Efficient Software Implementation of Stream Programs

Cedersjö, Gustav

2017

Document Version:
Publisher's PDF, also known as Version of record

Efficient Software Implementation of Stream Programs

Gustav Cedersjö

LUND UNIVERSITY

AKADEMISK AVHANDLING som, med godkännande av tekniska fakulteten vid Lunds universitet, kommer att offentligen försvaras onsdagen den 7 juni 2017, klockan 13.15 i sal E:1406 i E-huset, Ole Römers väg 3, Lund, för avläggande av teknologie doktorsexamen.

Fakultetsopponent: tekn.dr Ingo Sander

ACADEMIC DISSERTATION which, by due permission of the Faculty of Engineering at Lund University, will be publicly defended on Wednesday, June 7, 2017 at 1.15 p.m. in room E:1406 of E-huset, Ole Römers väg 3, Lund, for the degree of Doctor of Philosophy in Engineering.

Faculty opponent: Dr. Ingo Sander

Organization
LUND UNIVERSITY
Department of Computer Science
Box 118, 221 00 Lund

Document name
DOCTORAL DISSERTATION

Date of disputation
2017-06-07

Sponsoring organization
Partially funded by ELLIIT (Excellence Center Linköping – Lund in Information Technology, funded by the Swedish Government)

Author(s)
Gustav Cedersjö

Title and subtitle
Efficient Software Implementation of Stream Programs

Abstract
The way we use computers and mobile phones today requires large amounts of processing of data streams. Examples include digital signal processing for wireless transmission, audio and video coding for recording and watching videos, and noise reduction for the phone calls. These tasks can be performed by stream programs—computer programs that process streams of data.

Stream programs can be composed of other stream programs. Components of a composition are connected in a network, i.e. the output streams of one component are sent as input streams to other components. The components, that perform the actual computation, are called kernels. They can be described in different styles and programming languages. There are also formal models for describing the kernels and the networks. One such model is the actor machine.

This dissertation evaluates the actor machine, how it facilitates creating efficient software implementation of stream programs. The evaluation is divided into four aspects: (1) analyzability of its structure, (2) generality in what languages and styles it can express, (3) efficient implementation of kernels, and (4) efficient implementation of networks. This dissertation demonstrates all four aspects through implementation and evaluation of a stream program compiler based on actor machines.

Key words
stream programs, dataflow with firing, Kahn processes, compilers, actor machine

Classification system and/or index terms (if any)

Supplementary bibliographical information

Language
English

ISSN and key title
ISSN 1404-1219
Efficient Software Implementation of Stream Programs

ISBN
978-91-7753-319-1 (print)
978-91-7753-320-7 (pdf)

Recipient's notes

Number of pages
155

Price

Security classification

Signature _____ Date 2017-05-02

Efficient Software Implementation of Stream Programs

Efficient Software Implementation of Stream Programs

Gustav Cedersjö

LUND UNIVERSITY

ISBN 978-91-7753-319-1 (print)
ISBN 978-91-7753-320-7 (pdf)
ISSN 1404-1219
Dissertation 57, 2017
LU-CS-DISS 2017-3

Abstract

The way we use computers and mobile phones today requires large amounts of processing of data streams. Examples include digital signal processing for wireless transmission, audio and video coding for recording and watching videos, and noise reduction for the phone calls. These tasks can be performed by stream programs—computer programs that process streams of data.

Stream programs can be composed of other stream programs. Components of a composition are connected in a network, i.e. the output streams of one component are sent as input streams to other components. The components, that perform the actual computation, are called kernels. They can be described in different styles and programming languages. There are also formal models for describing the kernels and the networks. One such model is the actor machine.

This dissertation evaluates the actor machine, how it facilitates creating efficient software implementation of stream programs. The evaluation is divided into four aspects: (1) analyzability of its structure, (2) generality in what languages and styles it can express, (3) efficient implementation of kernels, and (4) efficient implementation of networks. This dissertation demonstrates all four aspects through implementation and evaluation of a stream program compiler based on actor machines.

Effektiva datorprogram för strömmande data

Popular science summary in Swedish.

VAD ÄR ETT DATORPROGRAM?

Ett datorprogram kan vara ett tidsfördriv på bussen, eller ett designverktyg för en grafisk formgivare, eller ett livsviktigt styrsystem till pacemakern. Men rent tekniskt är det en lista med instruktioner som datorn ska utföra. Instruktionerna är väldigt primitiva. En instruktion kan till exempel vara att addera två tal som finns i olika register och spara summan i ett tredje register. När datorn har utfört en instruktion fortsätter den med nästa i listan. En instruktion kan också vara att hoppa i listan, om det står en nolla i register tre, hoppa elva steg framåt. Det är ungefär som enkätundersökningar:

1. Tycker du om kaffe?
☐ Ja
☐ Nej
Om du svarade nej, hoppa till fråga 4.

SPRÅK FÖR DATORN OCH MÄNNISKAN

Ett problem när man ska skriva program är att instruktionerna är så primitiva att det blir svårt att beskriva en komplex uppgift. För att underlätta att bygga program finns speciella språk, s.k. programmeringsspråk. Dessa språk finns i gränslandet mellan vad människan och maskinen kan förstå; samtidigt som språket är anpassat för att kunna förstås av människor, måste det kunna översättas maskinellt till instruktionslistor för datorn.

Det finns tusentals mer eller mindre kända programmeringsspråk och till varje språk finns program som översätter programtext till instruktionslistor för datorn. Översättningen kan antingen göras i förväg, eller under tiden som programmet körs. En och samma programtext kan dessutom översättas till olika instruktionslistor så länge resultatet, när datorn kör programmet, stämmer överens med vad programtexten beskriver. Det forskas mycket på hur översättarna kan skapa effektiva instruktionslistor som går snabbt att utföra.

Vissa språk är inspirerade av matematik, där texten mestadels är matematiska funktioner och uttryck. Programmering med sådana språk kallas ibland för funktionsprogrammering. Andra språk är inspirerade av datorns instruktionslista, men där orden och strukturen är anpassad för att kunna enklare förstås av en människa. Det kallas ofta imperativ programmering, efter verbformen imperativ, eftersom programmet säger åt datorn vad den ska göra.

STRÖMMANDE DATA

Denna avhandling handlar om språk som är anpassade för att behandla strömmande data. Som programmerare kan man tänka sig programmet som en mojäng som successivt tar emot data genom ingångar på ena sidan och producerar ny data som skickas genom utgångar på andra sidan. Inne i mojängen finns en beskrivning av vad som ska göras med den data som kommer in och vilken data som ska skickas ut.

Mojängerna, som är datorprogram, kan sättas ihop i stora nätverk, där utdatan från en mojäng matas som indata till en annan mojäng. Avhandlingen handlar om en modell för att beskriva hur mojängerna fungerar inuti och hur man utifrån den modellen kan översätta ett nätverk av mojänger till en effektiv lista med instruktioner som en dator kan utföra.

Figur 1: Ett program som kan ta emot tre dataströmmar och producera två dataströmmar.

Contents

Acknowledgments

First of all, I would like to express my gratitude to my doctoral advisor Jörn Janneck for his guidance and honesty, and for always being available for discussions and advice. He has introduced me to stream computing, which has opened my mind for new ways of expressing computer programs. I would also like to thank my assistant doctoral advisor Krzysztof Kuchcinski for his excellent leadership of the research group and for sharing his experience of doing research in Sweden.

I am also grateful to the rest of the research group—Flavius, Jonas, Mehmet, Patrik, Per, Shub, Usman and Vivek—for all your feedback and for sharing your expertise, your humor and your music. Furthermore, I would like to thank my co-workers and co-authors from outside the research group, for widening my perspectives on the research. I would also like to thank Lena, Camilla, Anne-Marie, Peter and the rest of the administrative and technical staff for their support and service.

One of the things that has made my workplace great is the "fika" and the lunch in the department's lunchroom. Without the (sometimes too long) "fika" and lunch discussions, this work wouldn't have been the same. We have discussed politics, teaching, housing, family, programming, text editors, keyboard switches and a lot of other things.

My thankfulness also goes to my parents for always encouraging me to learn and to educate myself. Most of all, I would like to express my deepest gratitude to my family—to my wife Mia for her love, care and support, especially during this project, and to my children Doris and Hilda for being such wonderful kids. You have brought enormous amounts of joy and love to my life.

Gustav Cedersjö
Lund, April 2017

Part I

Background

Chapter 1

Introduction

A stream program is a computer program that accepts input and produces output as streams of data items, called *tokens*. Figure 1.1 shows a visualization of a stream program in a state where three tokens are available for consumption and one token has been produced. The triangles in the figure represent *ports*, through which the tokens are consumed and produced. An output port of a stream program can be connected to an input port of another stream program, forming one parallel stream program from two concurrently executing programs. In fact, arbitrary networks of stream programs can be connected to

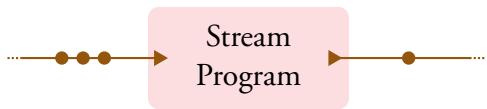


Figure 1.1: Stream program with three incoming tokens, and one that has been produced.

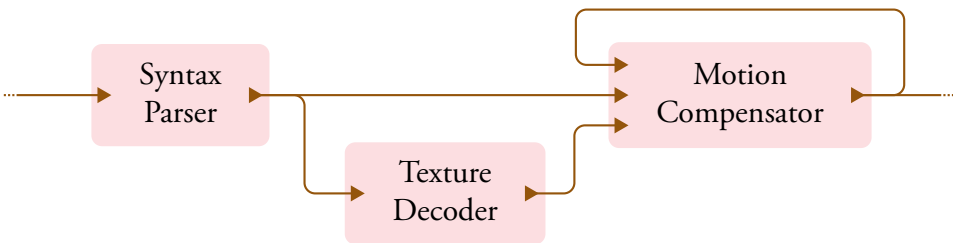


Figure 1.2: Top level network description of a video decoder, where each node, in turn, represents a network of stream programs. The compressed video stream arrives on the open edge to the left, and the uncompressed video stream is sent out to the open edge to the right. (For clarity, multiple edges between the nodes have been collapsed.)

form a parallel stream program. Since networks of stream programs are also stream programs, they can be composed hierarchically. Figure 1.2 shows the top level of a hierarchically defined video decoder.

The atomic stream programs, which are *not* composed of other stream programs, are called computational kernels, or just *kernels*. When implementing stream programs in software, one of the biggest challenges is to efficiently schedule the kernels on the available processors. Efficient scheduling is also one of the biggest strengths of stream programs compared to thread based programs, due to the explicit data dependencies defined by the connections.

A kernel can be described by a sequential program with read and write statements for consuming and producing tokens. Such kernels are called *processes*. Conceptually, they execute as continuous programs that pause when they need more input tokens.

