$$\texttt{WF}^{\uparrow}(A_{2n+1}) = \texttt{Affects}(A_{2n+1}) \setminus \texttt{Affects}(A_{2n}) = \{B_{n+1}, C_{n+1}\}$$

$\texttt{WF}^{\uparrow}(A_{2n}) \equiv \emptyset$ because both A_{2n-1} and A_{2n} depend on B_n. Suppose that in execution A_{2n-1}, a CM is sent from `A` to `B` with latency 1. Because $\texttt{WF}^{\uparrow}(A_{2n}) \equiv \emptyset$, the message is processed in $\texttt{WF}^{\uparrow}(A_{2n+1})$; specifically it is processed in execution B_{n+1}. Because both A_{2n-1} and A_{2n}



(a) Stream graph. (b) Dependency graph.

Figure 5.4: Downstream information wavefront example.

consume data from B_n, it is legitimate that the CM with latency 1 be processed in $B_{n+1} \in \texttt{WF}^{\uparrow}(A_{2n+1})$.

Let us take the stream graph in Figure 5.4(a) to illustrate the downstream latency concept. Figure 5.4(b) shows the corresponding dependency graph. Based on the dependency graph, we have:

$$\texttt{WF}^{\downarrow}(A_{2n+1}) = \texttt{DependsOn}(A_{2n+1}) \setminus \texttt{DependsOn}(A_{2n+2}) = \emptyset$$
$$\texttt{WF}^{\downarrow}(A_{2n+2}) = \texttt{DependsOn}(A_{2n+2}) \setminus \texttt{DependsOn}(A_{2n+3}) = \{B_{n+1}, C_{n+2}\}$$

$\texttt{WF}^{\downarrow}(A_{2n+1}) \equiv \emptyset$ because both A_{2n} and A_{2n+1} contribute to the data consumed by B_n. Now in execution A_{2n}, a CM with latency 1 is sent to B. Because $\texttt{WF}^{\downarrow}(A_{2n+1}) \equiv \emptyset$, the message is processed within $\texttt{WF}^{\downarrow}(A_{2n+2})$; specifically, it is processed in execution B_{n+1}. While $\texttt{WF}^{\downarrow}(A_{2n+1}) \equiv \emptyset$, A_{2n+1} contributes to the data consumed by B_{n+1}, it is legitimate that the message be processed in B_{n+1}.

Note that our generalized definition of latency k does account for SDF graphs with loops, which are not handled in [115].

COSDF Schedulability Problem

Because COs often impose additional control constraints on the execution orders of actors in SDF programs, the SDF scheduling theory [70] developed for data signal processing computations is no longer sufficient. The extended scheduling method developed by Thies et al. [115] does not handle the case when CMs overlap on some actors on their data paths, called

the overlapping constraint problem. As a consequence, schedulable stream programs can be dubbed non-schedulable because the compiler cannot analyze these situations.

We tackle the schedulability problem by developing a dependency analysis method to account for the overlapping constraint situations. Let us take an example on the left of Figure 5.2 to explain our analysis method and the schedulability problem. Four actors `A, B, C, D` in the example form a pipeline. The solid arrows between actors denote data connections. The dashed arrows are control connections used to carry CMs. The values beside those dashed arrows are latencies of CMs.

Based on the CM latency semantics in the previous section and the semantics of SDF, we have the following constraints:

- Suppose that `D` is at its n^{th} execution. The stream graph indicates that CMs from `D` to `A` have latency 1 and `D` is *downstream* of `A`. When applying the latency semantics, `A` may have to process a possible CM[2] from D_n right *before* its execution in $\texttt{WF}(D_{n+1})$, which is A_{n+1}. Hence, A_{n+1} has to happen after D_n. Let us denote this as $D_n \prec A_{n+1}$.
- Via the data path, B_{n+1} consumes one token produced by A_{n+1}, we have $A_{n+1} \prec B_{n+1}$.
- We assume that multiple executions of a single actor must proceed in sequential order, so $B_{n+1} \prec B_{n+2}$, and hence, of course, $B_{n+1} \prec B_{n+10}$.
- Suppose that `B` is at its n^{th} execution. The latency for CMs from `B` to `C` is -10 and `B` is *upstream* of `C` as in the figure. `C` has to process a possible CM from B_n right *before* its execution in $\texttt{WF}(B_{n+(-10)})$, which is C_{n-10}. Therefore, C_{n-10} has to happen after B_n: $B_n \prec C_{n-10}$. Equivalently, $B_{n+10} \prec C_n$.
- Finally, D_n consumes one token produced by C_n, therefore $C_n \prec D_n$.

Summing up, we have a set of dependency constraints for the example: $D_n \prec A_{n+1} \prec B_{n+1} \prec B_{n+10} \prec C_n \prec D_n$. These dependency constraints create a cycle of dependencies, so no evaluation order exists and the system is deadlocked.

We identify two factors contributing to cyclic dependencies of actor executions in a stream graph: 1) the structure of the stream graph, and 2) the latencies of CMs. Let us take an example to illustrate the importance of the two factors. With the same graph structure on the left side of Figure 5.2, however, the latency for CMs between actor `B` and actor `C` is 0. In this case, there exists a valid evaluation order: $\ldots \prec A_n \prec B_n \prec C_n \prec D_n \prec A_{n+1} \prec B_{n+1} \prec \ldots$. For the same stream graph *structure*, different CM latencies can lead to different results; the first one is a deadlock while the second one has a valid schedule.

Thies et al. [115] call the graph structure on the left of Figure 5.2 "overlapping constraints" because the data paths between the actors involved in control messaging have overlapping actors. In this example, the overlapping actors are `B` and `C`. Constraints imposed by multiple CMs on those overlapping actors form overlapping constraints. The StreamIt

[2] At its n^{th} execution, `D` may send or may not send a CM as CMs are infrequent.

compiler simply rejects graph structures that have overlapping CM situations regardless of message latencies even if the latencies can result in valid schedules, as is the case when the CM latency between actor `B` and actor `C` is 0.

We will present a scalable and systematic method to check for such circular dependencies by constructing dependency graphs for stream programs. We then show how to use the constructed dependency graphs to study parallelism of stream programs.

5.3 Schedulability of COSDF

Thies et al. [115] use Teleport messaging (TMG) to call control messaging (CMG). We will use CMs in place of Teleport messages in this chapter.

Concrete CM Execution Model

With the CMG approach illustrated in Figure 5.1(b), actor `Source` can send CMs containing new weights directly to each `Multiply` actor before the arrival of data tokens that need to be computed with the new weights. This separation between COs and data computation makes it easier to maintain and to debug programs. It also helps avoid the error-prone task of manually embedding and processing control information within data tokens. And later in this chapter, we will see that it also eases the compositional verification of the programs.

CM Timing with `SDEP`

Thies et al. [115] present an approach for finding processing times of CMs at receiving actors using the `SDEP` function based on senders-receivers data dependencies in SDF graphs[3].

Execution Model of CMG: Based on `SDEP` function and the semantics of CMs in Section 5.2, we provide the formal semantics of CMs based on Thies's PhD thesis [112] as follows:

Definition 10 *Suppose that actor* `S` *sends a CM to actor* `R` *with latency* k *during the* n^{th} *execution of* `S`*, then the message must be processed immediately* before `R`*'s* m^{th} *execution, where:*

- *If* `R` *is upstream of* `S`*:*

$$m = \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n + k - 1) + 1 \tag{5.5}$$

- *If* `R` *is downstream of* `S`*:*

$$m = \min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n + k\} \tag{5.6}$$

[3]Note that the CM latency semantics in Section 5.2 is not necessarily restricted in SDF graphs.

- *If* R *is both upstream and downstream of* S *(e.g., a stream graph with loops):*

$$m = \min\{\min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+k\}, \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1\} \quad (5.7)$$

First, we will explain the reason leading to equation (5.5). Because R is upstream of S, as a result, $R_m \in \texttt{WF}^{\uparrow}(A_{n+l})$ where $l = \min_{t\geq k; \exists R_m \in \texttt{WF}^{\uparrow}(S_{n+t})} t$. Expanding using equation (5.2) leads to $R_m \in \texttt{Affects}(S_{n+l})$ and $R_m \notin \texttt{Affects}(S_{n+k-1})$. Because we can always find l such that $R_m \in \texttt{Affects}(S_{n+l})$, the only necessary condition left is $R_m \notin \texttt{Affects}(S_{n+k-1})$. $R_m \notin \texttt{Affects}(S_{n+k-1})$ implies $m > \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)$. So if we chose the most recent R_m, then $m = \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1$. Similarly for equation (5.6), we have $R_m \in \texttt{DependsOn}(S_{n+k})$. Because $R_m \in \texttt{DependsOn}(S_{n+k})$ implies $\texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m) \geq n+k$ and we chose the most recent R_m, then $m = \min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+k\}$. Equation (5.7) is just the combination of (5.5) and (5.6).

To illustrate how to use SDEP to find the appropriate executions of receiving actors, we take the FIR example in Figure 5.1(b) and find the m^{th} execution of a Multiply that a CM has to be processed right before. Suppose that, at its 5^{th} execution ($n = 5$), actor Source sends a CM to a Multiply actor with latency $k = 2$. We have:

$$\texttt{SDEP}_{\texttt{Source}\leftarrow\texttt{Multiply}}(m') \geq n+k = 7$$

Each time actor Source executes, it produces one token and each time one Multiply actor executes, it produces one token and consumes one token. Therefore, in order for a Multiply actor executes m' times, actor Source has to execute m' times. In other words, $\texttt{SDEP}_{\texttt{Source}\leftarrow\texttt{Multiply}}(m') = m'$. Hence, $m' \geq 7$. Because $m = \min\{m'\}$, as a result, $m = 7$.

Characterizing Execution Dependencies

The COSDF schedulability problem in Section 5.2 hinders potential interesting applications whenever their stream graphs have *overlapping constraints* of CMs[4] even though the applications are schedulable. The StreamIt compiler considers a program invalid when it has upstream CMs with negative latencies. This specific checking rule is reasoned by the compiler designers [115]. Checking for invalid cases for general stream graph structures with overlapping of CMs is not straightforward due to the mis-matching input/output rates of actors.

Dependency Graphs

To check for circular dependencies, we can construct dependency graphs (DG) capturing the execution dependency between SDF actors and check for cycles in the graphs. If there is no cycle in a DG, then the graph is directed acyclic and a schedule can be found using

[4]The overlapping constraints of CMs actually depends on semantics of applications.

a topological sort. The construction of such DGs is done in two steps. First, we replicate the executions of the actors of an SDF graph. These executions form the vertices of a DG. Second, we add dependency edges between these vertices. Constructing such DGs requires characterizing various kinds of execution dependencies in stream programs.

Actor Execution Dependencies

Definition 11 *Execution dependency: An execution e_1 of an actor is said to be dependent on another execution e_2 of some actor (can be the same actor) when e_1 has to wait until e_2 has finished before it can commit its results.*

We characterize three kinds of execution dependency as follows:

- Sequential dependency: The $(n+1)^{th}$ execution of $\texttt{A}$ by definition happens after the n^{th} execution of $\texttt{A}$, or $A_n \prec A_{n+1}$. We have the corresponding edge:$A_{n+1} \rightarrow A_n$.
- Data dependency: Let $\texttt{B}$ be downstream of $\texttt{A}$. An n^{th} execution of $\texttt{B}$ will be data-dependent on an m^{th} execution of $\texttt{A}$ if $B_n \in \texttt{DependsOn}(A_m)$. We utilize the $\texttt{SDEP}$ function to derive such dependencies. For any two actors, upstream $\texttt{A}$ and downstream $\texttt{B}$, such that $\exists n, m \in \mathbb{N}, B_n \in \texttt{DependsOn}(A_m)$, we create a data-dependency edge from the n^{th} execution of $\texttt{B}$ to the $\texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n)^{th}$ execution of $\texttt{A}$, denoted as $A_{\texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n)} \leftarrow B_n$. Note that, based on sequential dependency condition, $A_{m-1} \prec A_m$, as a result, $A_m \prec B_n, \forall m \leq \texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n)$. Thus, we do not need to add any dependency edges between B_n and $A_m, \forall m < \texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n)$, as those dependencies are *implicit* and can be inferred from $A_m \prec A_{m+1}$ and $A_{\texttt{SDEP}_{\texttt{A}\leftarrow\texttt{B}}(n)} \prec B_n$.
- Control dependency due to CMs sent between actors. Because an actor $\texttt{S}$ at its n^{th} execution may send a CM to an actor $\texttt{R}$ with latency k, then for all the m^{th} executions of $\texttt{R}$ satisfying the Definition 10, R_m is said to be control dependent on S_n as it may consume some control information from S_n. We create an edge $R_m \rightarrow S_n$. According to Definition 10, $\texttt{R}$ has to process the message right *before* its m^{th} execution:

  There are three cases:

  - If $\texttt{R}$ is downstream of $\texttt{S}$: $m = \min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+k\}$. The created edge $R_m \rightarrow S_n$ becomes $R_{\min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+k\}} \rightarrow S_n$.
  - If $\texttt{R}$ is upstream of $\texttt{S}$: $m = \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1$. The created edge $R_m \rightarrow S_n$ becomes $R_{\texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1} \rightarrow S_n$.
  - If $\texttt{R}$ is both upstream and downstream of $\texttt{S}$ (there exists a loop in the stream graph): $m = \min\{\min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+k\}, \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1\}$. The created edge $R_m \rightarrow S_n$ becomes $R_{\min\{\min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+k\}, \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1\}} \rightarrow S_n$.

Illustrative Example: Let us come back to the example in Figure 2.1 to illustrate our method. Within one iteration of the whole stream graph, according to the SDF model of computation, actors `A,B,C,D` and `E` execute 3, 3, 2, 2, and 2 times respectively. Each execution of an actor is replicated as one vertex in the DG as in Figure 5.5. A_2^i denotes the 2^{nd} *relative* execution of actor `A` within the i^{th} iteration of the stream graph. A_2^i corresponds to the $(i*3+2)^{th}$ *absolute* (from the beginning when the program starts) execution of actor `A` as `A` executes 3 times in one iteration for the stream graph in Figure 2.1[5].

Figure 5.5 shows the DG when `E` sends CMs to `A` with latency 1 and `B` sends CMs to `D` with latency -2. In the figure, the sequential dependency edges are dashed arrows, data dependency edges are dash-dot arrows, and control dependency edges are solid arrows. We use the `SDEP` function between actors `B` and `D` in Table 2.1 to illustrate our method. For any n^{th} iteration of the stream graph, we add data dependency edges $D_1^n \rightarrow B_2^n$ and $D_2^n \rightarrow B_3^n$ as in Figure 5.5 because within one iteration, the first execution of `D` is data-dependent on the second execution of `B` and the second execution of `D` is data-dependent on the third execution of `B`. For control dependency edges, we apply the formula $D_{\min\{m'|\texttt{SDEP}_{\texttt{B}\leftarrow\texttt{D}}(m')\geq n+k\}} \rightarrow S_n$ for CMs from `B` to `D` with a latency $k = -2$.

The above naive graph construction process will produce *infinite* graphs as stream applications are presumed to run forever. Because we cannot verify if infinite graphs do not contain a cycle, we do not know for sure if a COSDF program will be deadlocked or not. To tackle this problem, we can translate infinite DGs into equivalent *finite* graphs that preserve the circular dependency attribute. As the SDF model of computation is periodic, although DGs of SDF models are infinite, they are *periodic* [90]. We employ a technique to reduce infinite DGs into corresponding weighted finite graphs, called reduced dependency graphs (RDG), pioneered by Karp, Miller and Winograd [61]. Darte presents a nice summary of work related to the technique in [33]. The basic idea is to translate periodic DGs into corresponding RDGs. Checking for cycles in periodic DGs is equivalent to checking for zero-weight cycles in RDGs.

Reduced Dependency Graphs

Definition 12 *A directed periodic graph $G^\infty = (V^\infty, E^\infty)$ is induced by a RDG $G = (V, E, T)$, where V is the set of vertices, E is the set of edges, $T : E \rightarrow \mathbb{Z}^k$ is a weight function on the edges of G, via the following expansion:*

$$V^\infty = \{v^p | v \in V, p \in \mathbb{Z}^k\}$$
$$E^\infty = \{(u^p, v^{p+t_{uv}}) | (u, v) \in E, t_{uv} \in T, p \in \mathbb{Z}^k\}$$

[5]To make this conversion more clear, we take an example. Suppose that an actor `A` has executed n times since a program starts and in each iteration of the program, `A` executes a times. Then we can calculate that execution A_n belongs to $i = n \div a$ iteration of the whole program (based on SDF semantics) and it is the $r = (n \mod i)$ execution of `A` within the i^{th} iteration of the program. In other words $A_n \Leftrightarrow A_{n \mod a}^{n \div a}$. We call n the *absolute* execution, r the *relative* execution, and i the iteration index. We will use this conversion frequently in the next sections.

where $t_{uv} \in T$ represents the number of k-dimensional *periods* it takes to travel from u's period to v's period along the edge. The vertex v^p of G^∞ can be interpreted as vertex v of G in a k-dimensional *period* p. Edge $(u^p, v^{p+t_{uv}})$ represents *travelling* from u in period p and arriving at v by t_{uv} periods later. Intuitively, a k-dimensional periodic graph is obtained by replicating a basic graph (cell) in a k-dimensional orthogonal grid. Each vertex within the basic graph is connected with a finite number of other vertices in other replicated basic graphs and the inter-basic-graph connections are the same for each basic graph.

The DG of a SDF stream graph is an infinite 1-dimensional periodic graph with its basic graph composed of the vertices of actor executions within one iteration. The basic cell is repeatedly put in a 1-dimensional time grid. Data, sequential and control dependencies induce the directed edges between the vertices. As the SDF model of computation is periodic by nature, the pattern of the inter-cell (inter-iteration) connections is the same for each cell (iteration).

Translating to RDGs

Figure 5.6 shows the corresponding RDG of the DG in Figure 5.5. In the graph, all the edges without specified weights are of zero weight. Intuitively, all the vertices within one arbitrary iteration, say iteration n^{th}, are kept to form the vertices in the corresponding RDG after removing iteration indices, e.g., A_1^n becomes A_1. Directed edges between vertices within one iteration are also kept and their weights are set to 0. For directed edges crossing iterations, only *outgoing* edges (edges from this iteration to some other iterations) are used to translate to the corresponding edges in the RDG. The translation is done as follows. Suppose that an outgoing edge is $S_x^n \to R_y^m$, we add a directed edge $S_x \to R_y$ with weight $n - m$. $n - m$ is called *relative iteration*, which is the gap between the iterations of two actor executions. For example, the directed edge $D_2^n \to B_2^{n+1}$ in Figure 5.5 becomes the edge $D_2 \to B_2$ with weight -1 in Figure 5.6. Note that an edge $S_x \to R_y$ is equivalent to *any edge* $S_x^i \to R_y^j$ in the execution dependency graph as long as $i - j = n - m$ because of the repetitive property of the SDF model of computation. This translation process could be understood as folding SDF DGs into corresponding RDGs at period boundaries.

Graph Correspondence

We cite the following lemma from Lemma 1 in [90].