Another way of describing a kernel is by a set of *transitions* that the kernel can make, together with *conditions* for when the transitions can be made. The conditions make sure that a transition is not initiated unless there is enough input to complete the transition and there is space available for its output. In a transition, the kernel may consume and produce tokens and update its internal state variables. In addition to how many tokens will be consumed and produced, the conditions of a transition may also include any Boolean expression on the internal state of the kernel or the values of the tokens it will read. Kernels on this form are executed in steps, as a sequence of transitions. For each step, the conditions are checked to determine which transitions can be made, and then one of the eligible transitions is selected and performed. This family of stream programs is called *dataflow with firing*, and its kernels are called *actors*. The act of making a transition is called a *firing*.

These two kinds of stream program kernels—processes and actors—are very different in how they are described and executed. Nevertheless, they share a common interface—the token streams—that make them interoperable. To get to an efficient implementation, however, the inner workings of the kernels need to be exposed to orchestrate the whole stream program efficiently. This limits the interoperability in practice, because most tools only support a particular style or language.

An example of an efficient implementation technique is static scheduling for *synchronous dataflow* [1]—a restricted class of dataflow with firing. A synchronous dataflow actor has the same token rates on each firing. Each input and output port is associated with a number—the token rate—and all transitions read and write that many tokens on the ports. By looking at the token

rates, a schedule for the actors can be computed a priori, when the program is compiled. This eliminates the need for scheduling decisions at runtime.

This technique is not applicable to dataflow with firing in general, because for different transitions, a kernel may read different numbers of tokens, and the transition selection might depend on the values of the tokens. Nevertheless, it is often possible to create schedules with some decisions being made at compile time and others at runtime [2, 3].

The actor machine [4] is a formal model for dataflow actors with firing, that is explicit about how the transition selection is done—how the conditions of the transitions are checked. The actor machine is also compositional; a network of actor machines can be fused to a single actor machine, creating a sequential implementation of a network. Some scheduling decisions can be made in the fusion, while others can (or must) be postponed to runtime.

THESIS

The research that is presented in this dissertation is about how to efficiently implement stream programs in software.

Thesis: The use of actor machines as kernel representation facilitates the creation of efficient software implementation of stream programs.

The investigation of this thesis has been divided into four aspects.

Analyzability Many efficient implementation techniques for stream programs are only applicable to programs with certain properties, such as synchronous dataflow. The first aspect is the analyzability of actor machines—how actor machines can be analyzed to detect such properties.

Generality Stream program kernels can be written in different styles, with the declarative nature of dataflow with firing on the one hand, and the sequential description of processes on the other. The second aspect is the generality of actor machines; what kind of kernels can take advantage of the techniques developed for the actor machine's analysis and efficient implementation.

Kernel implementation To create efficient implementations of stream programs, the kernels need to be efficiently implemented. In this regard, the actor machine is concerned with the decision process in kernels that can do

different things depending on the situation, i.e. kernels that have more than one transition. The third aspect is how the actor machine can be used to efficiently implement the transition selection process.

Network implementation The efficiency of a stream program implementation also depends on how a network of kernels is implemented. Actor machine composition is a technique for fusing actor machines. The fourth aspect is how actor machine composition can be done to create efficient sequential implementations of networks of kernels.

Chapter 2

Stream Programs

In the introduction chapter, stream programs are described as network-structured parallel programs. The nodes of the network are called *kernels*, but the naming varies between different kinds of stream programs. In dataflow programs with firing, for example, the kernels are usually called *actors*. The networks are sometimes also described differently. They can be referred to as graphs of nodes that communicate over the edges.

This chapter starts with a brief history of stream programs, focused on dataflow with firing. It continues with some background on process models. Finally, the actor machine—the subject of this dissertation—is described in more detail. In the chapter following after this, alternatives to the actor machine are discussed. The four aspects, for which the actor machine is evaluated, are used when discussing the alternative models.

DATAFLOW WITH FIRING

Dataflow with firing is a stream program model that has shown up in different contexts, for example, parallel computing, signal processing and media coding.

Computation Graphs

One of the earliest models of parallel computing is a graph-structured model of stream programs, described by Richard M. Karp and Raymond E. Miller in 1966 [5]. The problem they were facing was that computers started to get parallel processing capabilities, and they wanted to use that to speed up computation. They showed that the programs of their model are determinate, meaning the result of the computation is not affected by variations in speed of the different parts of the program. This property is very useful when the goal is to make some computation faster, since correctness can be verified without taking

timing into account. They also showed how to compute whether a program will terminate, which implies that the model is not Turing-complete. Finally, they studied conditions for when the lengths of the communication queues can be bounded.

A program in this model is described by a directed graph, where each node is associated with an operation and the edges have queues of data items that the nodes read and write. The operation is a function with a fixed number of parameters and a fixed number of result values. On each invocation of a node, data items are first consumed from the incoming edges and then placed as arguments to the function, and when the result has been computed, the values are added to the queues of the outgoing edges. Every edge in the graph is associated with four non-negative integers: A, the number of initial values in the queue; U, how many values are produced by an invocation of the node that the edge starts in; W, how many values are consumed by an invocation of the node that the edge ends in; and T, a threshold of how many items must be in the queue for the node that the edge ends in can be invoked. The threshold must be at lest the number of consumed items, $T \geq W$.

The numbers A, U, W and T are used when analyzing the program, for example, to determine if the program will run forever or stop at some point.

Dennis' Dataflow

In the late 1960s and early 1970s, Jack B. Dennis set the foundation for what today is called *dataflow with firing*. Envisioning a highly parallel computing system, he created a graph-structured program representation to make use of its parallelism [6]. Similar to Karp and Miller's computation graphs, the nodes of these programs are also executed in steps, here called *firings*. One difference from computation graphs, though, is the ability to act on the values of Boolean control tokens. For example, a *T-gate*—an actor with two incoming edges, one for data tokens and one for control tokens, and one outgoing edge—acts whenever there is data on both incoming edges, but forwards the data token to the output only if the control token is *true*. The programs in this model are determinate, i.e. the output of a given program is determined by the input.

This dataflow model was designed to be an intermediate program representation between a procedural programming language and the target computer. By translating a procedure to a dataflow program, the parallelism inside the procedure is exposed. Dennis discusses how some, but not all, concepts of Dijkstra's *structured programming* translate to this dataflow model.

Synchronous Dataflow

Digital signal processing became an important application area for the dataflow model. In the 1980s, Edward A. Lee found that dataflow programs with some restrictions could be scheduled at compile time [1]. Nodes that always consume the same amount of tokens on their incident edges and produce the same amount of tokens on its emanating edges are called *synchronous*. A program with only synchronous nodes is a *synchronous dataflow* program. This model is very similar to the computation graphs of Karp and Miller, but synchronous dataflow restricts the token threshold to be exactly the number of consumed tokens $T = W$.

The scheduler decides on which processor and in which order the nodes are executed. By creating a schedule at compile time, the system does not need to make any scheduling decisions at runtime. Efficient scheduling on both parallel and sequential platforms is an interesting portability aspect of dataflow in general and synchronous dataflow provides an efficient solution.

Cyclo-Static Dataflow

Static scheduling has been extended to a slightly less restricted form of dataflow by Bilsen et al., called *cyclo-static dataflow* [7]. In this model, the number of tokens consumed and produced by a node is allowed to vary over time, but only in fixed cyclic patterns. A node can, for example, split a stream into two streams, forwarding every second token in one direction and the other tokens in another direction. Since cyclo-static dataflow programs can be transformed to synchronous dataflow [8] and synchronous dataflow programs are cyclo-static dataflow programs with cycle length one, they are equivalent in computational power. Cyclo-static dataflow programs, however, allow for a more compact program representation in some cases.

Boolean-Controlled Dataflow

Boolean-controlled dataflow is an extension of synchronous dataflow, introduced by Joseph T. Buck [9]. The extension enables nodes to do two different operations depending on a Boolean-valued token. The two operations may differ in how many tokens they produce and consume on the edges, but every activation of the respective operations must have the same token rates. This is also an extension of Dennis' dataflow, allowing user defined nodes that depend on the Boolean token value.

In his dissertation [9], Buck studies a few subclasses to the class of programs that can be expressed in this model. One of these subclasses is the programs

with a schedule whose length in time can be bounded. This class includes the synchronous dataflow programs with known execution time of the nodes, but also Boolean-controlled dataflow programs with if-then-else-like structures. He also shows that the class of Boolean-controlled dataflow programs, as a whole, is Turing-complete. Many decision problems about the behavior of Boolean-controlled dataflow programs are therefore undecidable.

StreamIt

Dataflow with firing is not just a theoretical model of how programs *could* be described. There are programming languages based on this model as well. StreamIt is a language that is largely based on synchronous dataflow, but with some extensions [10]. The user defined kernels of a StreamIt program are called *filters*. Filters have one input port and one output port. It has built-in kernels for splitting and joining streams. These actors have more inputs or outputs, and describe cyclo-static dataflow rather than synchronous dataflow. Another extension to synchronous dataflow is the ability to look at a fixed number of tokens without consuming them. For scheduling purposes, this extension reintroduces the token threshold from Karp and Miller's computation graphs.

StreamIt also has support for out-of-band communication through *teleports*. They are introduced to be able to update parameters of the filters in a controlled way without loosing the benefits of synchronous dataflow.

Another departure from synchronous dataflow is the ability to have filters with dynamic token rates. This feature has been added to be able to express more complex programs such as an MPEG-2 decoder [11]. However, most optimizations available to StreamIt programs are not available for these filters.

Cal and RVC-CAL

Cal is a language for dataflow actors with firing [12]. It was created by Johan Eker and Jörn W. Janneck as a scripting language for the Ptolemy II project. A subset of the language was later standardized by MPEG under the name RVC-CAL, as part of the reconfigurable video coding standard ISO/IEC 23001-4. Both Cal and RVC-CAL can describe a broad class of dataflow programs, including non-deterministic programs, and programs with data-dependent transition selection.

Several compilers exists for Cal and RVC-CAL, for example OpenDF [13], the Open RVC-CAL Compiler (Orcc) [14], Cal2C [15], CAL2Many [16], and the Tÿcho compiler, which is presented in this dissertation. These compilers can generate code for a broad variety of systems with various degrees of parallelism,

ranging from single-processor systems, multi-core and many-core systems, to FPGAs.

The following program listing defines an actor Filter that takes a predicate p as instantiation parameter.

```
actor Filter (p) In ⟹ Out :
  action In:[x] ⟹ Out:[x] guard p(x) end
  action In:[x] ⟹ guard not p(x) end
end
```

It has one input port, named In, and one output port, named Out, through which it communicates with other actors. The actor contains two actions that describe what this actor can do. Both actions read one token from In and binds its value to a fresh variable x. The first action writes the value of x to the output port, and the second action does not. The guard-expressions restrict the firing of the actions to when p(x) is true and false, respectively.

This actor could, for example, be instantiated with **lambda** (n) : n ≥ 0 **end**. It will, in that case, forward all values that arrive to In that are greater than or equal to zero to the output port Out.

A Cal actor can have internal state variables. The variables are shared between the actions and can be read and updated by them. Cal actors may also have action schedules, which define what sequences of action firings are allowed. Finally, a Cal actor can also define priorities a > b, stating that action a should be prioritized over b in cases where both actions could otherwise be executed.

PROCESS LANGUAGES

Dataflow with firing is not the only family of stream program model. Another family is the family of process languages.

Kahn Process Networks

About the same time as Dennis created dataflow with firing, Gilles Kahn, a French computer scientist, created what today is called *Kahn process networks* [17]. The nodes of these networks are called *processes*, and the edges are communication channels, through which the processes communicate. Both Dennis and Kahn make use of procedures in their models, but in very different ways. While Dennis' idea is to find fine-grained parallelism inside procedures, Kahn uses a procedural language as a sequential description of the processes. Kahn's process

language includes a blocking read statement and a non-blocking write statement for communicating over the channels.

By translating processes to functions from sequences to sequences, Kahn shows that both processes and networks of processes are monotonic on the prefix order. Monotonicity of a process f means that if the input sequence x is a prefix of y, i.e. y starts with the elements of x, then the output of the process $f(x)$ is a prefix of $f(y)$. Monotonicity on the prefix order also implies determinacy irrespective of the order in which the execution of the processes is scheduled.

A language for Kahn processes, but with bounded buffers, is used to program the Ambric Am2045—a many-core architecture with 336 processors.

Communicating Sequential Processes

A process model that can express non-deterministic computation is *communicating sequential processes*. It was introduced in 1978 by C.A.R. Hoare as a model for concurrent computation that is explicit about its concurrency. About the same time, Robin Milner introduced his *calculus of communicating systems* [18]. Inspired by Milner's calculus, S.D. Brookes, C.A.R. Hoare and A.W. Roscoe evolved Hoare's model into a process calculus as well [19]. Hoare's process model has also been the inspiration for several programming languages, most notably occam [20] and the Go programming language.

ACTOR MACHINE

The actor machine is an abstract machine model for dataflow actors with firing. It was introduced in 2011 by Jörn W. Janneck [4]. An actor machine implements one actor and has a finite set T of *transitions* that it can fire. Each transition may (1) read input tokens (2) update the internal state of the actor and (3) write output tokens. Transitions may only be fired when all tokens it will read are present. They can also have other requirements. These requirements are described by a finite set C of firing conditions or just *conditions*, for short.

Controller

The central component of an actor machine is its *controller*. It is a finite automaton that controls the testing of conditions and firing of transitions. The transitions of the controller are called *instructions*, to not be confused with the transitions of the actor. There are instructions of three kinds: TEST, EXEC and

WAIT. The TEST instruction tests a condition and it has two target states, one for each outcome of the test. The EXEC instruction fires a transition of the actor. The WAIT instruction indicates that the actor cannot currently make any progress—it might, for example, need one more token before it can continue.

The states encode knowledge about the results of tested conditions. A common way of representing the knowledge is by a value from the set $\{1, 0, \mathrm{X}\}$, where 1 and 0 indicate that the condition is known to be true and false, respectively, and X indicates that the condition is not known. The controller state can, for example, be represented by tuples $(k_1, k_2, \ldots, k_{|C|})$, where $k_i \in \{1, 0, \mathrm{X}\}$ is the knowledge about condition $c_i \in C$. For example, in state $(1, 1, 0, \mathrm{X})$, conditions c_1 and c_2 are known to be true, c_3 is known to be false, and c_4 is unknown. For brevity, we use juxtaposition to denote tuples, e.g. we let 110X denote the tuple $(1, 1, 0, \mathrm{X})$.

For example, the Filter actor on page 11 can be implemented as an actor machine using three conditions and two transitions. Let condition c_1 be the availability of a token on the In port, c_2 be the availability of space for a token[1] on Out, and c_3 be the expression p(x), where x is the first token on port In. The negated guard expression, **not** p(x), is represented by $\neg c_3$.

In the initial controller state, nothing is known about the three conditions. The initial state is therefore $k_1 k_2 k_3 = \mathrm{XXX}$. In this state, either the token condition c_1 or the space condition c_2 can be tested. The guard condition c_3, however, cannot be tested unless there is a token available, i.e. c_1 is true. The controller states can have several instructions, e.g. state XXX can have one TEST instruction for condition c_1 and another test instruction for condition c_2. When executing the controller, one instruction is selected and executed.

Controllers can be visualized as bipartite graphs with each node being either a state or an instruction. Figure 2.1 shows a visualization of one possible controller. Ellipses represent states and are annotated with the knowledge of the state. Rectangles and diamonds represent EXEC and TEST transitions, respectively, and are annotated with their transition or condition. The WAIT instructions are represented by rings. The TEST instructions have two emanating edges, one with a solid line that is followed when the condition is true, and one with a dashed line that is followed when the condition is false. Note that this example only has one instruction from each state.

[1] In many stream program formalisms, buffers are unbounded, making conditions on space for output tokens always true. Many implementations, however, use fixed-size buffers as an optimization, making the output conditions useful again.

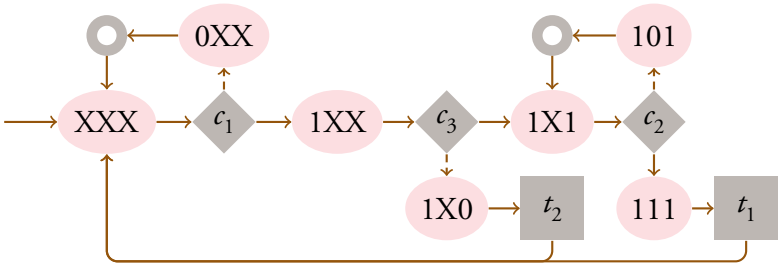


Figure 2.1: An actor machine controller for the **Filter** actor.

Operations on Actor Machines

Actor machine controllers that have states with more than one instruction can be pruned in an operation called *reduction*. In a reduction, any instruction except the last instruction in each state can be removed. States that become unreachable are implicitly removed. When the actor machine only has at most one instruction per state, it is called fully reduced. How the reduction is done can affect the performance of the resulting implementation, as discussed in Paper D.

Another important operation on actor machines is *composition*. When a network of actors are implemented as a sequential program, the actors need to be scheduled. If the actors are represented as actor machines, the schedule and the actors can be implemented as one actor machine by composition. When composing actor machines, the controllers are fused to a single controller that implements all the actors. By incorporating knowledge in the fused controller about how the component actor machines communicate, token and space conditions for the channels that are internal to the composite does not need to be tested.

Chapter 3

Related Work

This chapter discusses other stream program representations and how they relate to the four areas of interest described in the problem statement.

STREAM-BASED FUNCTIONS

The first model that is discussed is *stream-based functions* [21]. This model is designed to express Kahn processes as dataflow with firing. The core components of stream based functions are *objects* and *channels*. Objects describe stream program kernels, and channels are connections through which the kernels communicate.

An object consists of three parts: a set P of *step functions*, a *controller* and *state*. Each step function describes a step of execution, the controller determines which function should be invoked next, and the state evolves throughout the execution by the step functions and the controller. The state is divided into two separate parts: the data state D and the controller state C.

A step function $f \in P$ accept input tokens $x_1, \ldots, x_m$ and the current data state $d \in D$ as parameters and returns output tokens $y_1, \ldots, y_n$ and an updated data state $d' \in D$. The mapping between input ports and input parameters, as well as between return values and output ports is fixed for each step function, but may differ among the functions. When making a step using function $f \in P$, the tokens it require are consumed from the input ports and put as parameters to the function together with the current data state. The result tokens of the function are sent to the output ports of the object, and the returned state is the new state of the object that will be used as input for the next step function.

The controller determines, based on the current controller state $c \in C$ and the data state $d \in D$, which step function $f \in P$ should be invoked next. It

consists of two functions, a transition function and a binding function. The binding function $\pi : C \to P$ returns a step function given a controller state. The transition function $\omega : C \times D \to C$ is applied after each step function is applied. It returns a new controller state based on the current controller state and the returned data state of the step function that was just applied.

An example execution starts in the initial controller state $c_1 \in C$ and data state $d_1 \in C$. The binding function is applied to determine which step function to apply $\pi(c_1) = f_a$. Then the tokens that f_a requires are consumed from the input and put as arguments to the function together with the current data state, $f_a(x_1, \ldots, x_m, d_1) = (y_1, \ldots, y_n, d_2)$. The output tokens $y_1, \ldots, y_n$ are sent to their respective output ports. The transition function is applied $\omega(c_1, d_2) = c_2$ to determine the new controller state. Now the procedure starts again by applying $\pi(c_2)$.

Stream-based functions are used in Compaan—a compiler that transforms programs described as nested loops in Matlab to Kahn process networks, expressed as stream-based functions [22].

Analyzability Stream-based functions are designed to be simpler to analyze and transform than Kahn processes. An object with only one step function, for example, is a synchronous dataflow actor. It is, however, not as simple to detect if an object is a cyclo-static dataflow actor. The transition function needs to be analyzed to determine whether the state selection is data dependent. The analyzability regarding cyclo-static dataflow therefore depends on how the transition function is expressed.

Generality As mentioned earlier, stream-based functions can represent Kahn processes and can therefore express any deterministic computation. They cannot, however, describe non-deterministic stream program kernels.

Kernel implementation The selection of step function is effectively performed by the transition function when computing the next controller state. The actual selection is then just a simple mapping from controller state to step function. This model neither imposes an inefficient kernel implementation or hinders the creation of an efficient one, regarding transition selection.

Network implementation Stream-based function objects are not composable in general. There are, however, situations when objects can be com-

posed [21]. In the context of Compaan, this model also offers decomposition and other transformations of objects, described in [23].

FUNCTIONAL DATAFLOW INTERCHANGE FORMAT

The formal foundation of the *functional dataflow interchange format* is based on a model of dataflow with firing called *enable-invoke dataflow* [24]. An actor in this model has a set of *modes*, and, at any point in time, a subset of these are *active*. The actor is defined by two functions; an *enabling function* and an *invoking function*. The enabling function takes as parameters the number of input tokens that are available on the input ports and a mode and returns whether the actor can be invoked in that mode with that many input tokens. A runtime system for such actors can check the enabling function for all active modes to see if the actor can be invoked. The invoking function takes as parameters the values of the input tokens and one of the active modes for which the enabling function returned true. It returns a new set of active modes, together with the output tokens that are produced.

Enable-invoke dataflow is a model that can represent a broad class of dataflow with firing. It can for example represent the Turing-complete boolean controlled dataflow. The actors can also be non-deterministic. It cannot, however, represent all Cal programs. The enabling function considers the mode and the number of tokens, but Cal actions can be conditioned on the values of the tokens using guard expressions.

The *functional dataflow interchange format* (functional DIF) is based on a restricted version of enable-invoke dataflow called *core functional dataflow*. It is restricted to one active mode, that is describing only deterministic actors. Functional DIF is implemented in the DIF package—a Java library for prototyping an analyzing stream programs.