Lemma 1 *Let $G = (V, E, T)$ be a RDG graph. For $u, v \in V$ and $s, d \in \mathbb{Z}$, there is a one-to-one canonical correspondence between the set of finite paths from $u^s \to v^d$ in G^∞ and the set of paths in G from u to v with transit time $s - d$.*

The above lemma is instrumental in proving the following theorem:

Theorem 3 *One dependency cycle in a DG corresponds to one cycle of zero length in the corresponding RDG and vice-versa.*

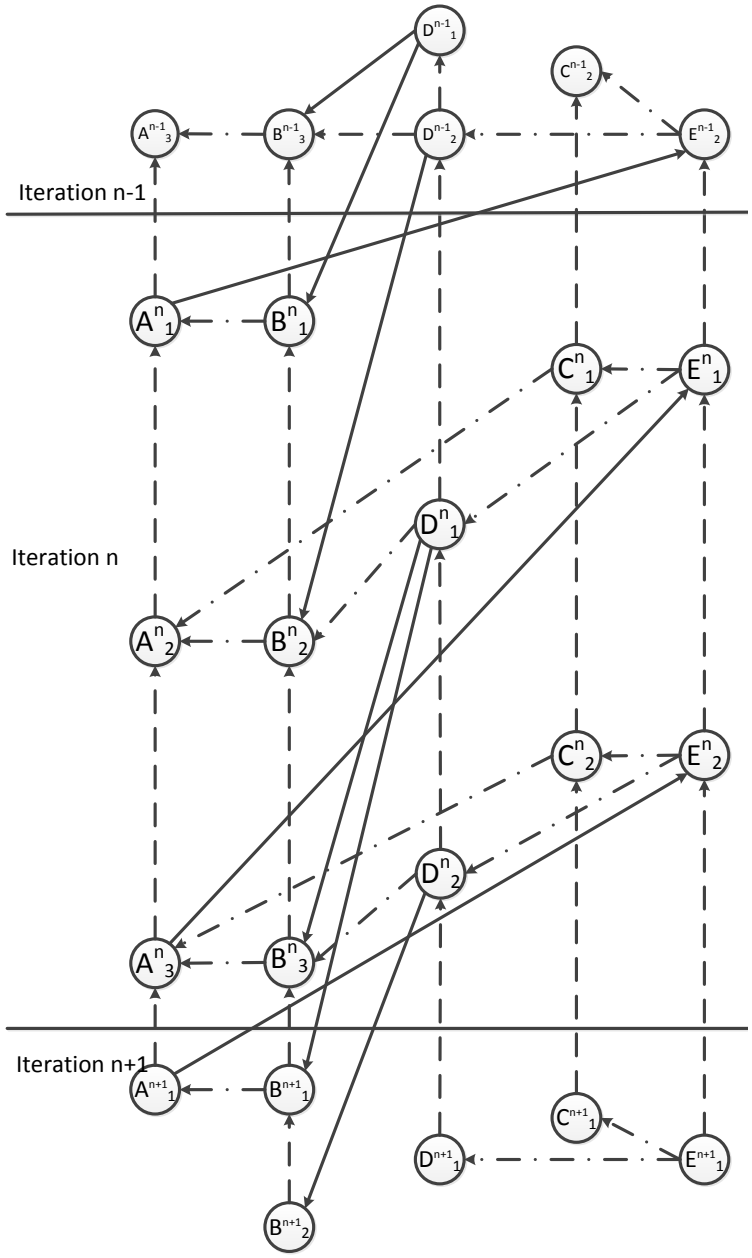


Figure 5.5: Infinite periodic dependency graph.

Proof: Suppose that there is a cycle in a DG, say $X_{i_1}^{n_1} \rightarrow X_{i_2}^{n_2} \rightarrow \ldots \rightarrow X_{i_m}^{n_m} \rightarrow X_{i_1}^{n_1}$. By Lemma 1, this cycle corresponds to one directed cycle with edges $(X_{i_1} \rightarrow X_{i_2})$, $(X_{i_2} \rightarrow X_{i_3})$, $\ldots$,$(X_{i_{m-1}} \rightarrow X_{i_m})$, $(X_{i_m} \rightarrow X_{i_1})$ of weights $(n_1 - n_2), (n_2 - n_3), \ldots, (n_{m-1} - n_m), (n_m - n_1)$, respectively. The sum of the weights of the edges in the corresponding directed cycle is: $(n_1 - n_2) + (n_2 - n_3) + \ldots + (n_{m-1} - n_m) + (n_m - n_1) = 0$.

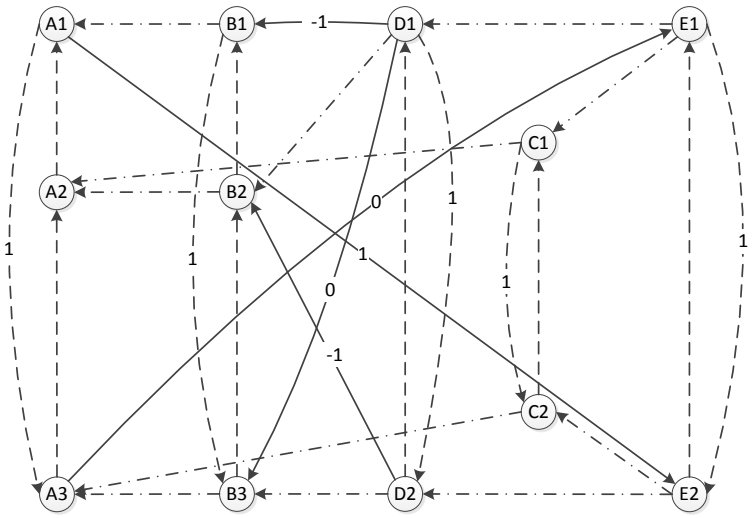


Figure 5.6: Reduced dependency graph.

For one zero directed cycle with edges $(X_{i_1} \to X_{i_2})$, $(X_{i_2} \to X_{i_3})$, $\ldots$,$(X_{i_{m-1}} \to X_{i_m})$, $(X_{i_m} \to X_{i_1})$ of weights $w_1, \ldots, w_m$ respectively in a RDG, we chose some very large n_1 and derive $n_2 = n_1 - w_1, n_3 = n_2 - w_2, \ldots, n_m = n_{m-1} - w_m$. n_1 should be chosen to be large enough so that $n_2, \ldots, n_m > 0$. By Lemma 1, the zero directed cycle corresponds to one cycle of dependencies $X_{i_1}^{n_1} \to X_{i_2}^{n_2} \to \ldots \to X_{i_m}^{n_m} \to X_{i_1}^{n_1}$ in the corresponding DG ■

Detecting Zero Cycles

We have shown that a cycle of dependencies corresponds to a cycle of zero weight in a RDG. In [55], Iwano and Steiglitz propose an algorithm for detecting zero cycles in an 1-dimensional[6] RDG with complexity $\mathsf{O}(|V|^3)$ (Theorem 4 in [55]).

Illustrative Example

To illustrate the cycle checking method better, we take the translated RDG in Figure 5.6. Running the zero cycle detection algorithm in [55] on the graph, we find a zero cycle A_1, E_2, D_2, B_2, A_2, which corresponds to the cycle of dependencies in the dependency graph in Figure 5.5, $A_1^{n+1} \succ E_2^n \succ D_2^n \succ B_2^{n+1} \succ A_2^{n+1} \succ A_1^{n+1}$.

Now, suppose that the latency of CMs from `E` to `A` is 3, then the RDG has no zero cycles, thus, there is no cycle of dependencies in the DG. As a result, the set of constraints imposed by the control communications is feasible.

[6]Distances between vertices have only one dimension.

5.4 Direct Construction of RDGs

In the previous section, we show how to translate from infinite DGs to RDGs to check for cycles of dependencies. However, constructing infinite DGs from stream programs is not possible and not necessary. Instead, we can use a similar mechanism to construct RDGs directly from stream programs as we know the repetitive dependency structures across iterations of SDF stream programs. Algorithm 5.7 shows how to directly construct RDGs from stream graphs.

Similar to infinite DGs, RDGs are constructed in two steps. First, the executions of one actor are replicated the same number of times that the actor executes *within one iteration* of an SDF graph. Each execution becomes one vertex in the RDG. Second, we add dependency edges between the vertices based on the three kinds of dependency presented in Section 5.3.

In Algorithm 5.7, the `get_num_reps` function returns the number of repetitions of an actor within one iteration of the entire stream graph. The function `compute_rel_iter_exe` computes *relative* iterations i_r and *relative* executions e_r of CMG receiving actors as follows. Suppose that a CMG sender $\mathtt{S}$ at its absolute n^{th} execution may send a message to a receiver $\mathtt{R}$ with latency k. Then the absolute m^{th} execution of the receiver $\mathtt{R}$ will be computed as in Section 5.3. We use the conversion from absolute executions of actors to relative executions and iterations in Section 5.3. Suppose that $\mathtt{S}$ and $\mathtt{R}$ execute $s = |\mathcal{S} \wedge \mathtt{S}|$ and $r = |\mathcal{S} \wedge \mathtt{R}|$ times within one iteration $\mathcal{S}$ of a stream graph respectively. We compute the iterations $i^{\mathtt{S}}, i^{\mathtt{R}}$ and the relative executions $r^{\mathtt{S}}, r^{\mathtt{R}}$ of $\mathtt{S}$ and $\mathtt{R}$ respectively from their corresponding absolute executions n^{th} and m^{th} as follows:

$$r^{\mathtt{S}} = n \mod s, \qquad i^{\mathtt{S}} = n \div s$$
$$r^{\mathtt{R}} = m \mod r, \qquad i^{\mathtt{R}} = m \div r$$

then the relative iteration $i_r = i^{\mathtt{S}} - i^{\mathtt{R}}$ and $e_r = r^{\mathtt{R}}$. The above calculation method for relative iterations and relative executions is specific to an n^{th} execution of $\mathtt{S}$. The immediate question is how to pick n to compute i_r. The following theorem presents an answer to the question.

Theorem 4 *For a fixed relative execution $r^{\mathtt{S}}$, relative execution i_r is invariant to absolute iteration $i^{\mathtt{S}}$. Concretely, i_r is the same for $i^{\mathtt{S}}$ and $i^{\mathtt{S}} + j$ with $j \in \mathbb{Z}$.*

Proof: Suppose that we consider the same relative execution of $\mathtt{S}$ in j iterations later of the stream graph, say iteration $i^{\mathtt{S}}+j$, then the absolute execution of $\mathtt{S}$ is $r^{\mathtt{S}}+(i^{\mathtt{S}}+j)*s = n+s*j$. We have three cases:

- If $\mathtt{R}$ is downstream of $\mathtt{S}$. Note that SDF is periodic, therefore, if $m = \min\{m' | \mathtt{SDEP}_{\mathtt{S}\leftarrow\mathtt{R}}(m') \geq n + k\}$ then $m + r * j = \min\{m' | \mathtt{SDEP}_{\mathtt{S}\leftarrow\mathtt{R}}(m' + r * j) \geq n + s * j + k\}$ based on equation (2.5). Thus, for j iterations later of $\mathtt{S}$, we still have: $i_r = ((n + s * j) \div s) - ((m + r * j) \div r) = (i^{\mathtt{S}} + j) - (i^{\mathtt{R}} + j) = i^{\mathtt{S}} - i^{\mathtt{R}}$.

```
1  (V, E, T) ← (∅, ∅, ∅)
2  sched ← compute_SDF_schedule(streamGraph)
3                                                          ▷ Create the set of vertices
4  forall the actor do
5      for exe = 1 → sched.get_num_reps(actor) do
6          V ← V + new_vertex(actor, exe)        ▷ Each vertex is one actor execution in one iteration
7      end
8  end
9                                                          ▷ Add data dependency edges
10 forall the v ∈ V do
11     forall the actor ∈ upstream_actors(v.actor) do
12         absolute_exe ← SDEP_{actor←v.actor}(v.exe)   ▷ Translate from absolute execution to relative one
           iteration ← (absolute_exe − 1) ÷ sched.get_num_reps(actor)
13         exe ← 1 + (absolute_exe − 1)  mod sched.get_num_reps(actor)
14         u ←get_vertex(actor, exe)
15         e ←new_edge(u, v)
16         E ← E + e
17         T ← T + (weigth(e) ← (−iteration))
18     end
19 end
20                                                         ▷ Add sequential dependency edges
21 forall the v ∈ V do
22     if v.exe > 1 then
23         u ← get_vertex(v.actor, v.exe + 1)        ▷ Edges within one SDF iteration have weight 0
24         e ← new_edge(v, u)
25         E ← E + e
26         T ← T + (weigth(e) ← 0)
27     end
28     else
29         u ← get_vertex(v.actor, sched.get_num_executions(actor))        ▷ Edges to previous SDF
           iterations have weight 1 e ← new_edge(v, u)
30         E ← E + e
31         T ← T + (weigth(e) ← 1)
32     end
33 end
34                                                         ▷ Add control dependency edges
35 forall the actor do
36     if send_control_msg(actor) then
37         for exe = 1 → sched.get_num_reps(actor) do
38             forall the recv ∈ get_control_receivers(actor) do
39                 k ← get_latency(actor, recv)  ▷ Get control message latency  ▷ Determine relative
                   iterations and relative executions of control receivers
                   (i_r, e_r) ← funccompute_rel_iter_exe(actor, recv, exe, k)
40                 s ← get_vertex(actor, exe)
41                 r ← get_vertex(recv, e_r)
42                 e ← new_edge(s, r)
43                 E ← E + e
44                 T ← T + (weigth(e) ← i_r)    ▷ Relative iteration of each receiver is the edge weight
45             end
46         end
47     end
48 end
```

Figure 5.7: Constructing reduced dependency graphs

- If R is upstream of S then $m = \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1$. As SDF is periodic, therefore, $m+r*j = \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+s*j+k-1)+1$ based on equation (2.5). Thus, for j iterations later of S, we still have: $i_r = ((n+s*j) \div s) - ((m+r*j) \div r) = (i^{\texttt{S}}+j)-(i^{\texttt{R}}+j) = i^{\texttt{S}}-i^{\texttt{R}}$.

- If R is both upstream and downstream of S then $m = \min\{\min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+k\}, \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+k-1)+1\}$. As SDF is periodic, therefore, $m + r * j = \min\{\min\{m' | \texttt{SDEP}_{\texttt{S}\leftarrow\texttt{R}}(m') \geq n+s*j+k\}, \texttt{SDEP}_{\texttt{R}\leftarrow\texttt{S}}(n+s*j+k-1)+1\}$ based on equation (2.5). Thus, for j iterations later of S, we still have: $i_r = ((n+s*j) \div s) - ((m+r*j) \div r) = (i^{\texttt{S}}+j) - (i^{\texttt{R}}+j) = i^{\texttt{S}} - i^{\texttt{R}}$. ■

Based on Theorem 4, for one relative execution $r^{\texttt{S}}$, we can take an arbitrary $i^{\texttt{S}}$ that is large enough to find the absolute m^{th} execution of R from the absolute n^{th} execution of S where $n = i^{\texttt{S}} * s + r^{\texttt{S}}$ using the techniques in Section 5.3. After finding m, we calculate $i^{\texttt{R}}$ and $r^{\texttt{R}}$ as shown above. Subsequently, we find i_r and e_r.

5.5 Scaling the COSDF Schedulability Checking Process

The basic schedulability checking method presented in the previous sections may not scale well for large RDGs because the zero-cycle detection algorithm by Iwano and Steiglitz [55] is of complexity $\texttt{O}(|V|^3)$ in all cases (due to three nested `for` loops of $|V|$ iterations each). For example, for the MPEG2 decoder benchmark [35], the RDG has around 4 million vertices and the zero-cycle detection algorithm would take 400,000 years to finish [7]. This running time makes the checking process hardly have any practical usages for verifying large programs. However, the method in the previous sections still serves as the basic reasoning process to find a solution. We will present two optimization techniques to make the checking process more feasible.

Pruning RDGs

In this section, we will present a graph pruning method that significantly reduces sizes of RDGs.

Lemma 2 *Zero-cycles in RDGs must contain vertices of actors involved in control communication, called control vertices.*

Proof: Circular dependencies happen when additional constraints by CMs impose on basic SDF schedules. Therefore, cycles of dependencies must contain control vertices. As a result, the corresponding zero-cycles in RDGs must contain control vertices. ■

[7] We predict this running time based on the numbers of vertices and the running times of other smaller benchmarks.

The above lemma implies that we only need to construct pruned RDGs and/or pruned DGs of control vertices. Concretely, we only create control vertices for RDGs and *directly* derive data dependency edges between the control vertices using the `SDEP` function. For example, in Figure 5.6, the non-control vertices C_1 and C_2 can be removed. We draw direct edges between E_1 to A_1, A_2 and E_3 to A_3.

Theorem 5 *A cycle of dependencies in a DG corresponds to a cycle in the corresponding pruned DG.*

Proof: Lemma 2 shows that a cycle of dependencies must contain control vertices. Furthermore, as non-control vertices only serve as proxies to carry data dependencies between control vertices in DGs. Because the data dependency edges between control vertices in pruned DGs can be derived directly using the `SDEP` function, a cycle $\mathcal{C}$ of dependencies in a DG corresponds to the cycle in the corresponding pruned DG constructed by taking only the control vertices of $\mathcal{C}$. ■

The above graph pruning technique significantly reduces the RDG of the MPEG2 decoder benchmark to 25 thousands vertices from 4 million vertices of the original RDG and it would take the zero-cycle detection algorithm around 40 days[8] to finish. This is still rather expensive as programmers would have to wait for days to know whether their programs are valid or not.

Augmenting with Negative-Zero Cycle Detection

To make the schedulability checking process more practical, we need an algorithm that performs on average much faster than the used zero-cycle detection algorithm. The zero-cycle detection algorithm is more expensive on average than the negative cycle detection algorithm [29]. Now, we prove that a DG has cycles of dependencies when the corresponding RDG has negative or zero cycles.

Theorem 6 *A negative cycle in a RDG implies a cycle of dependencies in the corresponding DG.*

Proof: Suppose that a negative cycle in a RDG contains a vertex X_i. The negative cycle has length $l < 0$. There exists a corresponding path in the corresponding DG from X_i^n to X_i^m where $n - m = l$ by Lemma 1. As a result, $X_i^m \prec X_i^n$. Furthermore, by sequential dependency, $X_i^n \prec X_i^m$ because $n - m = l < 0$. Summing up, we have a cycle of dependencies $X_i^m \prec X_i^n \prec X_i^m$. ■

Note that, the above theorem does not conflict with Theorem 3, rather, it presents a way to *early detect* a cycle of dependencies when there exists a negative cycle. Combining both Theorems 3 and 6, we arrive at:

[8]The checking algorithm does not terminate after one day of running. We predict this running time based on the benchmark graph size and running times of other benchmarks.

- If a RDG has a negative or zero cycle, then the corresponding DG has a cycle of dependencies
- If there exists cycle of dependencies in the DG, then we will eventually find a zero cycle in the corresponding RDG.

These results enable using more efficient algorithms that detect both negative and zero cycles. We modify the Tarjan's negative cycle detection algorithm [29] to check for *both* negative and zero cycles. The algorithm has $O(|E||V|)$ worst-case complexity, but in practice, it usually performs much better than the zero-cycle detection. For the MPEG2 decoder benchmark, the algorithm finishes in 58 seconds in comparison with the projected 40-day running time of the zero-cycle detection algorithm on the same processor.

5.6 Soundness and Completeness of the Schedulability Checking Method

In this section, we prove that our checking method is sound and complete according to the definitions in [101].

Theorem 7 *The schedulability checking method by either the zero-cycle detection algorithm or the negative-zero cycle detection algorithm is sound and complete.*

Proof: The method is sound because Theorem 3 says if there exists a cycle of dependencies in a DG, then we will eventually find a zero cycle in the corresponding RDG by either the zero-cycle detection algorithm or the negative-zero cycle detection algorithm.

The method is complete because, based on Theorems 6 and 3, if either the zero-cycle detection algorithm or the negative-zero cycle detection algorithm detects a negative or a zero cycle in a RDG, then the corresponding DG must have a cycle of dependencies. ■

5.7 Scheduling and Parallelism Study

The schedulability checking method presented in the previous sections only determines whether a stream graph is schedulable. The auto-discovery scheduling method implemented in the StreamIt compiler for programs with CMs can find execution orders for stream programs. The scheduling method lets actors interact autonomously with each other to discover schedules, e.g., an actor can start executing whenever it has enough data, otherwise, it will wait. As a consequence, this method hinders the optimizability by exploiting the stationary property of SDF programs. In this section, we use RDGs to derive static schedules and study parallelism for SDF programs with or without CMs.