Here follows a short analysis of the functional dataflow interchange format. It is analyzed using the four areas of interest described for the actor machine in the problem statement of this dissertation.

Analyzability The dataflow interchange format has different constructs for each subclass of dataflow actors that it supports—one for core functional dataflow, one for cyclo-static dataflow etc. If an actor is expressed as core functional dataflow, it is in general hard to determine if this actor is a cyclo-static dataflow actor. It requires analysis of what mode the invocation function returns; in particular, that the returned mode only depends

on the mode parameter. This is not feasible in general, but the invocation function could be expressed in a way that makes it easier to prove.

Other actor variants of the dataflow interchange format are, however, simpler to analyze. A cyclo-static actor explicitly provides lists of the token production and consumption patterns [25].

Generality The functional dataflow interchange format can express boolean controlled dataflow and can thus express any deterministic computation. It cannot express non-determinism, even though enable-invoke dataflow can.

Kernel implementation The model is carefully designed to not require any re-computation between the enabling and invoking function. The enabling function tests whether the actor is enabled in the given mode for the number of available tokens. The invoking function is then not allowed to check how many tokens are available.

Network implementation In [26], new scheduling approach for core functional dataflow is presented. It uses the modes of the actors to deconstruct the dataflow graph to a set of dynamically interacting synchronous dataflow graphs. The schedule is described as a generalized schedule tree [27]. This approach provides great speedups in some cases compared to a round-robin schedule.

SYSTEMOC AND ACTOR FSM

SysteMoC is a SystemC library for describing dataflow actors with firing [28]. The model that SysteMoC implements is based on FunState (functions driven by state machines) [29]. An actor in SysteMoC consists of input and output *ports*, *functionality*, and a *firing finite state machine*. The functionality is a set of *actions* (actor transitions) and *guards* (conditions on the state or the input token values). The firing finite state machine controls when the actions can be fired. Each transition is annotated with an action and with the conditions for firing that action.

The performance of this library compared to normal SystemC threads has been tested using a two-dimensional IDCT. The comparison was performed on a single-core personal computer, resulting in a speed-up factor of 1.3 for SysteMoC.

In [30] and [31], a similar actor representation is described, and its use is extended to normal SystemC actors, i.e. not the ones described with the SysteMoC library. The control flow graphs of the SystemC methods are modified. The basic blocks are split such that all read statements precede the write statements. The actor finite state machine is then constructed from this modified control flow graph. This transformation is essentially a transformation from processes to dataflow with firing.

Analyzability The definition of the firing finite state machine in the SysteMoC actor code is designed to be easily extracted. This state machine can then be analyzed to detect synchronous dataflow, cyclo-static dataflow and others. Compared to stream-based functions and core functional dataflow, this model is more explicit about the action selection, and therefore simpler to analyze.

Generality The SysteMoC library can represent very broad range of dataflow with firing and the action selection can be, and have to be, tailored to fit the actions. The translation from processes, described as SystemC actors, to a similar model also shows the generality of this kind of model.

Kernel implementation The explicitness about the conditions is a good starting point for efficient kernel implementations with optimized action selection. The model is sufficiently explicit, for example, to generate an actor machine from.

Network implementation Efficient network implementations are achieved by classifying the actors, and then clustering the actors into statically schedulable regions. A SystemC implementation of a Motion-JPEG decoder has been evaluated for this model. It showed a speed-up factor of 2.2 for the whole decoder and 4.5 for the two-dimensional IDCT part of the program, compared to the dynamically scheduled SystemC program.

Chapter 4

Contributions

In my research, I have investigated the use of actor machines as kernel representation for creating efficient software implementation of stream programs. I have identified four important aspects of this investigation: (1) the analyzability of kernels, (2) the generality in what it can express, (3) implementation efficiency for kernels and (4) implementation efficiency for networks. My contributions regarding those aspects are the following:

Analyzability In Paper B, I have described and implemented an actor classifier that detects synchronous dataflow, cyclo-static dataflow, prefix monotonicity (Kahn processes) and determinacy. This paper shows the analyzability of the actor machine.

Generality In Paper E, I have described and implemented a translation from Kahn processes to Cal, and in Paper A, a translation from Cal to actor machines. These papers, together, demonstrate the generality of the actor machine—that it can express both processes and very general dataflow with firing.

Kernel implementation Transitions and conditions of stream program kernels are described using normal procedural and functional languages. These parts are very important to optimize to efficiently implement kernels. Since these parts are similar to conventional programming languages, optimizations for them are already widely available. Optimizations for transition selection, however, is a less explored area or research. In Paper D, I have described how to create efficient transition selectors with actor machines, using profiling based heuristics. This paper shows that the transition selection can be optimized using actor machines.

Network implementation There are several aspects of the implementation of stream program networks that are important for its efficiency. If the target for the implementation is software, one important aspect is the scheduling of the kernels for the available processors. Actor machines can implement schedules by composition, i.e. fusion of two or more actor machines into one that performs the computation of all its component actor machines. I have in Papers C and F contributed two algorithms for actor machine composition.

INCLUDED PAPERS

The following is a description of all papers that are included in this dissertation; what the papers contribute to the research area, and my contributions to the papers.

"Toward Efficient Execution of Dataflow Actors" (Paper A)

Gustav Cedersjö and Jörn Janneck

This paper introduces a translation from Cal actors to actor machines. In [4], where the actor machine is introduced, Cal is used as an example language, but no complete translation is described. Our translation supports both the action schedules and the action priorities of Cal. It is compared to a traditional way of implementing Cal actors by the number of test instructions that are performed when executing a program. The comparison is done using a video decoder consisting of 11 actors. I am the main author of this paper and I have done the technical work.

"Actor Classification using Actor Machines" (Paper B)

Gustav Cedersjö and Jörn Janneck

This paper is an investigation in the analyzability of actor machines. It provides definitions for classifying actor implementations represented as actor machines. The classification can, for example, be used to decide how the actor scheduling should be done. Similar classifiers exist for other actor representations. I am the main author of this paper and I have done the technical work.

"Software Code Generation for Dynamic Dataflow Programs" (Paper C)

Gustav Cedersjö and Jörn Janneck

Actor machine composition is a technique for scheduling networks of actors by fusing them to a single kernel [4]. This paper presents the first implementation

and evaluation of actor machine composition. It also presents a software implementation strategy for the actor machine controller. I have done the compiler implementation and the evaluation for this paper. Janneck is the main author of section C.1 to C.5, and I am the main author of the rest of the paper.

"Finding Fast Action Selectors for Dataflow Actors" (Paper D)

Gustav Cedersjö, Jörn Janneck and Jonas Skeppstedt

Actor machine reduction is a refinement of the action selection process of actor machines. A reduction takes as input an actor machine that can do one of potentially several things in each state, and produces as output one that can only do one thing in each state. The selection of what to do in each state affects the performance of the resulting program. This paper introduces two reduction heuristics that use platform independent profiling information as the basis for the reduction decisions. I am the main author of the paper and I did the technical work.

"Processes and Actors: Translating Kahn Processes to Dataflow with Firing" (Paper E)

Gustav Cedersjö and Jörn Janneck

This paper presents a Kahn process language that is designed to work well together with Cal. It also introduces a way of reasoning about action firings in the semantics of Kahn processes. This is the basis of a translation from Kahn processes to dataflow with firing, that is also presented in the paper. I have designed the process language and its implementation by translation to Cal. I have also found the way of reasoning about action firings in the semantics of Kahn processes. I am also the main author of the paper.

"Tÿcho: A Framework for Compiling Stream Programs" (Paper F)

Gustav Cedersjö and Jörn Janneck

This paper presents the Tÿcho framework and stream program compiler. Tÿcho is a framework for building stream program compilers based on the actor machine. It contains an example compiler with frontends for Cal and the Kahn process language described earlier. The paper also presents a novel actor machine composition algorithm. I have implemented most of the framework and the compiler and created the composition algorithm. I am also the main author of the paper.

RELATED PAPERS

I have contributed to other papers about stream programs and compilation techniques.

"Realizing Efficient Execution of Dataflow Actors on Manycores" [32]

Essayas Gebrewahid, Mingkun Yang, Gustav Cedersjö, Zain Ul-Abdin, Veronica Gaspes, Jörn W. Janneck and Bertil Svensson

This paper presents a compiler from actor machines, via *action execution intermediate representation* to three different targets. One backend generates sequential C code. Another backend generates code for the Epiphany platform from Adapteva. A third backend generates code for the Ambric platform. I contributed minor technical work to the compiler.

"JavaRAG: A Java Library for Reference Attribute Grammars" [33]

Niklas Fors, Gustav Cedersjö and Görel Hedin

This paper presents a library for reference attribute grammars in Java, called JavaRAG. Attribute grammars were introduced by Knuth [34] as a declarative way of computing attributes on trees. Reference attribute grammars is an extension by Hedin [35] that allow attributes to be references to nodes in the tree. I have built the library that is presented in this paper, and have used it in the Tÿcho compiler and the actor classifier.

"Dataflow Machines" [36]

Jörn W. Janneck, Gustav Cedersjö, Endri Bezati and Simone Casale Brunet

Actor machines are designed for software implementations of dataflow programs. The controller is a sequential program for selecting transitions. The *dataflow machine* replaces the controller of the actor machine with a more parallel transition selection mechanism, called the *selector*. This model is designed to be a better starting point for hardware implementations of stream programs. I contributed to the discussions that lead to this model.

"Mapping and Scheduling of Dataflow Graphs—A Systematic Map" [37]

Usman Mazhar Mirza, Mehmet Ali Arslan, Gustav Cedersjö, Sardar Muhammad Sulaman and Jörn W. Janneck

This paper is a systematic map of research on mapping and scheduling of dataflow graphs. The paper concludes that most research is done on static scheduling and the most used optimization goals are memory usage and throughput. I contributed to the study by reading and mapping some papers.

Chapter 5

Conclusions

This dissertation investigates the thesis that the use of actor machines as kernel representation facilitates the creation of efficient software implementation of stream programs.

CONCLUSIONS

As shown in [37], a lot of research has been conducted on static scheduling of dataflow programs. Actor machines, however, cannot in general be statically scheduled. Nevertheless, if a network (or subnetwork) of actor machines is classified to be statically schedulable, a lot of research results becomes available for the actor machine. This dissertation shows that the transition selection that actor machines describe is sufficient for implementing such classifier.

Even though statically schedulable subclasses of dataflow is popular target for a lot of research, not all programs can be expressed using those classes of dataflow. Cal is an example of a language that can express a broad range of stream programs. It can describe non-deterministic actors, and actors that act on the absence of input tokens. It is also an important language, because of its use in the reconfigurable video coding standard ISO/IEC 23001-4. This dissertation shows that the actor machine is general enough to describe Cal actors. Dataflow with firing is not the only way to describe stream programs. Kahn processes is an appealing alternative with its simplicity of programming. This dissertation shows that even Kahn processes, using a translation via Cal, can be described using actor machines.

For simple actors, such as those that can be scheduled statically, it is easy to determine what the actor should do next. For more complex actors, such as Cal actors that have more than one candidate transition in a given state, the transition selection requires some more work to determine which transition to

execute. This dissertation shows that the actor machine facilitates creation of efficient transition selectors for the more complex actors.

When stream programs are implemented in software, the kernels need to be scheduled on the available processors. One way of implementing a schedule is by kernel fusions, i.e. fusing two or more kernels into one that performs the computation of its component kernels. This dissertation shows that kernel fusion through actor machine composition can implement efficient static and semi-static schedules for stream programs. However, due to the nature of the composition problem, there is a risk of state space explosion that needs to be avoided. One direction of future research to make actor machine composition more practical, is the development of heuristics for determining which actor machines could efficiently be composed. Another direction is a trade-off between runtime tests and program size, by representing less knowledge about the queue lengths between the component actor machines.

FUTURE WORK

The target of this research has, at least partly, been towards software implementations when there are fewer processors than the number of kernels. With the emergence of low-power many-core platforms, with a thousand processors or more, the problem descriptions for efficient software implementations radically change. What will be the challenges of these platforms? Will the platforms become more homogeneous or heterogeneous—will this kind of chip replace other specialized chips or will it become just one more specialized chip? Where will the bottlenecks be and how could they be avoided?

One research direction is to make the use of actor machines broader. An obvious topic in that direction would be to lift the restriction of a fixed network topology. Could the actor machine composition be done as an optimization in a just-in-time compiler? The research could also be taken to other programming models. Can only stream programs benefit from the actor machine, or are there other programming models that can efficiently be implemented using actor machines?

A third direction for this research is to apply ideas of the actor machine to other models. For example, what would be the equivalent of actor machine composition look like for enable-invoke dataflow or a finite state machine based model?

These three directions are somewhat orthogonal and could be combined. For example, develop a virtual machine and just-in-time compiler for many-core platforms with a kernel fusion optimization based on a different model with support for a dynamically changing network.

Part II

Publications

Paper A

Toward Efficient Execution of Dataflow Actors

Gustav Cedersjö and Jörn W. Janneck
Department of Computer Science, Lund University

ABSTRACT Dataflow descriptions are a natural match to application areas such as signal processing, cryptography, networking, image processing, and media coding. This paper addresses the problem of efficiently executing the basic elements of a dataflow program, its *actors*, written in a language such as MPEG's RVC-CAL. Using actor machines as an execution model for dataflow actors, we devise a metric for measuring the quality of a translation in terms of program size and execution efficiency, and then build, evaluate and compare a number of translators with each other and prior art, using MPEG reference code as a benchmark.

A.1 INTRODUCTION

As part of the recent shift toward parallelism, interest in parallel programming models such as dataflow has increased considerably, especially in application areas such as signal processing, cryptography, networking, image processing, and media coding. For instance, MPEG and ISO standardized a subset of the Cal dataflow language [12], called *RVC-CAL* in ISO/IEC 23001-4. These languages embody a computational model known as *dataflow with firing* [38]: the

computational kernels (*actors*) that constitute the dataflow program execute by making a sequence of *steps* or *firings*, consuming input, producing output, and modifying their internal state if they have any.

More general and expressive dataflow languages, such as Cal/RVC-CAL, allow programmers to write actors whose behavior may depend on the input data and their internal state, and may even be non-deterministic. These actors are described as a collection of *actions*, i.e. possible steps the actor can execute, along with the *conditions* that need to be satisfied so that an action may be executed. These conditions include the availability of input data, as well as *guards*, boolean expressions on the value of input data and/or the actor state. The program in Listing A.1 is an example of such an actor written in Cal.

```
actor Split () A ⟹ P, N:

A1: action A: [v] ⟹ P: [v]
    guard v ≥ 0
    end

A2: action A: [v] ⟹ N: [v]
    guard v < 0
    end
end
```

Listing A.1: A simple Cal actor with multiple actions and guards.

Executing an actor like this involves alternating between choosing the next action to be executed, and then executing that action. The action selection phase consists of testing sufficiently many conditions to be able to identify at least one action that can be executed. Since in general several conditions may be testable at any point during action selection, the choice of the condition that is tested next may affect how many conditions will have to be tested before the next action is fired. Consequently, the way that choice is made will have an impact on the efficiency of the actor execution—fewer tests per action execution mean less effort per unit of useful work.

In [4] *actor machines* are presented as an execution model for actors, that makes explicit the testing and action execution steps that need to be performed during actor execution. However, while the model is clearly expressive enough to represent Cal actors, no explicit translation from Cal to actor machines is given. The purpose of this paper is to investigate this translation in more detail.

It turns out that the same Cal actor can be translated into several actor machines that differ from one another in relevant ways, such as their size (which affects the size of the code a compiler would produce when generating code for the actor machine), and the number of tests they perform for a given sequence of action executions (which is a measure for the efficiency of the generated code).

In this paper we devise a family of translations from Cal to actor machines, and explore their properties with respect to the metrics above, using examples from MPEG's reference code as benchmarks. We also explore the design space of a choice heuristic that is used when generating the different translations in this family.

A.2 RELATED WORK

Cal2C is a C code generator for RVC-CAL described in [15]. The code for action selection that Cal2C generates first tests which actions that have all conditions satisfied. Then among these actions, one of the highest prioritized actions is selected.

In [39] this selection process is improved by, at compile time order the actions by priority, and at runtime test the conditions for the actions in this order, starting with the highest prioritized action. The first action to have all its conditions satisfied is selected without testing the rest of the actions. The Cal simulator in OpenDF uses the same approach. We implemented this selection process for actor machines as a baseline in our comparisons.

Our approach to the testing of conditions is similar to what has been done with compilation of pattern matchers. A pattern matcher has a sequence of patterns and takes a value as input, the first pattern to match that value is selected. These patterns can be constructed out of simpler parts, and some parts can be common across the patterns. If the patterns are tested one by one, common parts of the patterns will be tested repeatedly. In [40], Augustsson describes the compilation of pattern matching in LML that avoids this repeated testing by matching the patterns part by part instead of pattern by pattern.

A.3 BACKGROUND

Before describing the translations from Cal, we first need some background in actor machines. An actor machine consists of actions, communication ports, state variables and a controller. Actions can read a fixed number of tokens from

the input ports, change values of the state variables and write a fixed number of tokens to the output ports. Each action is associated with a set of conditions that must be satisfied in order to be fired.

The controller of an actor machine is a state machine responsible for selecting which action to execute depending on the conditions of the actions. Figure A.1 is a graphical representation of a controller. Each state in the controller (the circles or ellipses) is associated with a set of instructions that can be executed in that state. There are three kinds of instructions:

test (diamond shaped),

wait (annular) and

call (rectangular).

The test instruction will test a condition and go to one of two states depending on the outcome of the test. A sequence of tests is performed atomically with respect to the state of the input ports. The wait instruction breaks the atomicity of a test sequence and proceeds to a new state. Call instructions execute an action and also break the test sequence atomicity.

An actor machine is executed by selecting one of the instructions in the current state, executing that instruction and changing current state to the destination state of the instruction.

In Cal, the action selection is not determined by the conditions of the actions alone. A Cal actor can also contain an action schedule and action priorities. The action schedule determines in which order actions can be fired. At each point in the schedule, a set of actions are available for selection. The schedule also defines where the schedule continues depending on which action is selected. The priorities set a partial order on the actions such that an action can not be selected unless all its higher prioritized actions have been disabled by some unsatisfied condition. Both action schedules and priorities can be encoded in the controller of an actor machine.

A.4 OUR WORK

We developed a number of translators from Cal to actor machines with different properties.

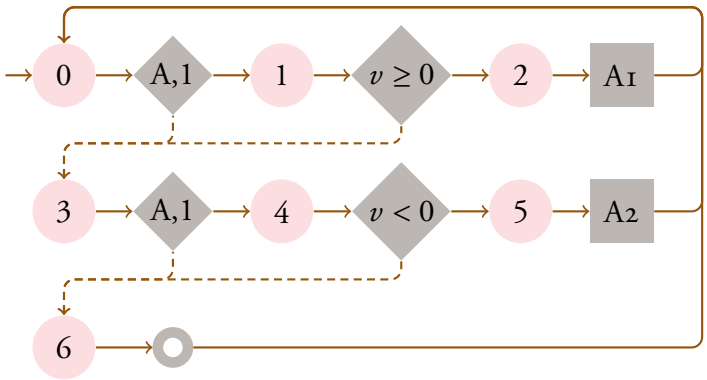


Figure A.1: Controller of the Split actor translated with RoundRobin.

A.4.1 RoundRobin

One of the translators we built, *RoundRobin*, mimics the behavior of traditional code generators, such as the one described in [39], mentioned in section A.2. The resulting actor machine has at most one instruction in each controller state, i.e. a *single-instruction actor machine* (SIAM). This is the baseline in the comparisons of the translators. Figure A.1 shows the controller of the Split actor in Listing A.1 translated to an actor machine with the RoundRobin translator.

An actor machine generated by the RoundRobin translator tests the conditions of one action at the time. If a test fails, the actor machine continues with the tests of another action, but if all tests of an action succeed, that action is fired and the actor machine goes back to the initial state of the controller. This way, all actions will have their conditions tested, and if no action can be selected, the actor machine will execute a wait-instruction and try again from the start.

For Cal actors that have an action schedule, the RoundRobin translator constructs one controller for each point in the schedule. The controllers are then connected by changing the destinations of the call instructions to the state where the controller of the next point in the schedule starts.

Action priorities are handled by ordering the actions such that the highest prioritized action is tested first. This will work because of the atomicity of test sequences; the conditions of all actions will be tested with the same view on the input ports.

A.4.2 MemorizeTestMIAM

One problem with the RoundRobin translator is that some tests are performed repeatedly when they can not change. From Figure A.1 we can see that the availability of one token on port A must be tested twice to be able to select action A2.

MemorizeTestMIAM is a translator that tries to address this problem by memorizing previous test results. It produces actor machines with many possible instructions in each state, called *multi-instruction actor machine* (MIAM). The results of this translator preserve the generality of the input specification, at the expense of size. Also, even when the actor itself is deterministic, the MIAM resulting from its translation may be executed in many different ways.

Controller states. To memorize the test results, let each state in the actor machine controller represent information about the conditions of the actions. A condition is either satisfied (1), unsatisfied (0) or unknown (X), depending on if it is tested to be true, false of if it has not been tested yet. In the example in Listing A.1, the two actions have three conditions in total:

- one token on port A
- that token is positive
- that token is negative.

A state is in this case represented by a tuple $I \times I \times I$, where $I = \{1, 0, \mathrm{X}\}$ is the information about a condition. The meaning of state $(1, \mathrm{X}, 0)$, for example, is that the first condition is satisfied, the second unknown and the third is unsatisfied. In the rest of the paper, we drop the commas and parentheses when describing a state, e.g. $(1, \mathrm{X}, 0)$ is written 1X0.