Static Scheduling

We derive static schedules for stream programs by applying a special topological sort on RDGs. The DGs we constructed for stream programs are infinite, therefore, normal topological sorting algorithms will never terminate. We employ the topological sort method for acyclic periodic graphs described in Section 3.8 of Kodialam's PhD thesis [66]. For an 1-dimensional acyclic periodic graph G^∞, we can calculate a value $A(v^p)$ for each vertex v^p in the periodic graph such that if there is a dependency constraint $v^p \prec u^l$ in the corresponding RDG, then $A(u^l) < A(v^p)$. The calculation is done first by constructing a RDG $G = (V, E, T)$ from G^∞ and then by solving the following linear program:

$$\begin{aligned} &\min_{(u,v)\in E} \sigma_{uv} \\ \pi_v - \pi_u + \gamma T_{uv} &\geq 0 \quad \forall (u,v) \in E \\ \pi_v - \pi_u + \gamma T_{uv} + \sigma_{uv} &\geq 1 \quad \forall (u,v) \in E \\ \sigma_{uv} &\geq 0 \quad \forall (u,v) \in E \end{aligned}$$

where T_{uv} is the weight of edge $(u, v) \in E$. The above linear program has a unique optimal solution for an acyclic periodic graph [66], let us call it $(\sigma^*, \pi^*, \gamma^*)$. Then the value assignment procedure for each vertex v^p is as follows:

$$A(v^p) = \pi_v^* - \gamma^* p \quad \forall v \in V \tag{5.8}$$

With this execution ordering method, we can derive steady state static execution orders of actors within one iteration of an entire stream graph for compiler optimizations.

Note that, we can apply the above topological sorting technique to pruned RDGs to lower running time. Topological sorts on pruned RDGs will only derive sketches of execution orders between control vertices. From the orders of the pivotal control vertices, we can subsequently derive execution orders for other vertices using data dependency constraints based on the pull schedule in Section 2.3. However, a detailed scheduling technique is beyond the scope of this chapter and not difficult to figure out.

Initialization Schedule

Note that an initialization schedule has fewer constraints than its subsequent steady-state sequence. In other words, an initialization schedule only contains a subset of dependency constraints of its subsequent steady-state schedule. In addition, note that, any execution with an iteration index smaller than 1 needs not be considered because they do not exist. Consequently, removing those execution vertices does not make the scheduling problem unschedulable because we simply remove constraints from the scheduling problem. This effectively reduces the number of dependency constraints. As a result, an initialization schedule can be derived by plugging in $n = 1, 2, 3, \ldots$ until finding a steady-state sequence that does not contain any execution whose iteration index is smaller than 1. Note that we

know the upper-bound on how far we need to unroll for the initialization based on repeated steady-state sequences. For the example in Section 5.8, when $n = 1$ we have a sequence: $(A_1^1) \prec (A_2^1, B_1^1) \prec (B_2^1, C_1^1) \prec (A_3^1)$. When $n = 2$ all the iteration indices are greater than 0, as a result, we can stop deriving the initialization schedule. Now, we can repeat the steady-state sequence.

Parallelism Study

The method in the previous section not only presents a way to find static schedules for stream programs with CMs but also helps study stream programs parallelism. In [97], Roychowdhury and Kailath prove that if $A(u^l) = A(v^p)$, then executions u^l and v^p can run in parallel. Thereby, the above scheduling technique presents a formal way to study parallelism of stream programs.

5.8 Experiments

We implemented the above algorithms in the StreamIt compiler, written in Java, to both check for circular dependencies and to find static schedules. We run our evaluations on Intel(R) Xeon(R) CPU E5450 3.00GHz.

Scalability of the Schedulability Checking Methods

We discussed the scalability of the schedulability checking methods in Section 5.5 for the MPEG2 decoder benchmarks. We will evaluate the scalability of the methods for 5 available CMG benchmarks: MPEG2 encoder/decoder, Frequency Hopping Radio (FHR), MP3 Latency and Mosaic.

Table 5.1 shows that the RDG pruning technique significantly reduces the numbers of vertices and edges in RDGs. The original RDGs of very large sizes are reduced to the pruned RDGs with reasonable sizes. For the MPEG2 encoder benchmark, the RDG's size is substantially reduced by around 600,000 times by the graph pruning algorithm. This indicates the number of control vertices is actually very small in comparison with the number of non-control vertices. Constructing original RDGs (without pruning) for MPEG2 encoder/decoder, MP3Latency, Mosaic benchmarks makes our computer run out of memory because the graph sizes are too large[9].

We then run the two cycle detection algorithms, the zero-cycle detection in Section 5.3 and the augmented negative-zero cycle detection in Section 5.5, on the pruned RDGs. Table 5.1 also demonstrates the practicability of the augmented negative-zero cycle detection algorithm. The augmented algorithm finishes within one minute for all the benchmarks.

[9]The number of vertices in one original RDG is obtained by summing all the numbers of actor executions in one iteration of the respective stream graph. These numbers are obtained by the SDF scheduling theory.

While for the MPEG2 decoder, the baseline zero-cycle detection does not terminate after 24 hours.

Table 5.1: Experiments

	Graph size evaluation						Running time evaluation (in ms)		
	# of vertices			# of edges				Cycle detection time	
Benchmark	Original	Pruned	Reduction	Original	Pruned	Reduction	Scheduling	Zero-Cycle	Negative-Zero-Cycle
MPEG2 encoder	58133280	90	645925.3x	- -	210	- -	5461	39184	32742
MPEG2 decoder	4088367	25752	158.8x	- -	84198	- -	943604	- -	58080
FHR	1665	516	3.2x	4868	524	9.3x	834	141	1257
MP3 Latency	7252	1218	5.9x	- -	1218	- -	2196	13041	241
Mosaic	7753576	20	387678.8x	- -	88	- -	929	6994	8098

Static Scheduling

After checking whether the benchmarks are schedulable, we run the static scheduling algorithm on the pruned RDGs, and obtain the results as in Table 5.1. We use the CPLEX (`http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`) solver. For the MPEG2 decoder benchmark, it takes around 0.64 second to schedule its pruned RDGs with 7 CLEX threads. The scheduling for the other benchmarks finishes almost in no time.

We take the stream graph in Figure 2.1 as an example with CMG latency from `E` to `A` is 3 and from `B` to `D` is -2. With this configuration, the stream graph does not have any cycle of dependencies. We apply the topological sorting algorithm by solving the respective linear program and obtain $\gamma^* = 5$ and $\pi^*_{A_1} = 9, \pi^*_{B_3} = 4, \pi^*_{A_2} = 8, \pi^*_{B_1} = 8, \pi^*_{C_2} = 3, \pi^*_{D_1} = 2, \pi^*_{B_2} = 7, \pi^*_{C_1} = 7, \pi^*_{E_1} = 1, \pi^*_{D_2} = 1, \pi^*_{E_2} = 0, \pi^*_{A_3} = 5$. As a result, a valid execution order for the stream program is: $(A_1^n, B_3^{n-1}) \prec (A_2^n, B_1^n, C_2^{n-1}) \prec (D_1^{n-1}, B_2^n, C_1^n) \prec (E_1^{n-1}, D_2^{n-1}) \prec (E_2^{n-1}, A_3^n)$.

5.9 Expressiveness of CMG

Readers may wonder about the usefulness of CMG. Although CMG is used to describe COs, its implication is broader than that. In this section, we will discuss and summarize a number of advantages of CMG.

Verifiability, Readability, Maintainability, and Performance Improvement of CMG

As discussed in Section 5.3, separating sporadic low-bandwidth control communication from regular high-bandwidth data communication helps reduce wasted communication bandwidths. It also improves the readability, maintainability, and backward compatibility of stream programs by avoiding mixing up data and control processing codes. For example

without CMG, to change a CM latency, programmers have to dive into program code and modify code segments related to data token processing. As a consequence, maintaining programs becomes more difficult without CMG. Furthermore, it is also difficult to check for the schedulability problem because compilers have to analyze complicated tokens processing code just to infer inherent information such as message latency and which actors will process which messages. Applying the RDG pruning technique is even more complicated.

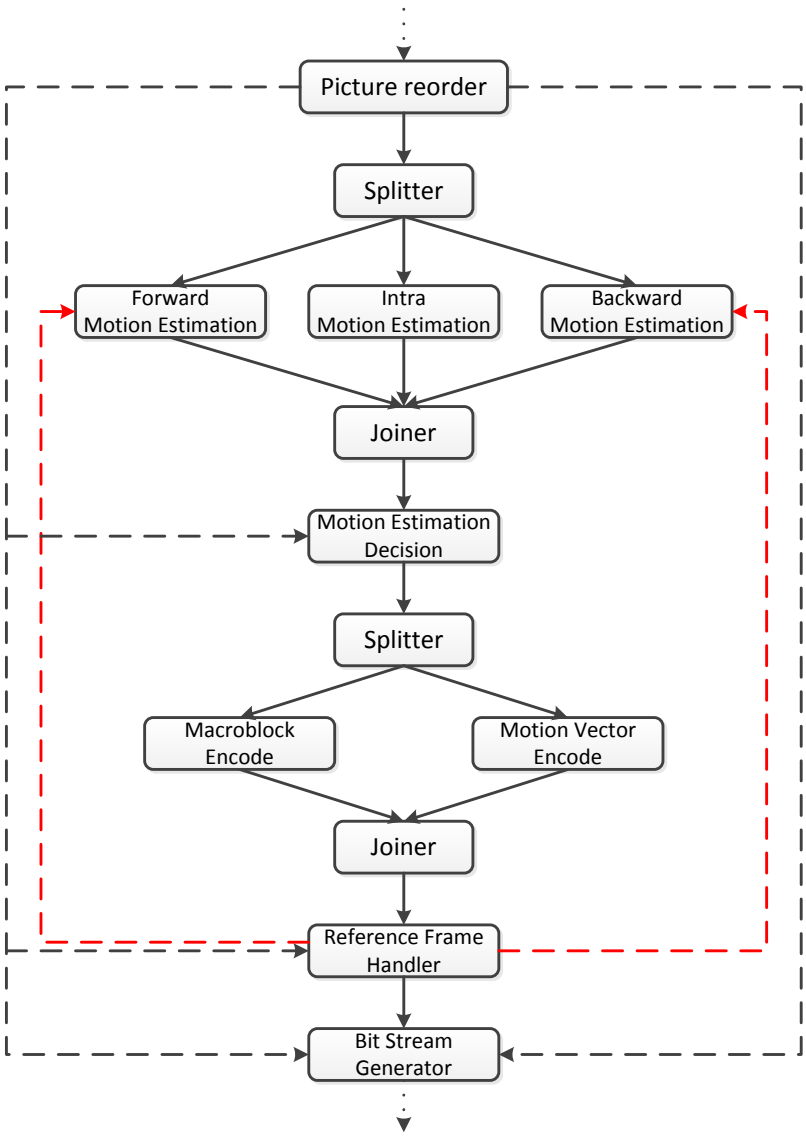


Figure 5.8: MPEG2 encoder stream graph.

Feedback Loops Reasoning

We will discuss about how CMG can help improve programming productivity. Beside the advantage of using CMG for expressing control communication, CMG can also help programmers to reason about data dependencies in complicated stream program structures. For example, when feedback loops cross multiple actors, it is difficult for programmers to determine which feedback tokens are relevant to certain input data tokens. For example, when implementing his MPEG2 benchmarks [35], Drake uses CMs, the red dash arrows from the actor `Reference Frame Handler` to actors `Forward Motion Estimation` and `Backward Motion Estimation` in Figure 5.8, for feedback loops that help multiplex between feedback tokens from actors `Forward Motion Estimation` and `Backward Motion Estimation` and the input data tokens from the upstream `Splitter` actor. The `SDEP` function helps ease that multiplexing reasoning process.

Optimizing Stream Programs with CMG

In Section 8.4 of [35], Drake shows how to use CMG to implement programmable `Split/Join` actors to avoid the overhead of processing unused data in the MPEG2 encoder/decoder benchmarks. For example in Figure 5.8, the result from either actor `Forward Motion Estimation` or actor `Backward Motion Estimation` is used, which is decided by actor `Motion Estimation Decision`. However, the regularity property of SDF semantics makes the current `Split/Join` actors regularly dispatch/retrieve data tokens to/from other actors. Actors `Split/Join` are not capable of distributing/gathering dynamically and sporadically. As a consequence, we waste the computation done by either actor `Forward Motion Estimation` or actor `Backward Motion Estimation`. Drake in [35] suggests that `Split/Join` actors that are capable of dynamically dispatching/retrieving data and synchronized using CMs can help programmers implement such an optimization. Note that this suggested extension creates additional overlapping of CMs in the MPEG2 benchmarks.

5.10 Related Work

Thies et al. [115] introduce CMG as a mechanism to integrate COs into SDF. They present an analysis that computes processing time of CMs. However, this analysis is applicable to only non-overlapping CMs. Consequently, SDF graphs with overlapping CMs cannot utilize this analysis. We address this limitation with our dependency analysis method applicable to any SDF graph with CMs. Furthermore, we show that it is possible to compute schedules for the SDF graphs with integrated COs. This problem is not well-defined in the work by Thies et al. [115].

As an extension of CMG in the StreamIt compiler, our work is based on SDF/CSDF [70, 18] semantics. We also adopt several results from the RDG work pioneered by Karp, Miller and Winograd [61]. Darte [33] summarizes work related to the pioneering work by Karp, Miller, and Winograd [61]. The loop compilation techniques in [6, 3, 62] also derive from

the work by Karp, Miller, and Winograd. Our graph pruning technique is related to the quotient graph theory [12].

Zhou and Lee [124] tackle a circular dependency analysis problem using causality interfaces. For the SDF case, they do not account for CMs in their SDF models. Moreover, we also support dependency analysis for SDF with overlapping CMs.

Horwitz et al. [52] propose a method for interprocedure program slicing by constructing DGs between program statements with data and control dependency edges. The graph construction method is similar to our method in that they construct DGs of statements and use the graphs to find dependency between interprocedure statements.

The deadlock analysis method for communicating processes by Brook and Roscoe [22] characterizes the properties and structures of networks of communicating processes that can cause deadlocks. For example, the work can answer which kind of communication can cause the dining philosophers problem. However, the method only works for some structures of networks of processes.

There are several algorithms on deadlock detection in distributed systems [26]. However, the algorithms proposed mainly focus on detecting deadlocks when they happens rather than on deadlock avoidance and static deadlock analysis.

Wang et al. [118] propose a method that helps avoid potential deadlocks in multithreaded programs sharing resources. The method translates control flow graphs into Petri nets. Petri net theories are used to synthesize control logic code to avoid potential deadlocks. This bears some similarity to our construction of DGs to detect deadlocks.

5.11 Conclusion

In this chapter, we give a formal definition of control communication latency conceived in [115]. We then present a method for checking for invalid sets of specifications when SDF models are extended with the control semantics by exploiting its periodic property. The application of this work is not only limited in current StreamIt benchmarks but also applicable to other design environments supporting SDF streaming semantics such as Ptolemy(`http://ptolemy.eecs.berkeley.edu`) and LabVIEW(`http://www.ni.com/labview/`). Users in such environments can create arbitrary streaming models and the frameworks need to quickly validate the models.

The other implication of this work is to study parallelism of stream programs formally through our formulated RDGs for stream programs. We have implemented the method in the StreamIt compiler. Furthermore, in the StreamIt compiler's backend, actors involved in CMG are not clustered because fused actors cannot be identified and fusing actors can cause false dependencies, therefore, generated code may be not efficient. A modular code generation method, such as the one proposed by Lublinerman et al. in [75], can avoid the problem of false dependencies caused by fusing actors.

COs' semantics are derived for the SDF model of computation, however, it is straightforward to apply the techniques in this chapter to the CSDF model of computation with almost

no required modification. It would be interesting to extend the work to the MDSDF model of computation for image processing applications as in [99] because MDSDF is natural to express image and video processing applications.

Chapter 6

Compositionality in Synchronous Data Flow

Hierarchical SDF models are not compositional: a composite SDF actor cannot be represented as an atomic SDF actor without loss of information that can lead to rate inconsistency or deadlock. Motivated by the need for incremental and modular code generation from hierarchical SDF models, we introduce in this chapter DSSF profiles. DSSF (Deterministic SDF with Shared FIFOs) forms a compositional abstraction of composite actors that can be used for modular compilation. We provide algorithms for automatic synthesis of non-monolithic DSSF profiles of composite actors given DSSF profiles of their sub-actors. We show how different tradeoffs can be explored when synthesizing such profiles, in terms of compactness (keeping the size of the generated DSSF profile small) versus reusability (maintaining necessary information to preserve rate consistency and deadlock-absence) as well as algorithmic complexity. We show that our method guarantees maximal reusability and report on a prototype implementation.

6.1 Introduction

Programming languages have been constantly evolving over the years, from assembly, to structural programming, to object-oriented programming, etc. Common to this evolution is the fact that new programming models provide mechanisms and notions that are more *abstract*, that is, remote from the actual implementation, but better suited to the programmer's intuition. Raising the level of abstraction results in undeniable benefits in productivity. But it is more than just building systems faster or cheaper. It also allows to create systems that could not have been conceived otherwise, simply because of too high complexity.

Modeling languages with built-in concepts of concurrency, time, I/O interaction, and so on, are particularly suitable in the domain of embedded systems. Indeed, languages such as Simulink, UML or SystemC, and corresponding tools, are particularly popular in this domain, for various applications. The tools provide mostly modeling and simulation,

but often also code generation and static analysis or verification capabilities, which are increasingly important in an industrial setting. We believe that this tendency will continue, to the point where modeling languages of today will become the programming languages of tomorrow, at least in the embedded software domain.

A widespread model of computation in this domain is Synchronous (or Static) Data Flow (SDF) [70]. SDF is particularly well-suited for signal processing and multimedia applications and has been extensively studied over the years (e.g., see [13, 108]). Recently, languages based on SDF, such as StreamIt [114], have also been applied to multicore programming.

In this chapter we consider *hierarchical* SDF models, where an SDF graph can be *encapsulated* into a *composite* SDF actor. The latter can then be connected with other SDF actors, further encapsulated, and so on, to form a hierarchy of SDF actors of arbitrary depth. This is essential for *compositional modeling*, which allows to design systems in a modular, scalable way, enhancing readability and allowing to master complexity in order to build larger designs. Hierarchical SDF models are part of a number of modeling environments, including the Ptolemy II framework [36]. A hierarchical SDF model is shown in Figure 6.1.

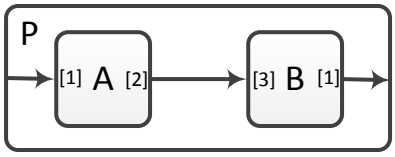


Figure 6.1: Example of a hierarchical SDF graph.

The problem we solve in this chapter is *modular code generation* for hierarchical SDF models. Modular means that code is generated for a given composite SDF actor `P` *independently from context*, that is, independently from which graphs `P` is going to be used in. Moreover, once code is generated for `P`, then `P` can be seen as an *atomic* (non-composite) actor, that is, a "black box" without access to its internal structure. Modular code generation is analogous to separate compilation, which is available in most standard programming languages: the fact that one does not need to compile an entire program in one shot, but can compile files, classes, or other units, separately, and then combine them (e.g., by *linking*) to a single executable. This is obviously a key capability for a number of reasons, ranging from incremental compilation (compiling only the parts of a large program that have changed), to dealing with IP (intellectual property) concerns (having access to object code only and not to source code). We want to do the same for SDF models. Moreover, in the context of a system like Ptolemy II, in addition to the benefits mentioned above, modular code generation is also useful for speeding-up simulation: replacing entire sub-trees of a large hierarchical model by a single actor for which code has been automatically generated and pre-compiled, removes the overhead of executing all actors in the sub-tree individually.

Modular code generation is not trivial for hierarchical SDF models because they are not compositional. Let us try to give the intuition of this fact here through an example. A

more detailed description is provided in the sections that follow. Consider the left-most graph of Figure 6.2, where the composite actor P of Figure 6.1 is used. This left-most graph should be equivalent to the right-most graph of Figure 6.2, where P has been replaced by its internal contents (i.e., the right-most graph is the "flattened" version of the left-most one). Observe that the right-most graph has no deadlock: indeed, actors A,B,C can fire (execute)[1] infinitely often according to the periodic schedule $(\mathtt{A}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{A}, \mathtt{B})^{\omega}$. Now, suppose we treat P as an atomic actor that consumes 3 tokens and produces 2 tokens every time it fires: this makes sense, since it corresponds to a complete iteration of its internal SDF graph, namely, $(\mathtt{A}, \mathtt{A}, \mathtt{B}, \mathtt{A}, \mathtt{B})$. We then find that the left-most graph has a deadlock: P cannot fire because it needs 3 tokens but only 2 are initially available in the queue from C to P; C cannot fire either because it needs 2 tokens but only 1 is initially available in the queue from P to C.

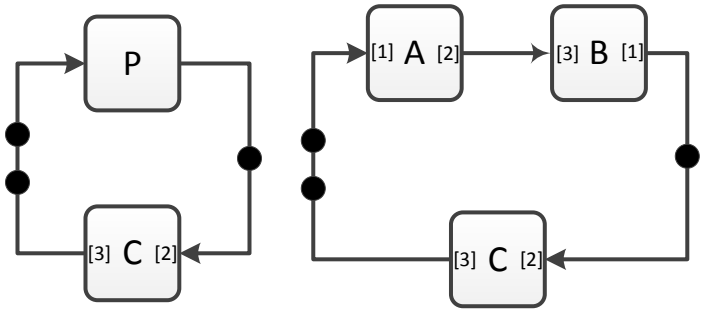


Figure 6.2: Left: using the composite actor P of Figure 6.1 in an SDF graph with feedback and initial tokens. Right: the same graph after flattening P.

The above example illustrates that composite SDF actors cannot be represented by atomic SDF actors without loss of information that can lead to deadlocks. Even in the case of acyclic SDF graphs, problems may still arise due to rate inconsistencies (see Figure 6.6 and related discussion). Compositionality problems also arise in simpler hierarchical models such as *synchronous block diagrams* (SBDs) which (in the absence of triggers) can be seen as the subclass of *homogeneous* SDF where token rates are all equal [77, 76, 75]. Our work extends the ideas of modular code generation for SBDs introduced in the above works. In particular, we borrow their notion of *profile* which characterizes a given actor. Modular code generation then essentially becomes a *profile synthesis* problem: how to synthesize a profile for composite actors, based on the profiles of its internal actors.

In SBDs, profiles are essentially DAGs (directed acyclic graphs) that capture the dependencies between inputs and outputs of a block, at the same synchronous round. In general, not all outputs depend on all inputs, which allows feedback loops with unambiguous semantics to be built. For instance, in a *unit delay* block the output does not depend on the input at the same clock cycle, therefore this block "breaks" dependency cycles when used in feedback loops.

[1] We use the word "fire" instead of "execute in this chapter so that we can illustrate the problem better.

The question is, what is the right model for profiles of SDF graphs. We answer this question in this chapter. For SDF graphs, profiles turn out to be more interesting than simple DAGs. SDF profiles are essentially SDF graphs themselves, but with the ability to associate multiple producers and/or consumers with a single FIFO queue. Sharing queues among different actors generally results in non-deterministic models, however, in our case, we can guarantee that actors that share queues are always fired in a deterministic order. We call this model *deterministic SDF with shared FIFOs* (DSSF). DSSF allows, in particular, to decompose the firing of a composite actor into an arbitrary number of *firing functions* that may consume tokens from the same input port or produce tokens to the same output port. Having multiple firing functions allows to decouple firings of different internal actors of the composite actor, so that deadlocks are avoided when the composite actor is embedded in a given context. Our method guarantees *maximal reusability* [77], i.e., the absence of deadlock in any context where the corresponding "flat" (non-hierarchical) SDF graph is deadlock-free, as well as consistency in any context where the flat SDF graph is consistent.

For example, two possible profiles for the composite actor P of Figure 6.1 are shown in Figure 6.3. The left-most one has a single firing function that corresponds to the internal sequence of firings $(\mathtt{A}, \mathtt{A}, \mathtt{B}, \mathtt{A}, \mathtt{B})$, which is problematic as explained above. The right-most profile is a DSSF graph where the above sequence is split into two firing functions: $\mathtt{P}.f_1$, corresponding to $(\mathtt{A}, \mathtt{A}, \mathtt{B})$, and $\mathtt{P}.f_2$, corresponding to $(\mathtt{A}, \mathtt{B})$. This profile is maximally-reusable, and also optimal in the sense that no less than two firing functions can achieve maximal reusability.

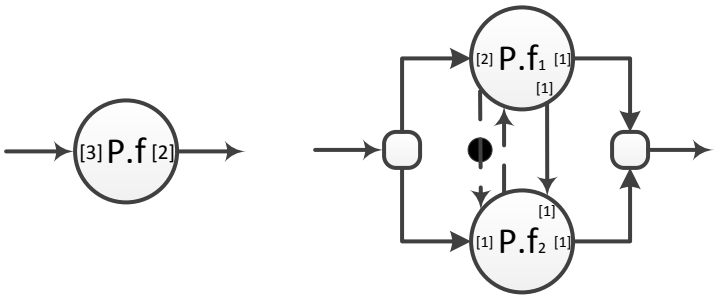


Figure 6.3: Two DSSF graphs that are also profiles for the composite actor P of Figure 6.1.

We show how to perform profile synthesis for SDF graphs automatically. This means to synthesize for a given composite actor a profile, in the form of a DSSF graph, given the profiles of its internal actors (also DSSF graphs). This process involves multiple steps, among which are the standard *rate analysis* and *deadlock detection* procedures used to check whether a given SDF graph can be executed infinitely often without deadlock and with bounded queues [70]. In addition to these steps, SDF profile synthesis involves *unfolding* a DSSF graph (i.e., replicating actors in the graph according to their relative rates produced by rate analysis) to produce a DAG that captures the dependencies between the different consumptions and productions of tokens *at the same port*.

Reducing the DSSF graph to a DAG is interesting because it allows to apply for our purposes the idea of *DAG clustering* proposed originally for SBDs [77, 75]. As in the SBD case, we use DAG clustering in order to group together firing functions of internal actors and synthesize a small (hopefully minimal) number of firing functions for the composite actor. These determine precisely the profile of the latter. Keeping the number of firing functions small is essential, because it results in further compositions of the actor being more efficient, thus allowing the process to scale to arbitrary levels of hierarchy.

As shown by [77, 75], there exist different ways to perform DAG clustering, that achieve different tradeoffs, in particular in terms of number of clusters produced vs. reusability of the generated profile. Among the clustering methods proposed for SBDs, of particular interest to us are those that produce *disjoint* clusterings, where clusters do not share nodes. Unfortunately, *optimal disjoint clustering*, that guarantees maximal reusability with a minimal number of clusters, is NP-complete [75]. This motivates us to devise a new clustering algorithm, called *greedy backward disjoint clustering* (GBDC). GBDC guarantees maximal reusability but due to its greedy nature cannot guarantee optimality in terms of number of clusters. On the other hand, GBDC has polynomial complexity.

The rest of this chapter is organized as follows. Section 6.2 discusses related work. Section 6.3 introduces DSSF graphs and SDF as a subclass of DSSF. Section 6.4 reviews analysis methods for SDF graphs. Section 6.5 reviews modular code generation for SBDs which we build upon. Section 6.6 introduces SDF profiles. Section 6.7 describes the profile synthesis procedure. Section 6.8 details DAG clustering, in particular, the GBDC algorithm. Section 6.9 presents a prototype implementation. Section 6.10 presents the conclusions and discusses future work.

6.2 Related Work

Dataflow models of computation have been extensively studied in the literature. Dataflow models with deterministic actors, such as Kahn Process Networks [58] and their various subclasses, including SDF, are compositional at the semantic level. Indeed, actors can be given semantics as continuous functions on streams, and such functions are closed by composition. (Interestingly, it is much harder to derive a compositional theory of *non-deterministic* dataflow, e.g., see [21, 57, 110].) Our work is at a different, non-semantical level, since we mainly focus on finite representations of the behavior of networks at their interfaces, in particular of the dependencies between inputs and output. We also take a "black-box" view of atomic actors, assuming their internal semantics (e.g., which function they compute) are unknown and unimportant for our purpose of code generation. Finally, we only deal with the particular subclass of SDF models.

Despite extensive work on code generation from SDF models and especially scheduling (e.g., see [13, 108], there is little existing work that addresses compositional representations and modular code generation for such models. [41] proposes abstraction methods that reduce the size of SDF graphs, thus facilitating throughput and latency analysis. His goal is to have

a conservative abstraction in terms of these performance metrics, whereas our goal here is to preserve input-output dependencies to avoid deadlocks during further composition.

Non-compositionality of SDF due to potential deadlocks has been observed in earlier works such as [95], where the objective is to schedule SDF graphs on multiple processors. This is done by partitioning the SDF graph into multiple sub-graphs, each of which is scheduled on a single processor. This partitioning (also called *clustering*, but different from DAG clustering that we use in this chapter, see below) may result in deadlocks, and the so-called "SDF composition theorem" given by [95] provides a sufficient condition so that no deadlock is introduced.

More recently, [37] also identify the problem of non-compositionality and propose *Cluster Finite State Machines* (CFSMs) as a representation of composite SDF. They show how to compute a CFSM for a composite SDF actor that contains standard, atomic, SDF sub-actors, however, they do not show how a CFSM can be computed when the sub-actors are themselves represented as CFSMs. This indicates that this approach may not generalize to more than one level of hierarchy. Our approach works for arbitrary depths of hierarchy.

Another difference between the above work and ours is on the representation models, namely, CFSM vs. DSSF. CFSM is a state-machine model, where transitions are annotated with guards checking whether a sufficient number of tokens is available in certain input queues. DSSF, on the other hand, is a data flow model, only slightly more general than SDF. This allows to re-use many of the techniques developed for standard SDF graphs, for instance, rate analysis and deadlock detection, with minimal adaptation.

The same remark applies to other automata-based formalisms, such as I/O automata [80], interface automata [34], and so on. Such formalisms could perhaps be used to represent consumption and production actions of SDF graphs, resulting in compositional representations. These would be at a much lower level than DSSF, however, and for this reason would not admit SDF techniques such as rate analysis, which are more "symbolic".

To the extent that we propose DSSF profiles as interfaces for composite SDF graphs, our work is related to so-called *component-based design* and *interface theories* [5]. Similarly to that line of research, we propose methods to synthesize interfaces for compositions of components, given interfaces for these components. We do not, however, include notions of refinement in our work. We are also not concerned with how to specify the "glue code" between components, as is done in *connector algebras* [9, 19]. Indeed, in our case, there is only one type of connections, namely, conceptually unbounded FIFOs, defined by the SDF semantics. Moreover, connections of components are themselves specified in the SDF graphs of composite actors, and are given as an input to the profile synthesis algorithm. Finally, we are not concerned with issues of timeliness or distribution, as in the work of [67].

Finally, we should emphasize that our DAG clustering algorithms solve a different problem than the clustering methods used in [95, 37] and other works in the SDF scheduling literature. Our clustering algorithms operate on plain DAGs, as do the clustering algorithms originally developed for SBDs [77, 75]. On the other hand, [37, 95] perform clustering directly at the SDF level, by grouping SDF actors and replacing them by a single SDF actor (e.g., see Figure 4 of [37]). This, in our terms, corresponds to monolithic clustering, which

is not compositional.

6.3 Hierarchical DSSF and SDF Graphs

Deterministic SDF with shared FIFOs, or DSSF, is an extension of SDF in the sense that, whereas shared FIFOs (first-in, first-out queues) are explicitly prohibited in SDF graphs, they are allowed in DSSF graphs, provided determinism is ensured.

Syntactically, a DSSF graph consists of a set of nodes, called *actors*,[2] a set of FIFO *queues*, a set of *external ports*, and a set of directed *edges*. Each actor has a set of *input ports* (possibly zero) and a set of *output ports* (possibly zero). An edge connects an output port of an actor to the input of a queue, or an output port of a queue to an input port of an actor. Actor ports can be connected to at most one queue.[3] An edge may also connect an external input port of the graph to the input of a queue, or the output of a queue to an external output port of the graph.

Actors are either *atomic* or *composite*. A composite actor `P` encapsulates a DSSF graph, called the *internal* graph of `P`. The input and output ports of `P` are identified with the input and output external ports of its internal graph. Composite actors can themselves be encapsulated in new composite actors, thus forming a hierarchical model of arbitrary depth. A graph is *flat* if it contains only atomic actors, otherwise it is *hierarchical*. A *flattening* process can be applied to turn a hierarchical graph into a flat graph, by removing composite actors and replacing them with their internal graph, while making sure to re-institute any connections that would be otherwise lost.

Each port of an atomic actor has an associated *token rate*, a positive integer number, which specifies how many tokens are consumed from or produced to the port every time the actor fires. Composite actors do not have token rate annotations on their ports. They inherit this information from their internal actors, as we will explain in this chapter.

A queue can be connected to more than one ports, at its input or output. When this occurs we say that the queue is *shared*, otherwise it is *non-shared*. An SDF graph is a DSSF graph where all queues are non-shared. Actors connected to the input of a queue are the *producers* of the queue, and actors connected to its output are its *consumers*. A queue stores *tokens* added by producers and removed by consumers when these actors *fire*. An atomic actor can fire when each of its input ports is connected to a queue that has enough tokens, i.e., more tokens than specified by the token rate of the port. Firing is an atomic action, and consists in removing the appropriate number of tokens from every input queue and adding the appropriate number of tokens to every output queue of the actor. Queues may store a

[2] It is useful to distinguish between actor *types* and actor *instances*. Indeed, an actor can be used in a given graph multiple times. For example, an actor of type *Adder*, that computes the arithmetic sum of its inputs, can be used multiple times in a given graph. In this case, we say that the Adder is *instantiated* multiple times. Each "copy" is an actor *instance*. In the rest of the paper, we often omit to distinguish between type and instance when we refer to an actor, when the meaning is clear from context.

[3] Implicit *fan-in* or *fan-out* is not allowed, however, it can be implemented explicitly, using actors. For example, an actor that consumes an input token and replicates to each of its output ports models fan-out.

number of *initial* tokens. Queues are of unbounded size in principle. In practice, however, we are interested in graphs that can execute forever using bounded queues.

To see why having shared queues generally results in non-deterministic models, consider two producers $\mathtt{A}_1, \mathtt{A}_2$ sharing the same output queue, and a consumer B reading from that queue and producing an external output. Depending on the order of execution of $\mathtt{A}_1$ and $\mathtt{A}_2$, their outputs will be stored in the shared queue in a different order. Therefore, the output of B will also generally differ (note that tokens may carry values).

To guarantee determinism, it suffices to ensure that $\mathtt{A}_1$ and $\mathtt{A}_2$ are always executed in a fixed order. This is the condition we impose on DSSF graphs, namely, that if a queue is shared among a set of producers $\mathtt{A}_1, ..., \mathtt{A}_a$ and a set of consumers $\mathtt{B}_1, ..., \mathtt{B}_b$, then the graph ensures, by means of its connections, a deterministic way of firing $\mathtt{A}_1, ..., \mathtt{A}_a$, as well as a deterministic way of firing $\mathtt{B}_1, ..., \mathtt{B}_b$.

Let us provide some examples of SDF and DSSF graphs. A hierarchical SDF graph is shown in Figure 6.1. P is a composite actor while A and B are atomic actors. Every time it fires, A consumes one token from its single input port and produces two tokens to its single output port. B consumes three tokens and produces one token every time it fires. There are several non-shared queues in this graph: a queue with producer A and consumer B; a queue connecting the input port of P to the input port of its internal actor A; and a queue connecting the output port of internal actor B to the output port of P. Non-shared queues are identified with corresponding directed edges.

Two other SDF graphs are shown in Figure 6.2, which also shows an example of flattening. The left-most graph is hierarchical, since it contains composite actor P. By flattening this graph we obtain the right-most graph. These graphs contain queues with initial tokens, depicted as black dots. The queue connecting the output port of C to the input port of P has two initial tokens. Likewise, there is one initial token in the queue from P to C.

Figure 6.3 shows two more examples of DSSF graphs. Both graphs are flat. Actors in these graphs are drawn as circles instead of squares because, as we shall see in Section 6.6, these graphs are also SDF profiles. External ports are depicted by arrows. The left-most graph is an SDF graph with a single actor $\mathtt{P}.f$. The right-most one is a DSSF graph with two actors, $\mathtt{P}.f_1$ and $\mathtt{P}.f_2$. This graph has two shared queues, depicted as small squares. The two queues are connected to the two external ports of the graph. The graph also contains three non-shared queues, depicted implicitly by the dashed-line and solid-line edges connecting $\mathtt{P}.f_1$ and $\mathtt{P}.f_2$. Dashed-line edges are called *dependency edges* and are distinguished from solid-line edges that are "true" *dataflow edges*. The distinction is made only for reasons of clarity, in order to understand better the way edges are created during profile generation (Section 6.7). Otherwise the distinction plays no role, and dependency edges can be encoded as standard dataflow edges with token production and consumption rates both equal to 1. Notice, first, that the dataflow edge ensures that $\mathtt{P}.f_2$ cannot fire before $\mathtt{P}.f_1$ fires for the first time; and second, that the dependency edge from $\mathtt{P}.f_2$ to $\mathtt{P}.f_1$ ensures that $\mathtt{P}.f_1$ can fire for a second time only *after* the first firing of $\mathtt{P}.f_2$. Together these edges impose a total order on the firing of these two actors, therefore fulfilling the DSSF requirement of determinism.

DSSF graphs can be *open* or *closed*. A graph is closed if all its input ports are connected;

otherwise it is open. The graph of Figure 6.1 is open because the input port of P is not connected. The graphs of Figure 6.3 are also open. The graphs of Figure 6.2 are closed.

6.4 Analysis of SDF Graphs

We review these analysis methods here, because we are going to adapt them and use them for modular code generation (Section 6.7).

Rate Analysis

Rate analysis seeks to determine if the token rates in a given SDF graph are *consistent*: if this is not the case, then the graph cannot be executed infinitely often with bounded queues. We illustrate the analysis in the simple example of Figure 6.1. The reader is referred to [70] or Section 2.2 for the details.

We wish to analyze the internal graph of P, consisting of actors A and B. This is an open graph, and we can ignore the unconnected ports for the rate analysis. Suppose A is fired $r_{\mathtt{A}}$ times for every $r_{\mathtt{B}}$ times that B is fired. Then, in order for the queue between A and B to remain bounded in repeated execution, it has to be the case that:

$$r_{\mathtt{A}} \cdot 2 = r_{\mathtt{B}} \cdot 3$$

that is, the total number of tokens produced by A equals the total number of tokens consumed by B. The above *balance equation* has a non-trivial (i.e., non-zero) solution: $r_{\mathtt{A}} = 3$ and $r_{\mathtt{B}} = 2$. This means that this SDF graph is indeed consistent. In general, for larger and more complex graphs, the same analysis can be performed, which results in solving a system of multiple balance equations. If the system has a non-trivial solution then the graph is consistent, otherwise it is not. At the end of rate analysis, if consistent, a *repetition vector* $(r_1, ..., r_n)$ is produced that specifies the number r_i of times that every actor $\mathtt{A}_i$ in the graph fires with respect to other actors. This vector is used in the subsequent step of deadlock analysis. Note that for disconnected graphs, the individual parts each have their own repetition vector and any linear combination of multiples of these repetition vectors is a PASS of the whole graph.

Deadlock Analysis

Having a consistent graph is a necessary, but not sufficient condition for infinite execution: the graph might still contain *deadlocks* that arise because of absence of enough initial tokens. Deadlock analysis ensures that this is not the case. An SDF graph is deadlock free if and only if every actor A can fire $r_{\mathtt{A}}$ times, where $r_{\mathtt{A}}$ is the repetition value for A in the repetition vector (i.e., it has a PASS [70]). The method works as follows. For every queue e_i in the SDF graph, an integer counter b_i is maintained, representing the number of tokens in e_i. Counter b_i is initialized to the number of initial tokens present in e_i (zero if no such tokens are present). For every actor A in the SDF graph, an integer counter $c_{\mathtt{A}}$ is maintained, representing the

number of remaining times that A should fire to complete the PASS. Counter $c_{\mathtt{A}}$ is initialized to $r_{\mathtt{A}}$. A tuple consisting of all above counters is called a *configuration* v. A transition from a configuration v to a new configuration v' happens by firing an actor A, provided A is *enabled* at v, i.e., all its input queues have enough tokens, and provided that $c_{\mathtt{A}} > 0$. Then, the queue counters are updated, and counter $c_{\mathtt{A}}$ is decremented by 1. If a configuration is reached where all actor counters are 0, there is no deadlock, otherwise, there is one. Notice that a single path needs to be explored, so this is not a costly method (i.e., not a full-blown reachability analysis). In fact, at most $\Sigma_{i=1}^{n} r_i$ steps are required to complete deadlock analysis, where $(r_1, ..., r_n)$ is the solution to the balance equations.