Figure A.2 shows the controller of the Split actor translated with MemorizeTestMIAM. The states nodes (ellipses) in the figure are annotated with the information about the conditions. The first part of the tuple is the port condition, the second part is if the token is positive, and the last part is if the token is negative.

Generation of controller. The controller is generated incrementally from a queue of states to be processed. Initially, the queue contains only the start state of the controller, which is the state where all conditions are unknown, XX...X.

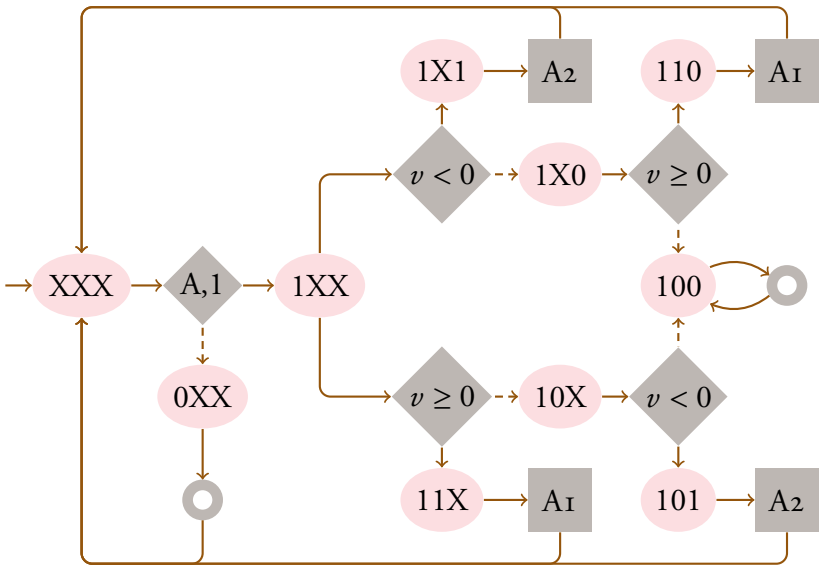


Figure A.2: Controller of the Split actor translated with MemorizeTestMIAM.

If an action can be selected based on the information that a state provides, there is no meaning in testing more conditions. In this case MemorizeTestMIAM will add call instructions to that state of the controller for every action that is enabled, as shown in Figure A.2 in state 1X1, 101, 11X and 110.

If more conditions need to be tested in order to select an action, then the translator will try to add tests to the current state. Actions that are disabled in that state do not need any further testing, therefore only tests for actions that are *not* disabled in current state are added. However, test instructions can depend on each other, e.g. there must be a token available to check if that token is positive or negative. MemoriseTestMIAM solves this problem by only adding tests for guard conditions of an action when all port conditions of that action are satisfied. In state 1XX in Figure A.2, the port conditions for action A1 and A2 are satisfied, hence, tests for the guard conditions are added to this state.

If no call or test instruction has been added, the translator will add a wait instruction that will throw away information about which tokens that are not yet available.

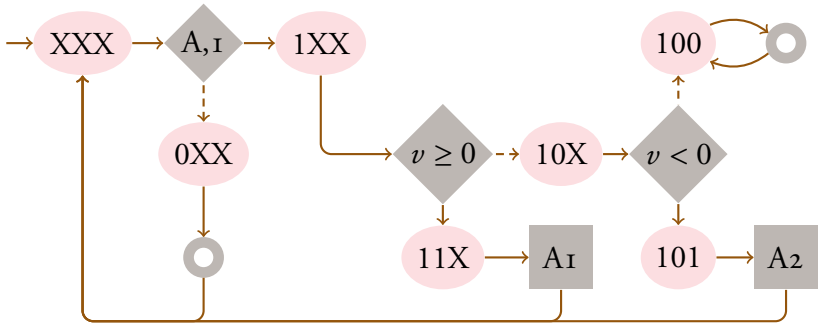


Figure A.3: Controller of the Split actor translated with MemorizeTestSIAM(*first*).

Calculation of destination states. When an instruction is added to a state in the controller, the destination states of that instruction should reflect the information that the execution of the instruction gives or removes. Also, the destination states should be added to the processing queue to be able to build the whole controller.

Test instructions have two destination states, and the information that these states represent should reflect the information gained by the test. Some conditions imply other conditions, for instance, the presence of two tokens on a port implies the presence of one token on that port, and the absence of one token implies the absence of two tokens. MemorizeTestMIAM handles these cases, but does not check if guard expressions are negations of each other.

When a call instruction is executed, the information about the ports that the action reads from is no longer valid, since those tokens are consumed. The guards can in general also change value after an action is fired. All conditions are therefore set to unknown in the destination state of a call instruction.

For the destination states of the wait instructions, information about absent tokens is removed. Since tokens can not be consumed unless an action is fired, information about present tokens as well as guards is retained. If no information is removed, the actor machine will not be able to select any more actions. This is the case for state 100 in Figure A.2, but to come to this state, both $v \geq 0$ and $v < 0$ need to be false at the same time.

Action Schedule and Priorities. Action schedules are handled by extending the information associated with the controller states with the points in the action schedule. I^n is extended to $I^n \times S$, where S are the points the action

schedule. A state in a controller with $n = 3$ conditions and a schedule point *start* could look like $(1, X, 0, start)$. The destination state of a call instructions reflects the change in the schedule.

When it comes to priorities, this affects the order in which conditions are tested. When choosing which tests to add to a state, the tests for the highest prioritized among the not disabled actions are added. Call instructions can of course only be added where all higher prioritized actions are disabled.

A.4.3 MemorizeTestSIAM

A MIAM can be reduced to a SIAM, by selecting one instruction in each state of the MIAM controller. In practice, the choice of instruction sets the order in which the tests are performed and this choice affects the amount of tests needed at runtime.

The actor machine controller in Figure A.2 has two possible instructions in state 1XX, but otherwise just a single instruction in each state. Hence, there are only two possible reductions of this particular actor machine and Figure A.3 shows one of them.

MemorizeTestSIAM(h) is a translator that use MemorizeTestMIAM to get a MIAM, and a heuristic h to choose an instruction in each state. A very simple choice heuristic, *first*, always picks the first in the list of instructions in each state.

To explore the design space of the choice heuristic, we generated reductions by picking instructions randomly in each state.

A.5 EVALUATION

For the evaluation of the translators, we used an MPEG-4 Part 2 SP decoder written in RVC-CAL with a short video file as input. The decoder consists of 11 actors and the whole application has two additional actors, one for feeding the decoder with input and one for getting the output. The input video is three frames of a sample video (foreman).

To get an idea of the size of the generated code, we measured the number of states in the controllers generated by the translators.

The efficiency of the translations was evaluated by counting the number of tests needed to decode the video. First, the original Cal actor network was executed (using [41]) to get a causation trace with the order in which the actions were fired. The actor machines were then simulated with the information from this trace. The result of each test instruction in the simulation was determined

Table A.1: Number of states in the controllers.

	RoundRobin	MemorizeSIAM(*first*)	MemorizeMIAM
iq	4	5	8
iDcPred	36	29	68
iAcPred	6	9	48
idct2d	4	5	8
parseheaders	557	852	1203
mvrecon	32	28	28
interpolation	4	5	8
add	28	26	29
is	4	5	8
blkexp	12	10	23
framebuf	37	63	63

by the number of tokens on the channels in the network, and the execution sequence from the original execution.

To evaluate the performance of different reductions from MIAM to SIAM, we made random reductions by picking an instruction at random in each state of the MIAM and simulated the resulting SIAM.

A.6 RESULT

The sizes of the controllers generated by the different translators are shown in Table A.1. MemorizeTestSIAM(*first*) generated on average 43% larger controllers than RoundRobin. In two cases, mvrecon and framebuf, the actor machines resulting from MemorizeTestMIAM were already SIAM. This is the reason why the controller size was not reduced in MemorizeTestSIAM(*first*).

As shown in Figure A.4, the actor machines from MemorizeTestSIAM(*first*) required fewer or the same amount of tests compared to the actor machines from RoundRobin. For the whole sequence the memorizing actor machine required 38% fewer tests than the baseline.

To explore how different choice heuristics affects the performance, random reductions of the MIAMs were compared to each other. For Figure A.5 we generated 1000 random reductions from MIAM to SIAM of the whole application and simulated them. However, only the actor parseheaders showed any difference in performance among the generated actor machines. The bars of

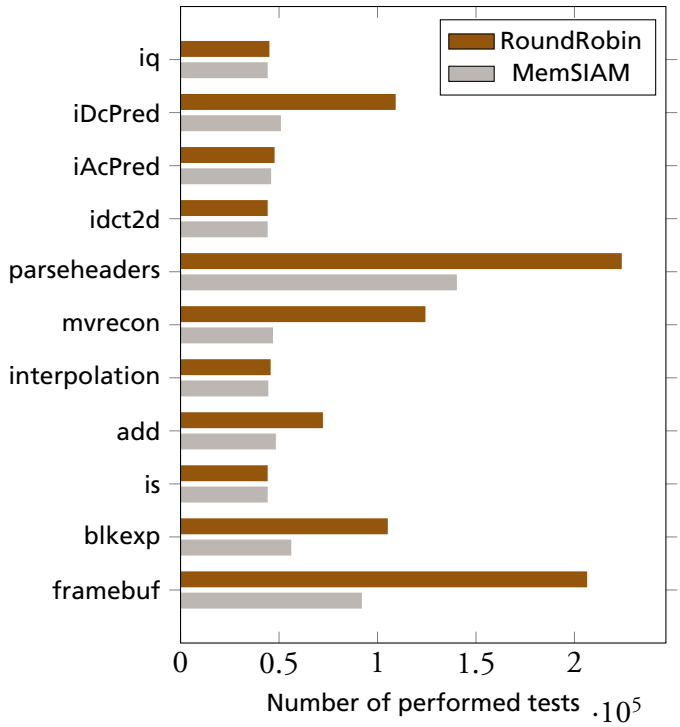


Figure A.4: Comparison of the number of tests for all actors in the test application between the translators RoundRobin and MemorizeTestSIAM(*first*).

the histogram shows how many of the parseheaders actor machines among the 1000 generated ones, that fall into that interval in number of tests needed by the actor machine to decode the test video. The actor machines that in the leftmost bar needed the fewest tests.

A.7 FUTURE WORK

There are many ways to expand and build upon the work presented here. One natural direction to continue is to find heuristics that yields efficient controllers in the reduction from MIAM to SIAM, such the ones to the left in the histogram in Figure A.5. Another interesting direction could be to do analysis of the guard expressions to see if one test can give answer to several conditions. In the example in Figure A.3 this would result in the failure-branch of the test $v \geq 0$ going directly to state 101.