We illustrate deadlock analysis with an example. Consider the SDF graph shown at the left of Figure 6.2 and suppose P is an atomic actor, with input/output token rates 3 and 2, respectively. Rate analysis then gives $r_{\mathtt{P}} = r_{\mathtt{C}} = 1$. Let the queues from P to C and from C to P be denoted e_1 and e_2, respectively. Deadlock analysis then starts with configuration $v_0 = (c_{\mathtt{P}} = 1, c_{\mathtt{C}} = 1, b_1 = 1, b_2 = 2)$. P is not enabled at v_0 because it needs 3 input tokens but $b_2 = 2$. C is not enabled at v_0 either because it needs 2 input tokens but $b_1 = 1$. Thus v_0 is a deadlock. Now, suppose that instead of 2 initial tokens, queue e_2 had 3 initial tokens. Then, we would have as initial configuration $v_1 = (c_{\mathtt{P}} = 1, c_{\mathtt{C}} = 1, b_1 = 1, b_2 = 3)$. In this case, deadlock analysis can proceed: $v_1 \xrightarrow{\mathtt{P}} (c_{\mathtt{P}} = 0, c_{\mathtt{C}} = 1, b_1 = 3, b_2 = 0) \xrightarrow{\mathtt{C}} (c_{\mathtt{P}} = 0, c_{\mathtt{C}} = 0, b_1 = 1, b_2 = 3)$. Since a configuration is reached where $c_{\mathtt{P}} = c_{\mathtt{C}} = 0$, there is no deadlock.

Transformation of SDF to Homogeneous SDF

A *homogeneous* SDF (HSDF) graph is an SDF graph where all token rates are equal (and without loss in generality, can be assumed to be equal to 1). Any consistent SDF graph can be transformed to an equivalent HSDF graph using a type of an *unfolding* process consisting in replicating each actor in the SDF as many times as specified in the repetition vector [70, 95, 108]. The unfolding process subsequently allows to identify explicitly the input/output dependencies of different productions and consumptions at the same output or input port. Examples of unfolding are presented in Section 6.7, where we adapt the process to our purposes and to the case of DSSF graphs.

6.5 Modular Code Generation Framework

As mentioned in the introduction, our modular code generation framework for SDF builds upon the work of [77, 75]. A fundamental element of the framework is the notion of *profiles*. Every SDF actor has an associated profile. The profile can be seen as an *interface*, or *summary*, that captures the essential information about the actor. Atomic actors have predefined profiles. Profiles of composite actors are synthesized automatically, as shown in Section 6.7.

A profile contains, among other things, a set of *firing functions*, that, together, implement the firing of an actor. In the simple case, an actor may have a single firing function. For

example, actors `A, B` of Figure 6.1 may each have a single firing function

```
A.fire(input x[1]) output (y[2]);
B.fire(input x[3]) output (y[1]);
```

The above signatures specify that `A.fire` takes as input 1 token at input port x and produces as output 2 tokens at output port y, and similarly for `B.fire`. In general, however, an actor may have more than one firing function in its profile. This is necessary in order to avoid *monolithic* code, and instead produce code that achieves *maximal reusability*, as is explained in Section 6.7.

The *implementation* of a profile contains, among other things, the implementation of each of the firing functions listed in the profile as a sequential program in a language such as C++ or Java. We will show how to automatically generate such implementations of SDF profiles in Section 6.7.

Modular code generation is then the following process:

- given a composite actor `P`, its internal graph, and profiles for every internal actor of `P`,
- synthesize automatically a profile for `P` and an implementation of this profile.

Note that a given actor may have multiple profiles, each achieving different tradeoffs, for instance, in terms of *compactness* of the profile and *reusability* (ability to use the profile in as many contexts as possible). We illustrate such tradeoffs in the sequel.

6.6 SDF Profiles

We will use a special class of DSSF graphs to represent profiles of SDF actors, called *SDF profiles*. An SDF profile is a DSSF graph such that all its shared queues are connected to its external ports. Moreover, all connections between actors of the graph are such that the number of tokens produced and consumed at each firing by the source and destination actors are equal: this implies that connected actors fire with equal rates. The shared queues of SDF profiles are called *external*, because they are connected to external ports. This is to distinguish them from *internal* shared queues that may arise in other types of DSSF graphs that we use in this chapter, in particular, in so-called *internal profiles graphs* (see Section 6.7).

Two SDF profiles are shown in Figure 6.3. They are two possible profiles for the composite actor `P` of Figure 6.1. We will see how these two profiles can be synthesized automatically in Section 6.7. We will also see that these two profiles have different properties. In particular, they represent different Pareto points in the compactness vs. reusability tradeoff (Section 6.7).

The actors of an SDF profile represent firing functions. The left-most profile of Figure 6.3 contains a single actor $\texttt{P}.f$ corresponding to a single firing function `P.fire`. Profiles

that contain a single firing function are called *monolithic*. The right-most profile of Figure 6.3 contains two actors, P.f_1 and P.f_2, corresponding to two firing functions, `P.fire1` and `P.fire2`: this is a *non-monolithic* profile. Notice that the dependency edge from P.f_1 to P.f_2 is redundant, since it is identical to the dataflow edge between the two. But the dataflow edge, in addition to a dependency, also encodes a transfer of data between the two firing functions.

Unless explicitly mentioned otherwise, in the examples that follow we assume that atomic blocks have monolithic profiles.

6.7 Profile Synthesis and Code Generation

As mentioned above, modular code generation takes as input a composite actor `P`, its internal graph, and profiles for every internal actor of `P`, and produces as output a profile for `P` and an implementation of this profile. Profile synthesis refers to the computation of a profile for `P`, while code generation refers to the automatic generation of an implementation of this profile. These two functions require a number of steps, detailed below.

Connecting the SDF Profiles

The first step consists in connecting the SDF profiles of internal actors of `P`. This is done simply as dictated by the connections found in the internal graph of `P`. The result is a flat DSSF graph, called the *internal profiles graph* (IPG) of `P`. We illustrate this through an example. Consider the composite actor `P` shown in Figure 6.1. Suppose both its internal actors `A` and `B` have monolithic profiles, with `A.`f and `B.`f representing `A.fire` and `B.fire`, respectively. Then, by connecting these monolithic profiles we obtain the IPG shown in Figure 6.4. In this case, the IPG is an SDF graph.

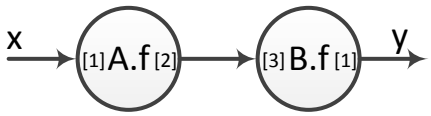


Figure 6.4: Internal profiles graph of composite actor `P` of Figure 6.1.

Two more examples of IPGs are shown in Figure 6.5. There, we connect the profiles of internal actors `P` and `C` of the (closed) graph shown at the left of Figure 6.2. Actor `C` is assumed to have a monolithic profile. Actor `P` has two possible profiles, shown in Figure 6.3. The two resulting IPGs are shown in Figure 6.5. The left-most one is an SDF graph. The right-most one is a DSSF graph, with two internal shared queues and three non-shared queues.

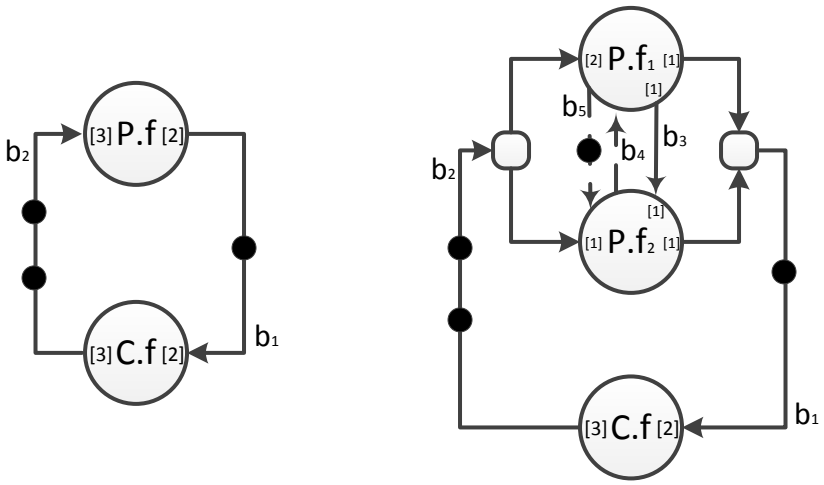


Figure 6.5: Two internal profiles graphs, resulting from connecting the two profiles of actor `P` shown in Figure 6.3 and a monolithic profile of actor `C`, according to the graph at the left of Figure 6.2.

Rate Analysis with SDF Profiles

This step is similar to the rate analysis process described in Section 6.4, except that it is performed on the IPG produced by the connection step, instead of an SDF graph. This presents no major challenges, however, and the method is essentially the same as the one proposed by [70].

Let us illustrate the process here, for the IPG shown to the right of Figure 6.5. We associate repetition variables r_p^1, r_p^2, and r_q, respectively, to $\texttt{P}.f_1$, $\texttt{P}.f_2$ and $\texttt{C}.f$. Then, we have the following balance equations:

$$\begin{aligned} r_p^1 \cdot 1 + r_p^2 \cdot 1 &= r_q \cdot 2 \\ r_q \cdot 3 &= r_p^1 \cdot 2 + r_p^2 \cdot 1 \\ r_p^1 \cdot 1 &= r_p^2 \cdot 1 \end{aligned}$$

As this has a non-trivial solution (e.g., $r_p^1 = r_p^2 = r_q = 1$), this graph is consistent, i.e., rate analysis succeeds in this example.

If the rate analysis step fails the graph is rejected. Otherwise, we proceed with the deadlock analysis step.

It is worth noting that rate analysis can sometimes succeed with non-monolithic profiles, whereas it would fail with a monolithic profile. An example is given in Figure 6.6. A composite actor `R` is shown to the left of the figure and its non-monolithic profile to the right. If we use `R` in the diagram shown to the middle of the figure, then rate analysis with the non-monolithic profile succeeds. It would fail, however, with the monolithic profile, since $\texttt{R}.f_2$ has to fire twice as often as $\texttt{R}.f_1$. This observation also explains why rate analysis must

generally be performed on the IPG, and not on the internal SDF graph using monolithic profiles for internal actors.

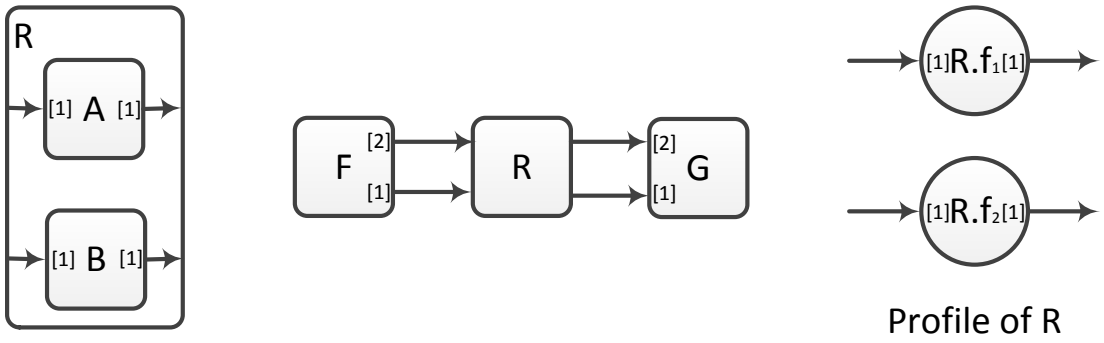


Figure 6.6: Composite SDF actor `R` (left); using `R` (middle); non-monolithic profile of `R` (right).

Deadlock Analysis with SDF Profiles

Success of the rate analysis step is a necessary, but not sufficient, condition in order for a graph to have a PASS. Deadlock analysis is used to ensure that this is the case. Deadlock analysis is performed on the IPG produced by the connection step. It is done in the same way as the deadlock detection process described in Section 6.4. We illustrate this on the two examples of Figure 6.5.

Consider first the IPG to the left of Figure 6.5. There are two queues in this graph: a queue from $\texttt{P}.f$ to $\texttt{C}.f$, and a queue from $\texttt{C}.f$ to $\texttt{P}.f$. Denote the former by b_1 and the latter by b_2. Initially, b_1 has 1 token, whereas b_2 has 2 tokens. $\texttt{P}.f$ needs 3 tokens to fire but only 2 are available in b_2, thus $\texttt{P}.f$ cannot fire. $\texttt{C}.f$ needs 2 tokens but only 1 is available in b_1, thus $\texttt{C}.f$ cannot fire either. Therefore there is a deadlock already at the initial state, and this graph is rejected.

Now consider the IPG to the right of Figure 6.5. There are five queues in this graph: a queue from $\texttt{P}.f_1$ and $\texttt{P}.f_2$ to $\texttt{C}.f$, a queue from $\texttt{C}.f$ to $\texttt{P}.f_1$ and $\texttt{P}.f_2$, two queues from $\texttt{P}.f_1$ to $\texttt{P}.f_2$, and a queue from $\texttt{P}.f_2$ to $\texttt{P}.f_1$. Denote these queues by b_1, b_2, b_3, b_4, b_5, respectively. Initially, b_1 has 1 token, b_2 has 2 tokens, b_3 and b_4 are empty and b_5 has 1 token. $\texttt{P}.f_1$ needs 2 tokens to fire and 2 tokens are indeed available in b_2, thus $\texttt{P}.f_1$ can fire and the initial state is not a deadlock. Deadlock analysis gives:

$$
\begin{aligned}
&(c_{p_1}=1, c_{p_2}=1, c_q=1, b_1=1, b_2=2, b_3=0, b_4=0, b_5=1) \xrightarrow{\texttt{P}.f_1}\\
&(c_{p_1}=0, c_{p_2}=1, c_q=1, b_1=2, b_2=0, b_3=1, b_4=1, b_5=0) \xrightarrow{\texttt{C}.f}\\
&(c_{p_1}=0, c_{p_2}=1, c_q=0, b_1=0, b_2=3, b_3=1, b_4=1, b_5=0) \xrightarrow{\texttt{P}.f_2}\\
&(c_{p_1}=0, c_{p_2}=0, c_q=0, b_1=1, b_2=2, b_3=0, b_4=0, b_5=1)
\end{aligned}
$$

Therefore, deadlock analysis succeeds (no deadlock is detected).

This example illustrates the tradeoff between compactness and reusability. For the same composite actor `P`, two profiles can be generated, as shown in Figure 6.3. These profiles achieve different tradeoffs. The monolithic profile shown to the left of the figure is more compact (i.e., smaller) than the non-monolithic one shown to the right. The latter is more reusable than the monolithic one, however: indeed, it can be reused in the graph with feedback shown at the left of Figure 6.2, whereas the monolithic one cannot be used, because it creates a deadlock.

Note that if we flatten the graph as shown in Figure 6.2, that is, remove composite actor `P` and replace it with its internal graph of atomic actors `A` and `B`, then the resulting graph has a PASS, i.e., exhibits no deadlock. This shows that deadlock is a result of using the monolithic profile, and not a problem with the graph itself. Of course, flattening is not the solution, because it is not modular: it requires the internal graph of `P` to be known and used in every context where `P` is used. Thus, code for `P` cannot be generated independently from context.

If the deadlock analysis step fails then the graph is rejected. Otherwise, we proceed with the unfolding step.

Unfolding with SDF Profiles

This step takes as input the IPG produced by the connection step, as well as the repetition vector produced by the rate analysis step. It produces as output a DAG (directed acyclic graph) that captures the input-output dependencies of the IPG. As mentioned in Section 6.4 the unfolding step is an adaptation of existing transformations from SDF to HSDF.

The DAG is computed in two steps. First, the IPG is *unfolded*, by replicating each node in it as many times as specified in the repetition vector. These replicas represent the different firings of the corresponding actor. For this reason, the replicas are ordered: dependencies are added between them to represent the fact that the first firing comes before the second firing, the second before the third, and so on. Ports are also replicated. In particular, port replicas are created for ports connected to the same queue. This is the case for replicas x_i and y_i shown in the figure. Note that we do not consider these to be shared queues, precisely because we want to capture dependencies between each separate production and consumption of tokens at the same queue. Finally, for every internal queue of the IPG, a queue is created in the unfolded graph with the appropriate connections. The process is illustrated in Figure 6.7, for the IPG of Figure 6.4. Rate analysis in this case produces the repetition vector ($r_{\mathtt{A}} = 3, r_{\mathtt{B}} = 2$). Therefore $\mathtt{A}.f$ is replicated 3 times and $\mathtt{B}.f$ is replicated 2 times. In this example there is a single internal queue between $\mathtt{A}.f$ and $\mathtt{B}.f$, and the unfolded graph contains a single shared queue.

In the second and final step of unfolding, the DAG is produced, by computing dependencies between the replicas. This is done by separately computing dependencies between replicas that are connected to a given queue, and repeating the process for every queue. We first explain the process for a non-shared queue such as the one between `A` and `B` in the IPG

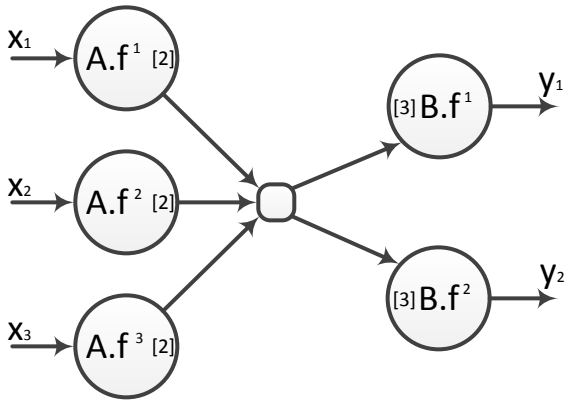


Figure 6.7: First step of unfolding the IPG of Figure 6.4: replicating nodes and creating a shared queue.

of Figure 6.4. Suppose that the queue has d initial tokens, its producer `A` adds k tokens to the queue each time it fires, and its consumer `B` removes n tokens each time it fires. Then the j-th occurrence of `B` depends on the i-th occurrence of `A` iff:

$$d + (i-1) \cdot k < j \cdot n \tag{6.1}$$

In that case, an edge from $\texttt{A}.f^i$ to $\texttt{B}.f^j$ is added to the DAG. For the example of Figure 6.4, this gives the DAG shown in Figure 6.8.

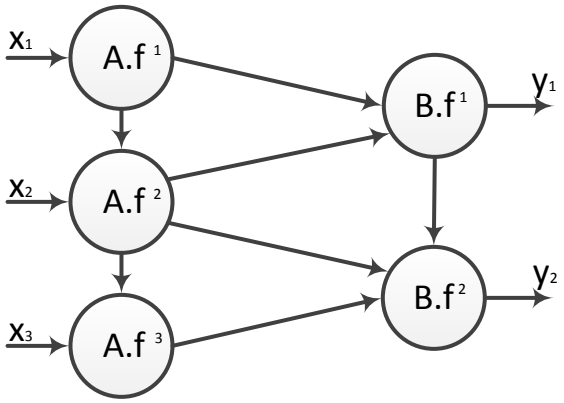


Figure 6.8: Unfolding the IPG of Figure 6.4 produces the IODAG shown here.

In the general case, a queue in the IPG of `P` may be shared by multiple producers and multiple consumers. Consider such a shared queue between a set of producers $\texttt{A}_1, ..., \texttt{A}_a$ and a set of consumers $\texttt{B}_1, ..., \texttt{B}_b$. Let k_h be the number of tokens produced by $\texttt{A}_h$, for $h = 1, ..., a$. Let n_h be the number of tokens consumed by $\texttt{B}_h$, for $h = 1, ..., b$. Let d be the number of initial

tokens in the queue. By construction (see Section 6.7) there is a total order $\mathtt{A}_1 \to \mathtt{A}_2 \to \cdots \to \mathtt{A}_a$ on the producers and a total order $\mathtt{B}_1 \to \mathtt{B}_2 \to \cdots \to \mathtt{B}_b$ on the consumers. As this is encoded with standard SDF edges of the form $\mathtt{A}_i \overset{1}{\to} \overset{1}{} \mathtt{A}_{i+1}$, this also implies that during rate analysis the rates of all producers will be found equal, and so will the rates of all consumers. Then, the j-th occurrence of $\mathtt{B}_u$, $1 \leq u \leq b$, depends on the i-th occurrence of $\mathtt{A}_v$, $1 \leq v \leq a$, iff:

$$d + (i-1) \cdot \sum_{h=1}^{a} k_h + \sum_{h=1}^{v-1} k_h < (j-1) \cdot \sum_{h=1}^{b} n_h + \sum_{h=1}^{u} n_h \tag{6.2}$$

Notice that, as should be expected, Equation (6.2) reduces to Equation (6.1) in the case $a = b = 1$.

Another example of unfolding, starting with an IPG that contains a non-monolithic profile, is shown in Figure 6.9. P is the composite actor of Figure 6.1. Its non-monolithic right-most profile of Figure 6.3 is used to form the IPG shown in Figure 6.9.

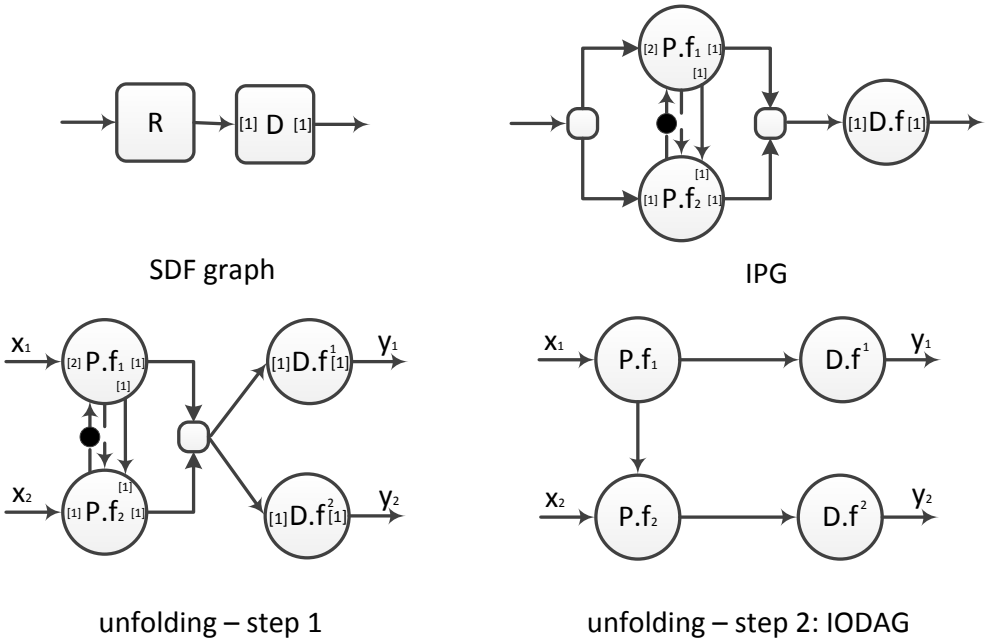


Figure 6.9: Another example of unfolding.

In the DAG produced by unfolding, input and output port replicas such as x_i and y_i in Figures 6.8 and 6.9 are represented explicitly as special nodes with no predecessors and no successors, respectively. For this reason, we call this DAG an IODAG. Nodes of the IODAG that are neither input nor output are called *internal* nodes.

DAG Clustering

DAG clustering consists in partitioning the internal nodes of the IODAG produced by the unfolding step into a number of *clusters*. The clustering must be *valid* in the sense that it must not create cyclic dependencies among distinct clusters[4]. Note that a monolithic clustering is trivially valid, since it contains a single cluster. Each of the clusters in the produced clustered graph will result in a firing function in the profile of `P`, as explained in Section 6.7 that follows. Exactly how DAG clustering is done is discussed in Section 6.8. There are many possibilities, which explore different tradeoffs, in terms of compactness, reusability, and other metrics. Here, we illustrate the *outcome* of DAG clustering on our running example. Two possible clusterings of the DAG of Figure 6.8 are shown in Figure 6.10, enclosed in dashed curves. The left-most clustering contains a single cluster, denoted C_0. The right-most clustering contains two clusters, denoted C_1 and C_2.

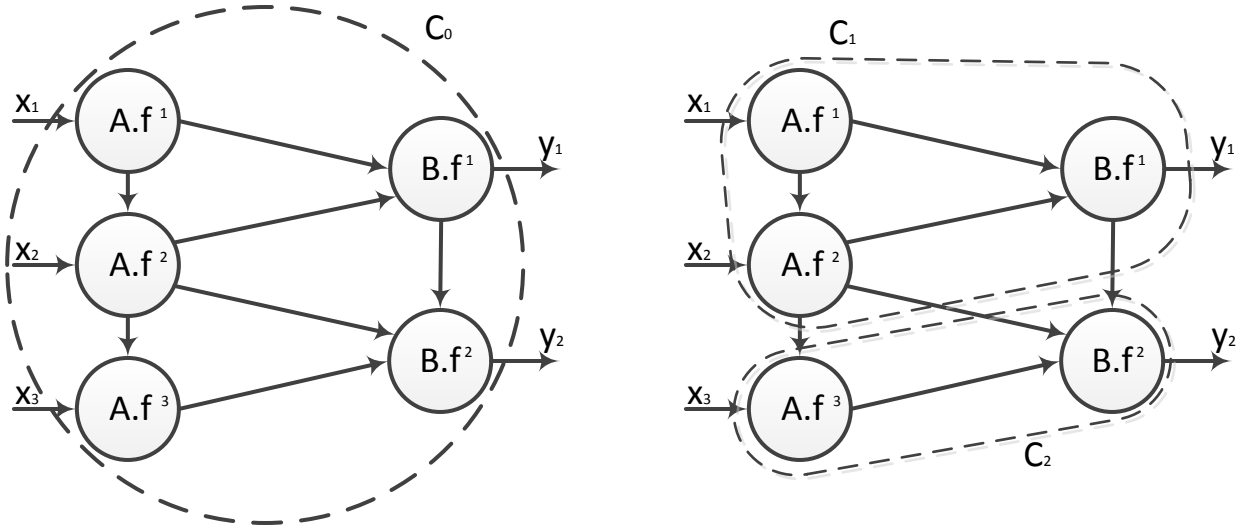


Figure 6.10: Two possible clusterings of the DAG of Figure 6.8.

Profile Generation

Profile generation is the last step in profile synthesis, where the actual profile of composite actor `P` is produced. The clustered graph, together with the internal graph of `P`, completely determine the profile of `P`. Each cluster C_i is mapped to a firing function `fire`i, and also to an atomic node $\mathtt{P}.f_i$ in the profile graph of `P`. For every input (resp. output) port of `P`, an external, potentially shared, queue L is created in the profile of `P`. For each cluster C_i, we

[4] A dependency between two distinct clusters exists iff there is a dependency between two nodes from each cluster.

compute the total number of tokens k_i read from (resp. written to) L by C_i: this can be easily done by summing over all actors in C_i. If $k_i > 0$ then an edge is added from L to $\mathtt{P}.f_i$ (resp. from $\mathtt{P}.f_i$ to L) annotated with a rate of k_i tokens.

Dependency edges between firing functions are computed as follows. For every pair of distinct clusters C_i and C_j, a dependency edge from $\mathtt{P}.f_i$ to $\mathtt{P}.f_j$ is added iff there exist nodes v_i in C_i and v_j in C_j such that v_j depends on v_i in the IODAG. Validity of clustering ensures that this set of dependencies results in no cycles. In addition to these dependency edges, we add a set of *backward* dependency edges to ensure a deterministic order of writing to (resp. reading from) shared queues also across iterations. Let L be a shared queue of the profile. L is either an external queue, or an internal queue of $\mathtt{P}$, like the one shown in Figure 6.7. Let $\mathtt{W}_L$ (resp. $\mathtt{R}_L$) be the set of all clusters writing to (resp. reading from) L. By the fact that different replicas of the same actor that are created during unfolding are totally ordered in the IODAG, all clusters in $\mathtt{W}_L$ are totally ordered. Let $\mathsf{C}_i, \mathsf{C}_j \in \mathtt{W}_L$ be the first and last clusters in $\mathtt{W}_L$ with respect to this total order, and let $\mathtt{P}.f_i$ and $\mathtt{P}.f_j$ be the corresponding firing functions. We encode the fact that the $\mathtt{P}.f_i$ cannot re-fire before $\mathtt{P}.f_j$ has fired, by adding a dependency edge from $\mathtt{P}.f_j$ to $\mathtt{P}.f_i$, supplied with an initial token. This is a backward dependency edge. Note that if $i = j$ then this edge is redundant. Similarly, we add a backward dependency edge, if necessary, among the clusters in $\mathtt{R}_\mathtt{L}$, which are also guaranteed to be totally ordered.

To establish the dataflow edges of the profile, we iterate over all internal (shared or non-shared) queues of the IPG of $\mathtt{P}$. Let L be an internal queue and suppose it has d initial tokens. Let m be the total number of tokens produced at L by all clusters writing to L: by construction, m is equal to the total number of tokens consumed by all clusters reading from L. For our running example (Figures 6.4, 6.8 and 6.10), we have $d = 0$ and $m = 6$.

Conceptually, we define m output ports denoted $z_0, z_1, ..., z_{m-1}$, and m input ports, denoted $w_0, w_1, ..., w_{m-1}$. For $i = 0$ to $m - 1$, we connect output port z_i to input port w_j, where $j = (d + i) \div m$, and $\div$ is the modulo operator. Intuitively, this captures the fact that the i-th token produced will be consumed as the $((d + i) \div m)$-th token of some iteration, because of the initial tokens. Notice that $j = i$ when $d = 0$. We then place $\lfloor \frac{d+m-1-i}{m} \rfloor$ initial tokens at each input port w_i, for $i = 0, ..., m - 1$, where $\lfloor v \rfloor$ is the integer part of v. Thus, if $d = 4$ and $m = 3$, then w_0 will receive 2 initial tokens, while w_1 and w_2 will each receive 1 initial token.

Finally, we assign the ports to producer and consumer clusters of L, according to their total order. For instance, for the non-monolithic clustering of Figure 6.10, output ports z_0 to z_3 and input ports w_0 to w_2 are assigned to C_1, whereas output ports z_4, z_5 and input ports w_3 to w_5 are assigned to C_2. Together with the port connections, these assignments define dataflow edges between the clusters. Self-loops (edges with same source and destination cluster) without initial tokens are removed. Note that more than one edges may exist between two distinct clusters, but these can always be merged into a single edge.

As an example, the two clusterings shown in Figure 6.10 give rise, respectively, to the two profiles shown in Figure 6.3. Another, self-contained example is shown in Figure 6.11. The two profiles shown at the bottom of the figure are generated from the clustering shown

at the top-right, assuming 6 and 17 initial tokens in the queue from A to B, respectively. Notice that if the queue contains 17 tokens then this clustering is not optimal, in the sense that a more coarse-grain clustering exists. However, the clustering is valid, and used here to illustrate the profile generation process.

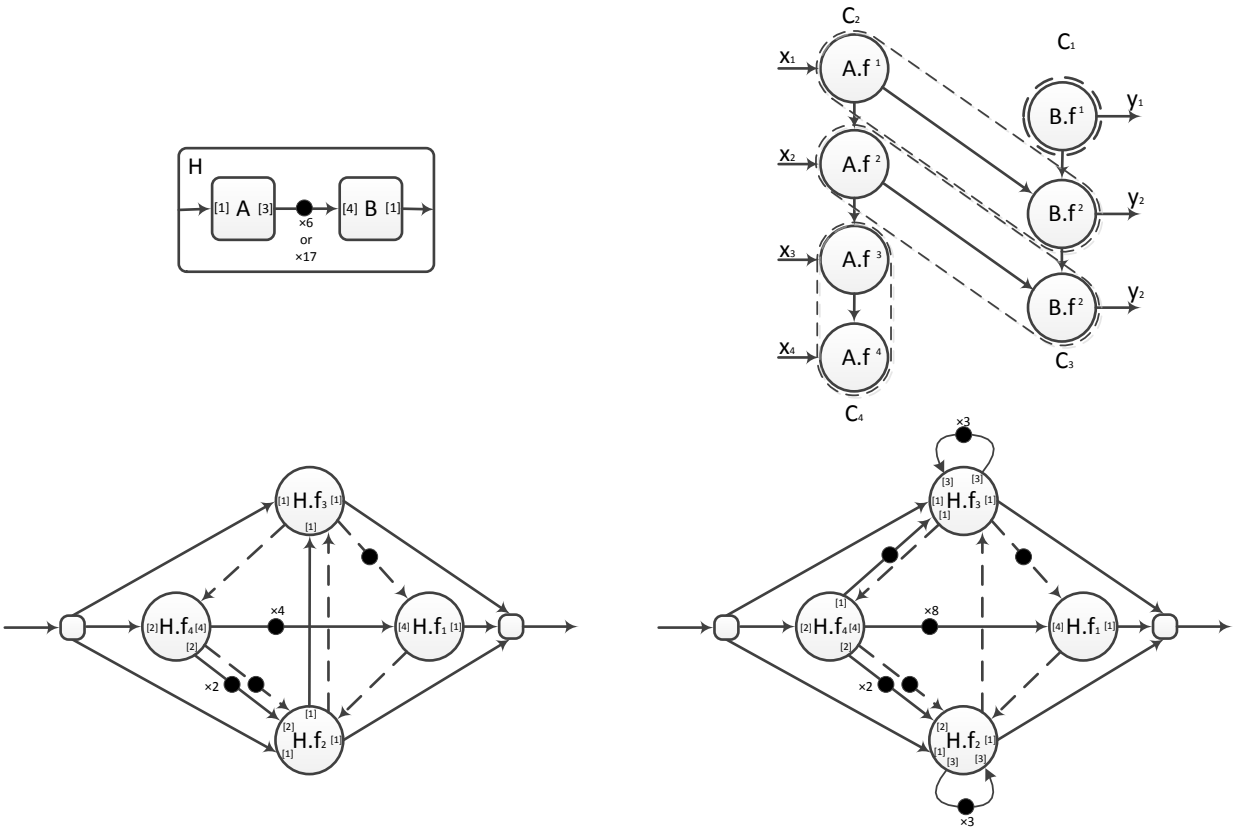


Figure 6.11: Composite SDF actor H (top-left); possible clustering produced by unfolding (top-right); SDF profiles generated for H, assuming 6 initial tokens in the queue from A to B (bottom-left); assuming 17 initial tokens (bottom-right). Firing functions $\mathtt{H}.f_1, \mathtt{H}.f_2, \mathtt{H}.f_3, \mathtt{H}.f_4$ correspond to clusters $\mathsf{C}_1, \mathsf{C}_2, \mathsf{C}_3, \mathsf{C}_4$, respectively.

Code Generation

Once the profile has been synthesized, its firing functions need to be implemented. This is done in the code generation step. Every firing function corresponds to a cluster produced by the clustering step. The implementation of the firing function consists in calling in a sequential order all firing functions of internal actors that are included in the cluster. This sequential order can be arbitrary, provided it respects the dependencies of nodes in the cluster. We illustrate the process on our running example (Figures 6.1, 6.3 and 6.10).

Consider first the clustering shown to the left of Figure 6.10. This will result in a single firing function for `P`, namely, `P.fire`. Its implementation is shown below in pseudo-code:

```
P.fire(input x[3]) output y[2]
{
  local tmp[4];
  tmp <- A.fire(x);
  tmp <- A.fire(x);
  y   <- B.fire(tmp);
  tmp <- A.fire(x);
  y   <- B.fire(tmp);
}
```

In the above pseudo-code, `tmp` is a local FIFO queue of length 4. Such a local queue is assumed to be empty when initialized. A statement such as `tmp <- A.fire(x)` corresponds to a call to firing function `A.fire`, providing as input the queue `x` and as output the queue `tmp`. `A.fire` will consume 1 token from `x` and will produce 2 tokens into `tmp`. When all statements of `P.fire` are executed, 3 tokens are consumed from the input queue `x` and 2 tokens are added to the output queue `y`, as indicated in the signature of `P.fire`.

Now let us turn to the clustering shown to the right of Figure 6.10. This clustering contains two clusters, therefore, it results in two firing functions for `P`, namely, `P.fire1` and `P.fire2`. Their implementation is shown below:

```
persistent local tmp[N];  /* N is a parameter */
assumption: N >= 4;

P.fire1(input x[2])      P.fire2(input x[1], tmp[1])
output y[1], tmp[1]      output y[1]
{                        {
  tmp <- A.fire(x);        tmp <- A.fire(x);
  tmp <- A.fire(x);        y   <- B.fire(tmp);
  y   <- B.fire(tmp);    }
}
```

In this case `tmp` is declared to be a *persistent* local variable, which means its contents "survive" across calls to `P.fire1` and `P.fire2`. In particular, of the 4 tokens produced and added to `tmp` by the two calls of `A.fire` within the execution of `P.fire1`, only the first 3 are consumed by the call to `B.fire`. The remaining 1 token is consumed during the execution of `P.fire2`. This is why `P.fire1` declares to produce at its output `tmp[1]` (which means it produces a total of 1 token at queue `tmp` when it executes), and similarly, `P.fire2` declares to consume at its input 1 token from `tmp`.

Dependency edges are not implemented in the code, since they carry no useful data, and only serve to encode dependencies between firing function calls. These dependencies must

be satisfied by construction in any correct usage of the profile. Therefore they do not need to be enforced in the implementation of the profile.

Discussion: Inadequacy of Cyclo-Static Data Flow

It may seem that *cyclo-static data flow* (CSDF) [18] can be used as an alternative representation of profiles. Indeed, this works on our running example: we could capture the composite actor `P` of Figure 6.1 using the CSDF actor shown in Figure 6.12. This CSDF actor specifies that `P` will iterate between two "firing modes". In the first mode, it consumes 2 tokens from its input and produces 1 token at its output; in the second mode, it consumes 1 token and produces 1 token; the process is then repeated. This indeed works for this example: embedding `P` as shown in Figure 6.2 results in no deadlock, if the CSDF model for `P` is used.

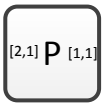


Figure 6.12: CSDF actor for composite actor `P` of Figure 6.1.

In general, however, CSDF models are not expressive enough to be used as profiles. We illustrate this by two examples[5], shown in Figures 6.6 and 6.13. Actors `R` and `W` are two composite actors shown in these figures. All graphs are homogeneous (tokens rates are omitted in Figure 6.13, they are implicitly all equal to 1; dependency edges are also omitted from the profile, since they are redundant). Our method generates the profiles shown to the right of the two figures. The profile for `R` contains two completely independent firing functions, $\texttt{R}.f_1$ and $\texttt{R}.f_2$. CSDF cannot express this independence, since it requires a fixed order of firing modes to be specified statically. Although two separate CSDF models could be used to capture this example, this is not sufficient for composite actor `W`, which features both internal dependencies and independencies.

6.8 DAG Clustering

DAG clustering is at the heart of our modular code generation framework, since it determines the profile that is to be generated for a given composite actor. As mentioned above, different tradeoffs can be explored during DAG clustering, in particular, in terms of compactness and reusability. In general: the more *fine-grain* the clustering is, the more reusable the profile and generated code will be; the more *coarse-grain* the clustering is, the more compact the

[5] [37] also observe that CSDF is not a sufficient abstraction of composite SDF models, however, the example they use embeds a composite SDF graph into a dynamic data flow model. Therefore the overall model is not strictly SDF. The examples we provide are much simpler, in fact, the models are homogeneous SDF models.