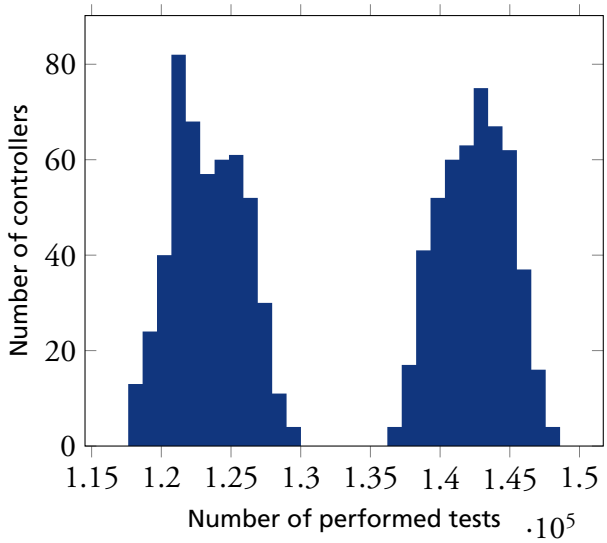


Figure A.5: Histogram over number of tests performed in parseheaders for 1000 random reductions from MIAM to SIAM.

Since this is a first step toward efficient execution of dataflow actors, it would be interesting to generate software from these actor machines and compare the performance to prior art. Composition of actor machines, as described in [4], is also an interesting direction for this work.

A.8 ACKNOWLEDGMENT

This work has been supported by the strategic research area ELLIIT.

Paper B

Actor Classification using Actor Machines

Gustav Cedersjö and Jörn W. Janneck
Department of Computer Science, Lund University

ABSTRACT Program analysis is an important tool in software development, both for verifying desired properties and for enabling optimizations. For dataflow programs, properties such as determinacy and static schedulability are important for verifying correctness and creating efficient implementations.

In this paper we develop an analyzer for dataflow actors, the computational units of a dataflow program, that classifies actors based on these properties. The analysis is performed on a language independent model for dataflow actors called *actor machine*.

B.1 INTRODUCTION

Testing and verification are integral parts of software development. Some aspects of a program can be verified automatically through static analysis of the program structure, a notable example being type systems. This paper presents a classifier for a certain kind of programs, called *dataflow programs*, based on how different parts of the program, called *actors*, communicate. The idea is that the programmer knows of which classes the actors should be, and this tool statically checks if that is actually the case.

The classifier is based on an abstract model for actors called *actor machines* [4], and the classes that are recognized [17, 42, 43] are well known and well studied. The main contributions of this paper in not primarily the classifier tool, but a way of reasoning about the communication behavior of actor machines.

This paper continues by describing dataflow programs and actor machines in section B.2 and B.3. The actor classes and their definitions on the actor machine are presented in section B.4. Finally section B.5 comments on the tool implementation, section B.6 on related work, and section B.7 concludes this paper.

B.2 DATAFLOW PROGRAMS

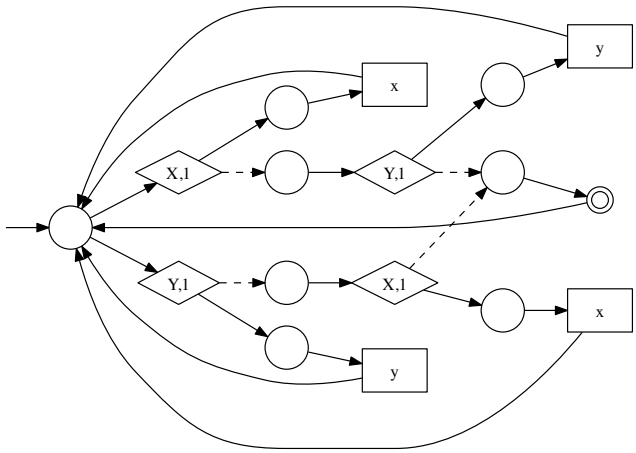
A dataflow program is a network of computational units called *actors*, that communicate by sending data packets, *tokens*, over channels. Actors have *input ports* and *output ports* on which they read and write tokens. Each channel in the network graph connects one output port to one input port, and delivers all tokens in first-in-first-out order.

Internally, actors have a set of *actions* that carry out the actual computation by reading and writing tokens on the ports and updating the internal state of the actor. The *token rate* is the number of tokens that an action reads and writes on the ports. Each action has a fixed token rate, but different actions in the same actor may have different token rates.

The *actor state* is defined as the values of its state variables and the tokens on its input ports. The *enabling conditions* of an action are the conditions an action pose on the actor state to be enabled for execution. Two kinds of conditions exist: a *port conditions* is a condition on the length of an input sequence, and a *predicate condition* is a predicate on the values of the state variables and the input tokens. The actor execution alternates between testing conditions until at least one action is enabled, and executing one of the enabled actions.

B.3 ACTOR MACHINE

The *actor machine* [4] is a machine model for dataflow actors that focuses on minimizing the overhead of action selection. Actor machines are used in [44] to reduce the runtime testing of conditions, and [4] shows how actor machines can be composed to eliminate testing of port conditions.




```
actor Merge () X, Y ⟹ Z :
  x: action X: [v] ⟹ Z: [v] end
  y: action Y: [v] ⟹ Z: [v] end
end
```

Figure B.1: Controller and Cal source code of a merge actor.

The central piece of the actor machine is the *controller* that performs the action selection. The controller is an automaton with a finite set of *controller states*, each of which is associated with a set of possible *instructions* that can be performed in that state. The controller is executed by performing one of the instructions in the current state, starting in the initial state. Possible instructions are TEST, EXEC and WAIT.

TEST Tests a condition and depending on the result proceeds to one of two specified controller states.

EXEC Executes an action and proceeds to a specified controller state.

WAIT Does nothing but proceeds to a specified controller state.

Figure B.1 shows a visual representation of a controller and the source code of a merge actor. (We use the language Cal [12] to describe actors in this paper, because the differences between most classes of actors presented here can be described in Cal.) The circles are the controller states, the diamonds are TEST-instructions, the rectangles are EXEC-instructions, and the rings are WAIT-

instructions. The dashed edges are followed when the test result is false, and the solid edges are followed otherwise.

B.4 ACTOR CLASSES AND CLASSIFIER

Since it is the actions that carry out the communication of the actor, different sequences of actions result in different communication patterns. To reason about which actions can be selected in an actor state, we look at the *possible* steps that a controller can take. If a predicate condition is tested and later tested again without any action execution in between, the only possible step on the second test is to follow the same branch as in the first test. However, port conditions may first be false and later true because tokens may arrive at any time.

Definition 1: A *decision path* is a sequence of possible steps in the controller that ends with an EXEC and contains only one EXEC.

At the end of a decision path, knowledge about the state of the actor has been acquired through the tests on the path, but not all test results are interesting in this regard. A decision path may test the same condition several times, which is what happens when waiting for a token to arrive, and then, only the last test gives adequate information about the actor state where the action is selected.

Definition 2: The *path knowledge* of a decision path is a mapping from conditions that are tested in the path to the result of the latest test of that condition in the path.

With these two definitions, decision path and path knowledge, we will go through a set of well studied classes of actors and define a classifier based on the structure of an actor machine, starting with deterministic actors.

B.4.1 Deterministic actors

Definition 3: An actor is deterministic if there exists no actor state where more than one action is enabled.

A non-deterministic actor can be hard to test for correctness, because such actor might behave differently in the testbed and on the target platform. A deterministic actor, however, can be tested by putting tokens on the input ports and running the actor to verify the output it produces. Merge in Figure B.1 is

non-deterministic because when there is input on both X and Y, actions x and y are both enabled.

Definition 4: An actor machine is classified as deterministic if for all pairs of decision paths starting in the same state and ending with EXEC-instructions for different actions, there exist a condition that is tested in both paths for which the path knowledge of that condition differ on the two paths.

As an example, the controller in Figure B.1 has two decision paths that start in the initial state and end with actions x and y respectively that contain only one test each. The actor machine is not classified as deterministic, because the tests on the two paths are for different conditions.

Theorem 1: A non-deterministic actor machine can not be classified as deterministic.

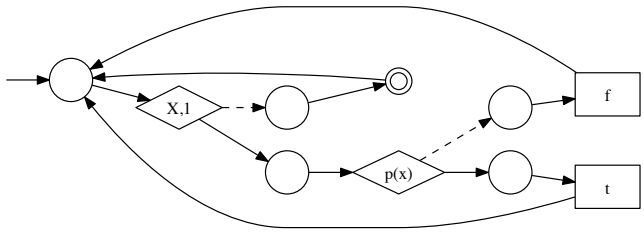
Proof. As a consequence of Definition 3, an actor that is non-deterministic has a state where more than one action is enabled. If there exists an actor state where an actor machine can select two different actions, the controller must contain a state with two decision paths to different actions that both are possible in that actor state. If two decision paths are possible in the same actor state, all conditions that are common for both paths will have the same result because the conditions are predicates on the actor state. An actor machine with two such paths can according to Definition 4 not be classified as deterministic. Hence, a non-deterministic actor machine can not be classified as deterministic. □

Deterministic actors as described above can produce different results if the input tokens arrive in a different order, even though the tokens on each port individually arrive in the same order. The actors in the next section, however, are not sensitive to timing.

B.4.2 Prefix monotonic actors

Prefix monotonic actors form an important subclass of the deterministic actors. Kahn shows in [17] that a network of prefix monotonic actors is also prefix-monotonic. This property makes it easier to verify correctness not only of a single actor, but a whole network of actors.

Prefix, denoted $\sqsubseteq$, is a partial order. On sequences s and s', prefix is defined as $s \sqsubseteq s'$ if and only if the initial part of s' is s. On n-tuples of sequences




```
actor Split (p) X ⟹ T, F:
  t: action [x] ⟹ T: [x] guard p(x) end
  f: action [x] ⟹ F: [x] end
  priority t > f; end
end
```

Figure B.2: Controller and Cal source code of a split actor.

$\mathbf{s} = (s_1, \ldots, s_n)$ and $\mathbf{s}' = (s'_1, \ldots, s'_n)$, prefix is defined as $\mathbf{s} \sqsubseteq \mathbf{s}'$ if and only if $\forall i(s_i \sqsubseteq s'_i)$. A function g is prefix monotonic if $x \sqsubseteq x'$ implies $g(x) \sqsubseteq g(x')$.

Definition 5: A prefix monotonic actor is an actor that can be described as a prefix monotonic function from k-tuples of sequences to n-tuples of sequences, where the k-tuples are the input sequences and the n-tuples are the output sequences.

Definition 6: An actor machine is classified as prefix-monotonic if it is classified as deterministic and the path knowledge for all tested port conditions in all decision paths is *true*.

Figure B.2 shows the controller of an actor machine that is classified as prefix-monotonic. The actor has one port condition and one predicate condition, and in all decision paths where the port condition is tested, the knowledge of the port condition is true.

Theorem 2: An actor machine that is classified as prefix monotonic is a prefix monotonic actor.