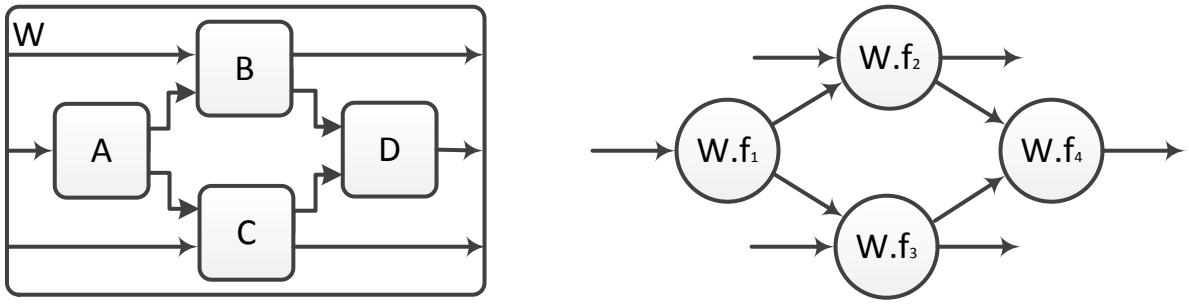


Figure 6.13: An example that cannot be captured by CSDF.

code is, but also less reusable. Note that there are cases where the most fine-grain clustering is wasteful since it is not necessary in order to achieve maximal reusability. Similarly, there are cases where the most coarse-grain clustering does not result in any loss of reusability. An instance of both cases is the trivial example where all outputs depend on all inputs, in which case the monolithic clustering is maximally reusable and also the most coarse possible. An extensive discussion of these and other tradeoffs can be found in previous work on modular code generation for SBDs [77, 76, 75]. The same principles apply also to SDF models.

DAG clustering takes as input the IODAG produced by the unfolding step. A trivial way to perform DAG clustering is to produce a single cluster, which groups together all internal nodes in this DAG. This is called *monolithic* DAG clustering and results in monolithic profiles that have a single firing function. This clustering achieves maximal compactness, but it results in non-reusable code in general, as the discussion of Section 6.7 demonstrates.

In this section we describe clustering methods that achieve *maximal reusability*. This means that the generated profile (and code) can be used in *any* context where the corresponding flattened graph could be used. Therefore, the profile results in no information loss as far as reusing the graph in a given context is concerned. At the same time, the profile may be much smaller than the internal graph.

To achieve maximal reusability, we follow the ideas proposed by [77, 75] for SBDs. In particular, we present a clustering method that is guaranteed not to introduce *false input-output dependencies*. These dependencies are "false" in the sense that they are not induced by the original SDF graph, but only by the clustering method.

To illustrate this, consider the monolithic clustering shown to the left of Figure 6.10. This clustering introduces a false input-output dependency between the third token consumed at input x (represented by node x_3 in the DAG) and the first token produced at output y (represented by node y_1). Indeed, in order to produce the first token at output y, only 2 tokens at input x are needed: these tokens are consumed respectively by the first two invocations of `A.fire`. The third invocation of `A.fire` is only necessary in order to produce the *second* token at y, but not the first one. The monolithic clustering shown to the left of Figure 6.10 loses this information. As a result, it produces a profile which is not

reusable in the context of Figure 6.2, as demonstrated in Section 6.7. On the other hand, the non-monolithic clustering shown to the right of Figure 6.10 preserves the input-output dependency information, that is, does not introduce false dependencies. Because of this, it results in a maximally reusable profile.

The above discussion also helps to explain the reason for the unfolding step. Unfolding makes explicit the dependencies between different productions and consumptions of tokens *at the same ports*. In the example of actor `P` (Figure 6.4), even though there is a single external input port x and a single external output port y in the IPG, there are three copies of x and two copies of y in the unfolded DAG, corresponding to the three consumptions from x and two productions to y that occur within a PASS.

Unfolding is also important because it allows us to re-use the clustering techniques proposed for SBDs, which work on plain DAGs [77, 75]. In particular, we can use the so-called *optimal disjoint clustering* (ODC) method which is guaranteed not to introduce false IO dependencies, produces a set of pairwise *disjoint* clusters (clusters that do not share any nodes), and is *optimal* in the sense that it produces a minimal number of clusters with the above properties. Unfortunately, the ODC problem is NP-complete [75]. This motivated us to develop a "greedy" DAG clustering algorithm, which is one of the contributions of this chapter. Our algorithm is not optimal, i.e., it may produce more clusters than needed to achieve maximal reusability. On the other hand, the algorithm has polynomial complexity. The greedy DAG clustering algorithm that we present below is "backward" in the sense that it proceeds from outputs to inputs. A similar "forward" algorithm can be used, that proceeds from inputs to outputs.

Greedy Backward Disjoint Clustering

The greedy backward disjoint clustering (GBDC) algorithm is shown in Figure 6.14. GBDC takes as input an IODAG (the result of the unfolding step) $G = (V, E)$ where V is a finite set of nodes and E is a set of directed edges. V is partitioned in three disjoint sets: $V = V_{\text{in}} \cup V_{\text{out}} \cup V_{\text{int}}$, the sets of input, output and internal nodes, respectively. GBDC returns a partition of V_{int} into a set of disjoint sets, called clusters. The partition (i.e., the set of clusters) is denoted $\mathcal{C}$. The problem is non-trivial when all $V_{\text{in}}, V_{\text{out}}$ and V_{int} are non-empty (otherwise a single cluster suffices). In the sequel, we assume that this is the case.

$\mathcal{C}$ defines a new graph, called the *quotient graph*, $G_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}})$. $G_{\mathcal{C}}$ contains clusters instead of internal nodes, and has an edge between two clusters (or a cluster and an input or output node) if the clusters contain nodes that have an edge in the original graph G. Formally, $V_{\mathcal{C}} = V_{\text{in}} \cup V_{\text{out}} \cup \mathcal{C}$, and $E_{\mathcal{C}} = \{(x, \mathsf{C}) \mid x \in V_{\text{in}}, \mathsf{C} \in \mathcal{C}, \exists f \in \mathsf{C} : (x, f) \in E\} \cup \{(\mathsf{C}, y) \mid \mathsf{C} \in \mathcal{C}, y \in V_{\text{out}}, \exists f \in \mathcal{C} : (f, y) \in E\} \cup \{(\mathsf{C}, \mathsf{C}') \mid \mathsf{C}, \mathsf{C}' \in \mathcal{C}, \mathsf{C} \neq \mathsf{C}', \exists f \in \mathsf{C}, f' \in \mathsf{C}', (f, f') \in E\}$. Notice that $E_{\mathcal{C}}$ does not contain self-loops (i.e., edges of the form (f, f)).

The steps of GBDC are explained below. E^* denotes the transitive closure of relation E: $(v, v') \in E$ iff there exists a path from v to v', i.e., v' depends on v.

Input: An IODAG $G = (V, E)$. $V = V_{\text{in}} \cup V_{\text{out}} \cup V_{\text{int}}$.
Output: A partition $\mathcal{C}$ of the set of internal nodes V_{int}.

```
1  foreach v ∈ V do
2  |  compute ins(v) and outs(v);
3  end
4  C ← ∅;
5  Out ← {f ∈ V_int | ∃y ∈ V_out : (f, y) ∈ E};
6  while ⋃C ≠ V_int do
7  |  partition Out into C_1, ..., C_k such that two nodes f, f' are grouped in the same set
   |  C_i iff ins(f) = ins(f');
8  |  C ← C ∪ {C_1, ..., C_k};
9  |  for i = 1 to k do
10 |  |  while
   |  |  ∃f ∈ C_i, f' ∈ V_int \ ⋃C : (f', f) ∈ E ∧ ∀x ∈ ins(C_i), y ∈ outs(f') : (x, y) ∈ E* do
11 |  |  |  C_i ← C_i ∪ {f'};
12 |  |  end
13 |  end
14 |  Out ← {f ∈ V_int \ ⋃C | ¬∃f' ∈ V_int \ ⋃C : (f, f') ∈ E};
15 end
16 while quotient graph G_C contains cycles do
17 |  pick a cycle C_1→C_2→ ··· →C_k→C_1;
18 |  C ← (C \ {C_1, ..., C_k}) ∪ {⋃_{i=1}^{k} C_i};
19 end
```

Figure 6.14: The GBDC algorithm.

Identify input-output dependencies Given a node $v \in V$, let $\mathsf{ins}(v)$ be the set of input nodes that v depends upon: $\mathsf{ins}(v) \leftarrow \{x \in V_{\text{in}} \mid (x, v) \in E^*\}$. Similarly, let $\mathsf{outs}(v)$ be the set of output nodes that depend on v: $\mathsf{outs}(v) \leftarrow \{y \in V_{\text{out}} \mid (v, y) \in E^*\}$. For a set of nodes F, $\mathsf{ins}(F)$ denotes $\bigcup_{f \in F} \mathsf{ins}(f)$, and similarly for $\mathsf{outs}(F)$.

Lines 1-3 of GBDC compute $\mathsf{ins}(v)$ and $\mathsf{outs}(v)$ for every node v of the DAG. We can compute these by starting from the output and following the dependencies backward. For example, consider the DAG of Figure 6.8. There are three input nodes, x_1, x_2, x_3 and two output nodes, y_1 and y_2. We have: $\mathsf{ins}(y_1) = \mathsf{ins}(\mathtt{B}.f^1) = \mathsf{ins}(\mathtt{A}.f^2) = \{x_1, x_2\}$, and $\mathsf{ins}(y_2) = \mathsf{ins}(\mathtt{B}.f^2) = \{x_1, x_2, x_3\}$. Similarly: $\mathsf{outs}(x_1) = \mathsf{outs}(\mathtt{A}.f^1) = \{y_1, y_2\}$, and $\mathsf{outs}(x_3) = \mathsf{outs}(\mathtt{A}.f^3) = \{y_2\}$.

Line 4 of GBDC initializes $\mathcal{C}$ to the empty set. Line 5 initializes Out as the set of internal nodes that have an output y as an immediate successor. These nodes will be used as "seeds" for creating new clusters (Lines 7-8).

Then the algorithm enters the while-loop at Line 6. $\bigcup \mathcal{C}$ is the union of all sets in $\mathcal{C}$, i.e.,

the set of all nodes clustered so far. When $\bigcup \mathcal{C} = V_{\text{int}}$ all internal nodes have been added to some cluster, and the loop exits. The body of the loop consists in the following steps:

Partition seed nodes with respect to input dependencies Line 7 partitions Out into a set of clusters, such that two nodes are put into the same cluster iff they depend on the same inputs. Line 8 adds these newly created clusters to $\mathcal{C}$. In the example of Figure 6.8, this gives an initial $\mathcal{C} = \{\{\mathtt{B}.f^1\}, \{\mathtt{B}.f^2\}\}$.

Create a cluster for each group of seed nodes The for-loop starting at Line 9 iterates over all clusters newly created in the previous step and attempts to add as many nodes as possible to each of these clusters, going backward, and making sure no false input-output dependencies are created in the process. In particular, for each cluster C_i, we proceed backward, attempting to add unclustered predecessors f' of nodes f already in C_i (while-loop at Line 10). Such a node f' is a candidate to be added to C_i, but this happens only if an additional condition is satisfied: namely $\forall x \in \mathsf{ins}(\mathsf{C}_i), y \in \mathsf{outs}(f') : (x, y) \in E^*$. This condition is violated if there exist an input node x that some node in C_i depends upon, and an output node y that depends on f' but not on x. In that case, adding f' to C_i would create a false dependency from x to y. Otherwise, it is safe to add f', and this is done in Line 11.

In the example of Figure 6.8, executing the while-loop at Line 10 results in adding nodes $\mathtt{A}.f^1$ and $\mathtt{A}.f^2$ in the cluster $\{\mathtt{B}.f^1\}$, and node $\mathtt{A}.f^3$ in the cluster $\{\mathtt{B}.f^2\}$, thereby obtaining the final clustering, shown to the right of Figure 6.10.

In general, more than one iteration may be required to cluster all the nodes. This is done by repeating the process, starting with a new Out set. In particular, Line 14 recomputes Out as the set of all unclustered nodes that have no unclustered successors.

Removing cycles The above process is not guaranteed to produce an acyclic quotient graph. Lines 16-19 remove cycles by repeatedly *merging* all clusters in a cycle into a single cluster. This process is guaranteed not to introduce false input-output dependencies, as shown in Lemma 5 of [75].

Termination and Complexity

Theorem 8 *Provided the set of nodes V is finite, GBDC always terminates.*

Proof: G is acyclic, therefore the set Out computed in Lines 5 and 14 is guaranteed to be non-empty. Therefore, at least one new cluster is added at every iteration of the while-loop at Line 6, which means the number of unclustered nodes decreases at every iteration. The for-loop and foreach-loop inside this while-loop obviously terminate, therefore, the body of the while-loop terminates. The second while-loop (Lines 16-19) terminates because the number of cycles is reduced by at least one at every iteration of the loops, and there can only be a finite number of cycles. ∎

Theorem 9 *GBDC is polynomial in the number of nodes in G.*

Proof: Let $n = |V|$ be the number of nodes in G. Computing sets ins and outs can be done in $O(n^2)$ time (perform forward and backward reachability from every node). Computing Out can also be done in $O(n^2)$ time (Line 5 or 14). The while-loop at Line 6 is executed at most n times. Partitioning Out (Line 7) can be done in $O(n^3)$ time and this results in $k \leq n$ clusters. The while-loop at Lines 10-12 is iterated no more than n^2 times and the safe-to-add-f' condition can be checked in $O(n^3)$ time. The quotient graph produced by the while-loop at Line 6 contains at most n nodes. Checking the condition at Line 16 can be done in $O(n)$ time, and this process also returns a cycle, if one exists. Executing Line 18 can also be done in $O(n)$ time. The loop at Line 16 can be executed at most n times, since at least one cluster is removed every time. ■

Note that the above complexity analysis is largely pessimistic. A tighter analysis as well as algorithmic optimizations are beyond the scope of this chapter and are left for future work. Also note that GBDC is polynomial in the IODAG G, which, being the result of the unfolding step, can be considerably larger than the original SDF graph. This is because the size of G depends on the repetition vector and therefore ultimately on the hyper-period of the system. Finding ways to deal with this complexity (which, it should be noted, is common to all methods for SDF graphs that rely on unfolding or similar steps) is also part of future work.

Correctness

GBDC is correct, in the sense that, first, it produces disjoint clusters and clusters all internal nodes, second, the resulting clustered graph is acyclic, and third, the resulting graph contains no input-output dependencies that were not already present in the input graph.

Theorem 10 *GBDC produces disjoint clusters and clusters all internal nodes.*

Proof: Disjointness is ensured by the fact that only unclustered nodes (i.e., nodes in $V_{\text{int}} \setminus \bigcup \mathcal{C}$) are added to the set Out (Lines 5 and 14) or to a newly created cluster C_i (Line 11). That all internal nodes are clustered is ensured by the fact that the while-loop at Line 6 does not terminate until all internal nodes are clustered. ■

Theorem 11 *GBDC results in an acyclic quotient graph.*

Proof: This is ensured by the fact that all potential cycles are removed in Lines 16-19. ■

Theorem 12 *GBDC produces a quotient graph $G_{\mathcal{C}}$ that has the same input-output dependencies as the original graph G.*

Proof: We need to prove that: $\forall x \in V_{\text{in}}, y \in V_{\text{out}} : (x, y) \in E^* \iff (x, y) \in E^*_{\mathcal{C}}$. We will show that this holds for the quotient graph produced when the while-loop of Lines 6-15 terminates. The fact that Lines 16-19 preserve IO dependencies is shown in Lemma 5 of [75].

The $\Rightarrow$ direction is trivial by construction of the quotient graph. There are two places where false IO dependencies can potentially be introduced in the while-loop of Lines 6-15: at Lines 7-8, where a new set of clusters is created and added to $\mathcal{C}$; or at Line 11, where a new node is added to an existing cluster. We examine each of these cases separately.

Consider first Lines 7-8: A certain number $k \geq 1$ of new clusters are created here, each containing one or more nodes. This can be seen as a sequence of operations: first, create cluster C_1 with a single node $f \in \text{Out}$, then add to C_1 a node $f' \in \text{Out}$ such that $\mathsf{ins}(f) = \mathsf{ins}(f')$ (if such an f' exists), and so on, until C_1 is complete; then create cluster C_2 with a single node, and so on, until all clusters $\mathsf{C}_1, ..., \mathsf{C}_k$ are complete. It suffices to show that no such creation or addition results in false IO dependencies.

Regarding creation, note that a cluster that contains a single node cannot add false IO dependencies, by definition of the quotient graph. Regarding addition, we claim that if a cluster C is such that $\forall f, f' \in \mathsf{C} : \mathsf{ins}(f) = \mathsf{ins}(f')$, then adding a node f'' such that $\mathsf{ins}(f'') = \mathsf{ins}(f)$, where $f \in \mathsf{C}$, results in no false IO dependencies. To see why the claim is true, let $y \in \mathsf{outs}(f'')$. Then $\mathsf{ins}(f'') \subseteq \mathsf{ins}(y)$. Since $\mathsf{ins}(f'') = \mathsf{ins}(f)$, for any $x \in \mathsf{ins}(f)$, we have $(x, y) \in E^*$. Similarly, for any $y \in \mathsf{outs}(f)$ and any $x \in \mathsf{ins}(f'')$, we have $(x, y) \in E^*$.

Consider next Line 11: The fact that f' is chosen to be a predecessor of some node $f \in \mathsf{C}_i$ implies that $\mathsf{ins}(f') \subseteq \mathsf{ins}(f) \subseteq \mathsf{ins}(\mathsf{C}_i)$. There are two cases where a new dependency can be introduced: Case 2(a): either between some input $x \in \mathsf{ins}(\mathsf{C}_i)$ and some output $y \in \mathsf{outs}(f')$; Case 2(b): or between some input $x' \in \mathsf{ins}(f')$ and some output $y' \in \mathsf{outs}(\mathsf{C}_i)$. In Case 2(a), the safe-to-add-f' condition at Line 10 ensures that if such x and y exist, then y already depends on x, otherwise, f' is not added to C_i. In Case 2(b), $\mathsf{ins}(f') \subseteq \mathsf{ins}(\mathsf{C}_i)$ implies $x' \in \mathsf{ins}(\mathsf{C}_i)$. This and $y' \in \mathsf{outs}(\mathsf{C}_i)$ imply that $(x', y') \in E^*$: indeed, if this is not the case, then cluster C_i already contains a false IO dependency before the addition of f'. ■

Clustering for Closed Models

It is worth discussing the special case where clustering is applied to a closed model, that is, a model where all input ports are connected. This in particular happens with top-level models used for simulation, which contain source actors that provide the input data. By definition, the IODAG produced by the unfolding step for such a model contains no input ports. In this case, a monolithic clustering that groups all nodes into a single cluster suffices and the GBDC algorithm produces the monolithic clustering for such a graph. Such a clustering will automatically give rise to a single firing function. Simulating the model then consists in calling this function repeatedly.

6.9 Implementation

We have built a preliminary implementation of the SDF modular code generation described above in the open-source Ptolemy II framework [36] (`http://ptolemy.org/`). The implementation uses a specialized class to describe composite SDF actors for which profiles can be generated. These profiles are captured in Java, and can be loaded when the composite actor is used within another composite. For debugging and documentation purposes, the tool also generates in the GraphViz format DOT (`http://www.graphviz.org/`) the graphs produced by the unfolding and clustering steps.

Using our tool, we can, for instance, generate automatically a profile for the Ptolemy II model depicted in Figure 6.15. This model captures the SDF graph given in Figure 3 of [37]. Actor `A2` is a composite actor designed so as to consume 2 tokens on each of its input ports and produce 2 tokens on each of its output ports each time it fires. For this, it uses the DownSample and UpSample internal actors: DownSample consumes 2 tokens at its input and produces 1 token at its output; UpSample consumes 1 token at its input and produces 2 tokens at its output. Actors `A1` and `A3` are homogeneous. The SampleDelay actor models an initial token in the queue from `A2` to `A3`. All other queues are initially empty.

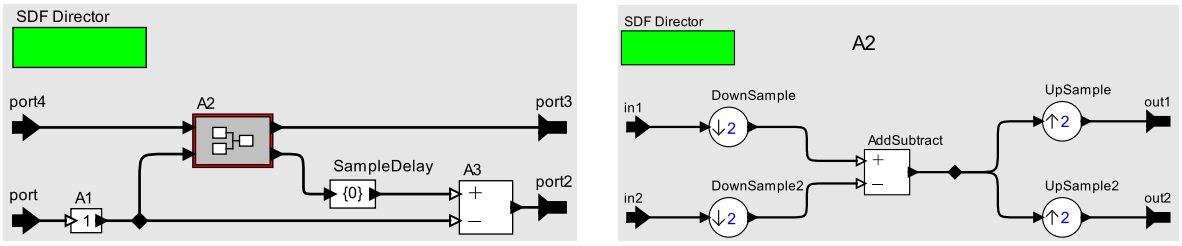


Figure 6.15: A hierarchical SDF model in Ptolemy II. The internal diagram of composite actor A2 is shown to the right.

Assuming a monolithic profile for A2, GBDC generates for the top-level Ptolemy model the clustering shown to the left of Figure 6.16. This graph is automatically generated by DOT from the textual output automatically generated by our tool. The two replicas of `A1` are denoted `A1_1_0` and `A1_2_0`, respectively, and similarly for `A2` and `A3`. Two clusters are generated, giving rise to the profile shown to the right of the figure. It is worth noting that there are 4 identical backward dependency edges generated for this profile (only one is shown). Moreover, all dependency edges are redundant in this case, thus can be removed. Finally, notice that the profile contains only two nodes, despite the fact that the Ptolemy model contains 9 actors overall.

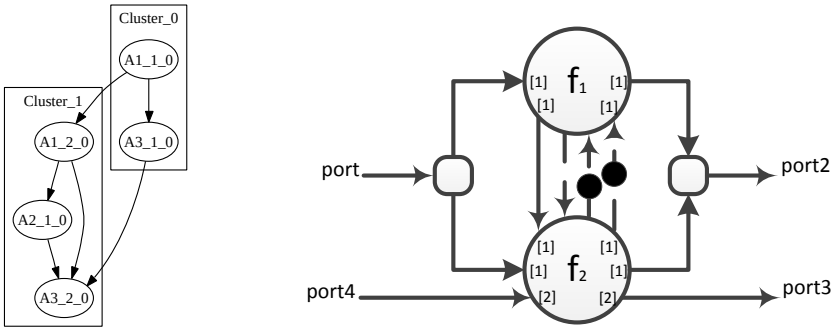


Figure 6.16: Clustering (left) and SDF profile (right) of the model of Figure 6.15.

6.10 Conclusions and Perspectives

Hierarchical SDF models are not compositional: a composite SDF actor cannot be represented as an atomic SDF actor without loss of information that can lead to deadlocks. Extensions such as CSDF are not compositional either. In this chapter we introduced DSSF profiles as a compositional representation of composite actors and showed how this representation can be used for modular code generation. In particular, we provided algorithms for automatic synthesis of DSSF profiles of composite actors given DSSF profiles of their subactors. This allows to handle hierarchical models of arbitrary depth. We showed that different tradeoffs can be explored when synthesizing profiles, in terms of compactness (keeping the size of the generated DSSF profile minimal) versus reusability (preserving information necessary to avoid deadlocks) as well as algorithmic complexity. We provided a heuristic DAG clustering method that has polynomial complexity and ensures maximal reusability.

In the future, we plan to examine how other DAG clustering algorithms could be used in the SDF context. This includes the clustering algorithm proposed by [77], which may produce overlapping clusters, with nodes shared among multiple clusters. This algorithm is interesting because it guarantees an upper bound on the number of generated clusters, namely, $n+1$, where n is the number of outputs in the DAG. Overlapping clusters result in complications during profile generation that need to be resolved.

Apart from devising or adapting clustering algorithms in the SDF context, part of future work is also to implement this algorithms in a tool such as Ptolemy, and compare their performance.

Another important problem is the efficiency of the generated code. Different efficiency goals may be desirable, such as buffer size, code length, and so on. Problems of code optimization in the SDF context have been extensively studied in the literature, see, for instance [13, 108]. One direction of research is to adapt existing methods to the modular SDF framework proposed here.

We would also like to study possible applications of DSSF to contexts other than modular code generation, for instance, compositional performance analysis, such as throughput or

latency computation. Finally, we plan to study possible extensions towards dynamic data flow models as well as towards distributed, multiprocessor implementations.

Chapter 7

Conclusion

This thesis focuses on mechanisms and methods to improve energy efficiency as well as programmability of stream programs. We demonstrate the use of the high-level abstractions of stream programs to synthesize *adaptive programs* for energy and resource efficient executions. Our technique for co-designing stream program stationary traffic properties with NoC's path diversity and DVFS capability can help significantly reduce communication energy of stream traffic. In the second part of this thesis, we give a formal definition for SDF integrated with COs, extended from the Teleport messaging work by Thies et al. [115]. SDF integrated with COs allows describing operation synchronization between computational parts of sophisticated stream programs. Our scheduling technique, developed for the COSDF programs, is not only more general than the previous techniques in [70] and [115] but also can help understand the formal parallelism of stream programs. Finally, we tackle the problem of incrementally compiling large SDF models to faithfully capture the executions of respective original models.

The adaptive program concept presented in this thesis can be generalized to other domains such as Cilk [38], PetaBricks [8] and SEDA [119]. We also show the usage of modular programming not only in helping programmers manage complex programs better but also in improving energy efficiency of programs by enabling runtime program transformation to dynamically adjust resource usage. Through the development of the routing technique for stream traffic on NoC, we present a case for *dynamic hardware-software co-design*. Our scheduling technique for COSDF programs can be used to find optimal buffer sizes as well as static communication schedules when mapping the programs to FPGA or coarse-grained reconfigurable arrays.

However, this thesis still has several limitations. We have not developed a method for estimating required computational resources at runtime for adaptive programs based on IO rates. The stream traffic routing technique has not accounted for the case then links and routers can be turned off to reduce leakage energy. We postulate that the scheduling technique developed for COSDF can help estimate required computational resources at runtime for adaptive programs.

In the future, we plan to apply the stream optimization techniques to another class of

stream applications such as streaming of big-data(`http://storm-project.net/`). When large programs are developed and deployed, it would be beneficial to decompose such programs into sub-modules and connect them via FIFO channels so that multiple programmers can contribute to the programs separately. This modular programming mechanism can also help scale the programs when sub-modules can be distributed to different machines. The Storm(`http://storm-project.net/`) is such a framework. However, modularity often comes at the price of communication and synchronization overhead between sub-modules when programs are decomposed into too many sub-modules. For example, each Storm sub-module is spawned as a Java process, as a consequence, a Storm program composed of many sub-modules can perform very badly because of context switching and Java Virtual Machine overheads [28]. A source-to-source translation process, which fuses multiple sub-modules together to reduce the number of sub-modules and mitigate inter-module communication overhead, can improve performance of large Storm programs significantly while still maintaining the programmability through the modular programming methodology.

Bibliography

[1] Ahmed H. Abdel-Gawad and Mithuna Thottethodi. TransCom: Transforming stream communication for load balance and efficiency in networks-on-chip. In *Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 237–247, New York, NY, USA, 2011. ACM.

[2] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pages 33 –42, april 2009.

[3] Alex Aiken and Alexandru Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, 1988.

[4] Farhana Aleen, Monirul Sharif, and Santosh Pande. Input-driven dynamic execution prediction of streaming applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 315–324, New York, NY, USA, 2010. ACM.

[5] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 148–165, London, UK, UK, 2001. Springer-Verlag.

[6] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, CC '84, 1984.

[7] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, Eric Rotenberg, and Frank Mueller. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *Proceedings of the 30th International Symposium on Computer Architecture*, ISCA '03, pages 350–361, New York, NY, USA, 2003. ACM.

[8] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[9] Farhad Arbab. Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming*, 55(1-3):3–52, 2005.

[10] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[11] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. Dimetrodon: Processor-level preventive thermal management via idle cycle injection. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 89–94, New York, NY, USA, 2011. ACM.

[12] Hyman Bass. Covering theory for graphs of groups. *Journal of Pure and Applied Algebra*, 89(12):3 – 47, 1993.

[13] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[14] Shuvra S. Bhattacharyya and Edward A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing Systems*, 6(3):271–288, 1993.

[15] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. In *DSP Block Diagrams into Efficient Software Implementations*, pages 33–60, 1997.

[16] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Optimized software synthesis for synchronous dataflow. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '97, pages 250–, Washington, DC, USA, 1997. IEEE Computer Society.

[17] Jeffrey C. Bier, Edwin E. Goei, Wai H. Ho, Philip D. Lapsley, Maureen P. OReilly, Gilbert C. Sih, and Edward A. Lee. Gabriel: A design environment for DSP. *IEEE Micro*, 10(5):28–45, September 1990.

[18] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397 –408, feb 1996.

[19] Simon Bliudze and Joseph Sifakis. The algebra of connectors: Structuring interaction in BIP. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software*, EMSOFT '07, pages 11–20, New York, NY, USA, 2007. ACM.

[20] Paul Bogdan, Radu Marculescu, Siddharth Jain, and Rafael T. Gavila. An optimal control approach to power management for multi-voltage and frequency islands multiprocessor platforms under highly variable workloads. In *Proceedings of the Sixth*

IEEE/ACM International Symposium on Networks on Chip, NOCS '12, pages 35–42, 2012.

[21] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In *Proceedings of the International Colloquium on Formalization of Programming Concepts*, pages 252–259, London, UK, UK, 1981. Springer-Verlag.

[22] Stephen D. Brookes and Andrew William Roscoe. Deadlock analysis in networks of communicating processes. *Logics and Models of Concurrent Systems*, 1985.

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

[24] Dai Bui and Edward A. Lee. StreaMorph: A case for synthesizing energy-efficient adaptive programs using high-level abstractions. In *Proceedings of the 13th ACM/IEEE International Conference on Embedded Software*, EMSOFT '13. ACM, 2013.

[25] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, July 2004.

[26] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions of Computing Systems*, 1, May 1983.

[27] Guangyu Chen, Feihui Li, Mahmut Kandemir, and Mary Jane Irwin. Reducing NoC energy consumption through compiler-directed channel voltage scaling. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 193–203, New York, NY, USA, 2006. ACM.

[28] Kuo-Yi Chen, J.M. Chang, and Ting-Wei Hou. Multithreading in Java: Performance and scalability on multicore systems. *IEEE Transactions on Computers*, 60(11):1521–1534, 2011.

[29] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. In *Proceedings of the Fourth European Symposium on Algorithms*, ESA '96, 1996.

[30] Kihwan Choi, Karthik Dantu, Wei-Chung Cheng, and Massoud Pedram. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '02, pages 732–737, New York, NY, USA, 2002. ACM.

[31] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & Cap: Adaptive DVFS and thread packing under power caps. In *Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 175–185, New York, NY, USA, 2011. ACM.

[32] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[33] Alain Darte. Understanding loops: The influence of the decomposition of Karp, Miller, and Winograd. In *Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, MEMOCODE '10, 2010.

[34] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[35] Matthew Drake. *Stream programming for image and video compression*. M.S. thesis, Massachusetts Institute of Technology, 2006.

[36] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[37] Joachim Falk, Joachim Keinert, Christian Haubelt, Jrgen Teich, and Shuvra S. Bhattacharyya. A generalized static data flow clustering algorithm for MPSOC scheduling of multimedia applications. In *Proceedings of the 8th ACM/IEEE International Conference on Embedded Software*, EMSOFT '08, pages 189–198. ACM, 2008.

[38] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[39] Marc Geilen and Twan Basten. Reactive process networks. In *Proceedings of the 4th ACM/IEEE International Conference on Embedded Software*, EMSOFT '04, pages 137–146, New York, NY, USA, 2004. ACM.

[40] Marc Geilen and Sander Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 125–134, New York, NY, USA, 2010. ACM.

[41] Marc C.W. Geilen. Reduction of Synchronous Dataflow Graphs. In *Proceedings of the 46th Design Automation Conference*, DAC '09. ACM, 2009.

[42] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[43] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association.

[44] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. In *Proceedings of the 19th International Symposium on Computer Architecture*, ISCA '92, pages 278–287, New York, NY, USA, 1992. ACM.

[45] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '04, pages 260–270, New York, NY, USA, 2004. ACM.

[46] Michael I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2010.

[47] Michael I. Gordon, William Thies, and Saman 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, 2006.

[48] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 199–212, New York, NY, USA, 2011. ACM.

[49] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 214–223, Washington, DC, USA, 2009. IEEE Computer Society.

[50] Amir H. Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. MacroSS: macro-SIMDization of streaming applications. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, 2010.

[51] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, 2011.

[52] Susan Horwitz, Thomas W. Reps, and Dave W. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, 1988.

[53] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51 –61, sept.-oct. 2007.

[54] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, ISSCC '10, pages 108 –109, feb. 2010.

[55] Kazuo Iwano and Kenneth Steiglitz. Testing for cycles in infinite graphs with periodic structure. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, STOC '87, 1987.

[56] Anoop Iyer and Diana Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '02, pages 379–386, New York, NY, USA, 2002. ACM.

[57] Bengt Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, 7(4):197–212, 1994.

[58] Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, Aug 1974.

[59] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. Orion 2.0: A power-area simulator for interconnection networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(1):191 –196, jan. 2012.

[60] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36, August 2003.

[61] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of ACM*, 14, July 1967.

[62] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[63] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.

[64] Michel A. Kinsy, Myong Hyon Cho, Tina Wen, Edward Suh, Marten van Dijk, and Srinivas Devadas. Application-aware deadlock-free oblivious routing. In *Proceedings of the 36th International Symposium on Computer Architecture*, ISCA '09, pages 208–219, New York, NY, USA, 2009. ACM.

[65] Jon Michael Kleinberg. *Approximation algorithms for disjoint paths problems*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1995.

[66] Muralidharan S. Kodialam. *The O-D shortest path problem and connectivity problems on periodic graphs*. Ph.D. thesis, Massachusetts Institute of Technology, 1992.

[67] Hermann Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems*, ISADS '99, pages 26–, Washington, DC, USA, 1999. IEEE Computer Society.

[68] Manjunath Kudlur and Scott 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.

[69] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[70] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[71] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[72] Feihui Li, Guangyu Chen, and Mahmut Kandemir. Compiler-directed voltage scaling on communication links for reducing power consumption. In *Proceedings of the 2005*

IEEE/ACM International Conference on Computer-Aided Design, ICCAD '05, pages 456–460, Washington, DC, USA, 2005. IEEE Computer Society.

[73] Feihui Li, Guangyu Chen, Mahmut Kandemir, and Ibrahim Kolcu. Profile-driven energy reduction in network-on-chips. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 394–404, New York, NY, USA, 2007. ACM.

[74] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, energy, and thermal considerations for SMT and CMP architectures. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 71–82, Washington, DC, USA, 2005. IEEE Computer Society.

[75] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, 2009.

[76] Roberto Lublinerman and Stavros Tripakis. Modular Code Generation from Triggered and Timed Block Diagrams. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08. IEEE CS Press, April 2008.

[77] Roberto Lublinerman and Stavros Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Proceedings of the 2008 Design, Automation and Test in Europe Conference*, DATE '08, pages 1504–1509. ACM, March 2008.

[78] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, and Antonia Zhai. Energy efficient speculative threads: Dynamic thread allocation in Same-ISA heterogeneous multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 453–464, New York, NY, USA, 2010. ACM.

[79] Nancy Lynch and Eugene W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, July 1989.

[80] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6-th ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

[81] Radu Marculescu, Umit Y. Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(1):3–21, January 2009.

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

[83] Asit K. Mishra, Reetuparna Das, Soumya Eachempati, Ravi Iyer, N. Vijaykrishnan, and Chita R. Das. A case for dynamic frequency tuning in on-chip networks. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture*, MICRO '42, pages 292–303, New York, NY, USA, 2009. ACM.

[84] Srinivasan Murali, David Atienza, Luca Benini, and Giovanni De Michel. A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip. In *Proceedings of the 43rd Design Automation Conference*, DAC '06, pages 845–848, New York, NY, USA, 2006. ACM.

[85] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[86] Praveen K. Murthy and Shuvra S. Bhattacharyya. A buffer merging technique for reducing memory requirements of synchronous dataflow specifications. In *Proceedings of 12th International Symposium on System Synthesis*, pages 78–84, 1999.

[87] Praveen K. Murthy, Shuvra S. Bhattacharyya, and Edward A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design*, 11(1):41–70, July 1997.

[88] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8), Aug 2002.

[89] Umit Y. Ogras, Radu Marculescu, Diana Marculescu, and Eun Gu Jung. Design and management of voltage-frequency island partitioned networks-on-chip. *IEEE Transaction on Very Large Scale Integrated Systems*, 17(3):330–341, March 2009.

[90] James B. Orlin. Some problems on dynamic/periodic graphs. *Progress in Combinatorial Optimization*, 1984.

[91] Keshab K. Parhi and David G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, February 1991.

[92] Jaehyun Park, Donghwa Shin, Naehyuck Chang, and Massoud Pedram. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 419–424, New York, NY, USA, 2010. ACM.

[93] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California, Berkeley, 1995.

[94] Thomas M. Parks, Jose L. Pino, and Edward A. Lee. A comparison of synchronous and cycle-static dataflow. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, ASILOMAR '95, page 204, Washington, DC, USA, 1995. IEEE Computer Society.

[95] Jose L. Pino, Shuvra S. Bhattacharyya, and Edward A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Technical Report UCB/ERL M95/36, EECS Department, University of California, Berkeley, 1995.

[96] Markku. Renfors and Yrjo Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Transactions on Circuits and Systems*, 28(3):196 – 202, Mar 1981.

[97] Vwani P. Roychowdhury and Thomas Kailath. Study of parallelism in regular iterative algorithms. In *Proceedings of the second ACM Symposium on Parallel Algorithms and Architectures*, SPAA '90, 1990.

[98] Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, PLDI '12, pages 13–22, New York, NY, USA, 2012. ACM.

[99] Mainak Sen, Shuvra S. Bhattacharyya, Tiehan Lv, and Wayne Wolf. Modeling image processing systems with homogeneous parameterized dataflow graphs. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP 05, 2005.

[100] Janis Sermulins, William Thies, Rodric Rabbah, and Saman 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.

[101] Sanjit A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *Proceedings of the 49th Design Automation Conference*, DAC '12, pages 356–365, New York, NY, USA, 2012. ACM.

[102] Li Shang, Li-Shiuan Peh, and Niraj K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 91–, Washington, DC, USA, 2003. IEEE Computer Society.

[103] Li Shang, Li-Shiuan Peh, and Niraj K. Jha. PowerHerd: Dynamic satisfaction of peak power constraints in interconnection networks. In *Proceedings of the 17th International Conference on Supercomputing*, ICS '03, pages 98–108, New York, NY, USA, 2003. ACM.

[104] Keun Sup Shim, Myong Hyon Cho, Michel Kinsy, Tina Wen, Mieszko Lis, G. Edward Suh, and Srinivas Devadas. Static virtual channel allocation in oblivious routing. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '09, pages 38–43, Washington, DC, USA, 2009. IEEE Computer Society.

[105] Gilbert C. Sih and Edward A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):625–637, June 1993.

[106] Vassos Soteriou, Noel Eisley, and Li-Shiuan Peh. Software-directed power-aware interconnection networks. *ACM Transactions on Architecture and Code Optimization*, 4(1), March 2007.

[107] Vaidyanathan Srinivasan, Gautham R. Shenoy, Srivatsa Vaddagiri, Dipankar Sarma, and Venkatesh Pallipadi. Energy-aware task and interrupt management in Linux. volume 2 of *Proceedings of the Linux Symposium*, Aug 2008.

[108] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2ed edition, 2009.

[109] Eugene W. Stark. Concurrent transition system semantics of process networks. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 199–210, New York, NY, USA, 1987. ACM.

[110] Eugene W. Stark. An algebra of dataflow networks. *Fundamenta Informaticae*, 22(1/2):167–185, 1995.

[111] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[112] William Thies. *Language and Compiler Support for Stream Programs*. Ph.D. thesis, Massachusetts Institute of Technology, Feb 2009.

[113] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, 2010.

[114] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, CC '02, Apr 2002.

[115] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, 2005.

[116] Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs. *ACM Transactions in Embedded Computing Systems (TECS)*, 12(3):83:1–83:26, April 2013.

[117] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[118] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, 2009.

[119] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM Symposium on Operating systems principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[120] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.

[121] Youfeng Wu, Shiliang Hu, Edson Borin, and Cheng Wang. A HW/SW co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing. In *Proceedings of the 9th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 236–245, Washington, DC, USA, 2011. IEEE Computer Society.

[122] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 49–62, New York, NY, USA, 2003. ACM.

[123] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of the ACM SIGPLAN 2001 Confer-*

ence on Programming Language Design and Implementation, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM.

[124] Ye Zhou and Edward A. Lee. A causality interface for deadlock analysis in dataflow. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software*, EMSOFT '06, 2006.