Proof. In a deterministic actor machine, all pairs of decision paths from a controller state to different actions have a condition for which the path knowledge differ. If it is classified as prefix monotonic, the condition on which the path

knowledge differ can not be a port condition. As a consequence, when one action is enabled, adding more input can not make another action enabled. Therefore, independent of when the tokens arrive on the input ports, as long as they arrive in the same order, the output of such actor must be the same. In other words, given input sequences $\mathbf{x}$ to an actor machine classified as prefix monotonic, let its output be $\mathbf{y}$, for all prefixes of $\mathbf{x}$ the output of the actor machine will be prefixes of $\mathbf{y}$. □

B.4.3 Kahn Processes

In [17], Kahn describes a process language with a read operation that blocks on one port until there is a token and with a write operation that is non-blocking. The language describes prefix-monotonic actors where every token requested from an input port is also consumed.

Definition 7: An actor machine is classified as a Kahn process if it is classified as prefix-monotonic, and for all decision paths, the action it leads to consumes all tokens that have been tested by the port conditions on the path.

B.4.4 Cyclo-Static Data Flow

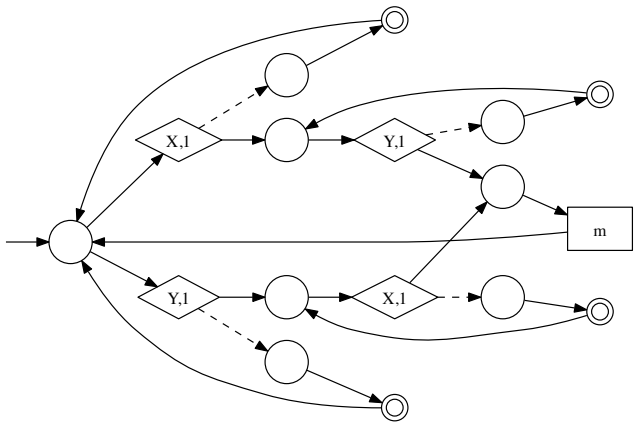
In a cyclo-static data flow actor, the token rates of successive actions appear in cyclic patterns. Engels et al. show in [43], as an extension to the class described in the next section (B.4.5), that a network of such actors can be efficiently implemented with a static schedule. A fixed bound on the sizes of the buffers in the communication channels can also be determined when the schedule is static.

In this paper, for classifying an actor as cyclo-static, we require the actions to appear in cycles rather than just the token rates of the actions. This restriction makes it possible to statically schedule not only the actors, but even the individual actions of the actors. We also allow any number of initial actions before the cycle starts.

Definition 8: An actor machine is classified as cyclo-static if it is a Kahn process and it does not have any predicate conditions and the controller has a cycle involving at least one exec-node.

Theorem 3: Actor machines classified as cyclo-static are cyclo-static.

Proof. In a Kahn process, different actions can not be selected based on the




```
actor SynchMerge () X, Y ⟹ Z :
  m: action [x], [y] ⟹ [x, y] end
end
```

Figure B.3: Controller and Cal source code of a synchronous merge actor.

length of the input sequences. By further restricting the action selection to not be based on predicate conditions, there can be only one successor action of every action. The initial action together with the successor of each action form a chain. If this chain is cyclic, then the actor is cyclo-static. □

B.4.5 Synchronous Data Flow

Lee and Messerschmitt introduce the class of synchronous data flow actors in [42] and show that a network of such actors can be scheduled statically and with fixed buffer sizes. A synchronous data flow actor is an actor where all actions have the same token rate. Figure B.3 shows the controller of a synchronous actor machine of a merge actor. Note that the order in which the two port conditions are tested is non-deterministic, but the actor itself is still deterministic.

Definition 9: An actor machine is a synchronous data flow actor if it is classified as cylco-static and all actions have the same token rate.

This definition follows directly from the definition of cyclo-static actors and synchronous actors.

B.5 IMPLEMENTATION

We have implemented a classifier tool based on the definitions in the previous section. The tool uses the front-end and core data structures of a compiler for dataflow programs with actor machines as intermediate representation. When a front-end for a new language is added to the compiler, the classifier will immediately be available for this language as well.

Some class definitions contain wordings like "all decision paths" and "all pairs of decision paths". Enumerating all decision paths can be impossible, because there can be infinitely many decision paths in a controller. In fact, most controllers we have seen contains a TEST-WAIT-loop, which generates infinitely many decision paths. However, in all definitions where all decision paths might be needed, the only thing that is required is the path knowledge of those paths. Since the number of conditions in an actor machine is finite, the number of variants of the path knowledge is also finite.

B.6 RELATED WORK

Zebelein et al. [31] describe an abstract actor model for SystemC applications on which they perform actor classification. Their classifier tries to find synchronous data flow actors [42] and cyclo-static data flow actors [43] to enable static scheduling of those actors in an application.

Wipliez and Raulet [45] present a classifier for actors written in RVC-CAL that tries to recognize the classes from [31] as well as parameterized synchronous data flow actors [46]. Their classifier use abstract interpretation to narrow down the class of an actor.

The Model Compiler in the ACTORS-project [47] also includes a classifier for actors written in Cal. This compiler recognizes three classes: *static*, *dynamic* and *timing-dependent*, where *static* corresponds to the class of cyclo-static actors, *dynamic* to prefix-monotonic actors, and *timing-dependent* to the actors that are not prefix-monotonic.

B.7 CONCLUSION

In this paper we develop a way to reason about the communication behavior of actor machines. This is applied when classifying actor machines into classes known from prior art, such as prefix monotonic actors and synchronous data flow. The classifier is implemented in a tool that uses the front-end and data structures of a compiler with actor machines as internal representation.

Paper C

Software Code Generation for Dynamic Dataflow Programs

Gustav Cedersjö and Jörn W. Janneck
Department of Computer Science, Lund University

ABSTRACT In this paper we address the problem of generating efficient software implementations for a large class of dataflow programs that is characterized by highly data-dependent behavior and which is therefore in general not amenable to compile-time scheduling. Previous work on implementing dataflow programs has emphasized classes of stream processing algorithms that exhibit sufficiently regular behavior to permit extensive compile-time analysis and scheduling, however many real-world stream programs, do not fall into these classes and exhibit behavior that can, for example, depend on the values and even the timing of their input data. Based on an abstract machine model, we partition the problem of implementing such programs in software into three parts, viz. reduction, composition, and code emission, and present solutions for each of them. Using the reference code of an MPEG decoder, we evaluate the resulting code quality and compare it to the state of the art compilers for the same class of stream programs, with favorable results.

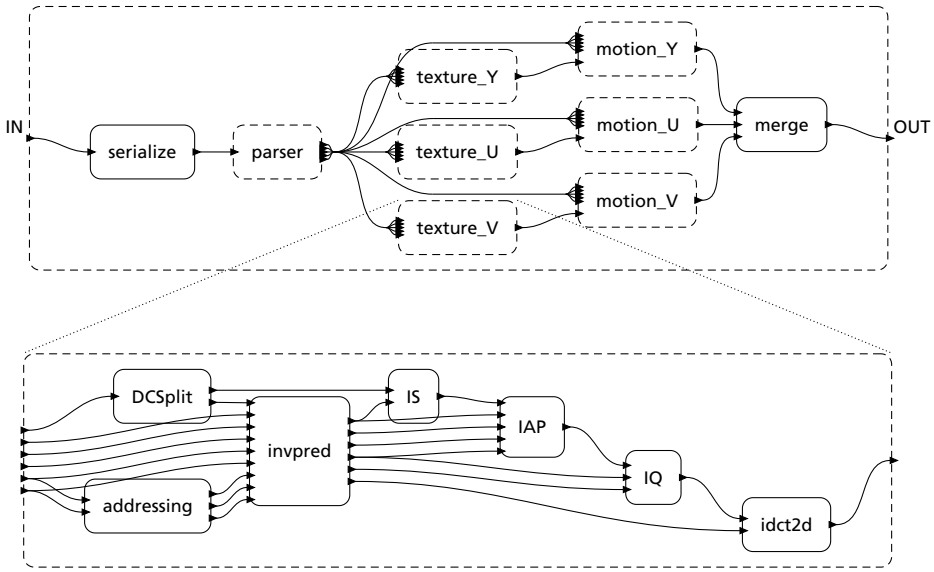


Figure C.1: Structural view of a dataflow program representing an MPEG-4 Simple Profile decoder. Boxes with solid border are actors, and boxes with dashed border are networks of actors. To get a better overview, some connections have been collapsed to a single arc. Source code: https://*github.com/orcc/orc—apps*

C.1 INTRODUCTION

The end of clock scaling in modern computers is leading to a rapid increase in and ubiquity of parallel computing platforms. At the same time, many of the most computationally demanding application areas are dominated by the processing of streams of data, e.g. digital signal processing, audio and video coding, cryptography, network and packet processing, image analysis, computer graphics. The confluence of these developments has created renewed interest in dataflow programming models, resulting, for instance, in the adoption of a dataflow model (and an associated dataflow programming language) by MPEG and ISO as the foundation for standardizing video codecs [48] and for 3D graphics coding [49].

We will use the term *dataflow program* for a network of computational kernels called *actors* such as the one depicted in Figure C.1, representing a simple video decoder. Each of the boxes with solid border represents an actor. We also call these actors *atomic* actors to distinguish them from the results we

get when we compose them hierarchically into larger units—for instance, the boxes with dashed border in the figure are assemblies of atomic actors (*composite* actors) that in turn are composed to create the complete decoder. Actors have input and output *ports* through which they receive and send packets of data, called *tokens*. The connections between actor ports are buffered, lossless, order-preserving *channels*. We conceptualize the buffer capacity of each of these channels initially as infinite, though of course any actual implementation will need to impose some kind of bound on their size. Exchanging tokens along these channels is the only way for actors to communicate—any internal state an actor may maintain is kept distinct and invisible to all other actors.

The kind of dataflow program we will be concerned with here has the additional property that each actor in it performs its computation in a sequence of discrete, atomic *steps*, also called *firings*. This variant of dataflow is also referred to as *dataflow with firing* [38] in order to differentiate it from models such as *Kahn processes* [17], where each computational kernel is a continuously executing process without any partitioning of its execution into steps. In each of its steps, an actor may (1) consume input tokens, (2) produce output tokens, and (3) modify its internal state.

In general, the steps of different actors may be executed independently of each other, subject only to the availability of input. However, if several actors are to be executed on a single processor, their steps need to be interleaved, i.e. *scheduled*. For classes of actors whose behavior bears little or no dependency on the values of the tokens being processed, and whose *token rates* (i.e. the number of tokens produced and consumed at each port on each step) are very predictable, scheduling can happen largely or even entirely at compile-time, removing significant run-time overhead. However, in many practical applications, actor behavior does vary depending on the input data, and token rates cannot be predicted statically. For instance, in our example in Figure C.1, only ten out of the 39 atomic actors could be identified as amenable to compile time scheduling (using the actor classifier in [50]).

In this paper, we investigate the problem of software code generation for a very general class of dataflow programs, which permits behavior that does depend on internal state as well as input data, and is not even required to be deterministic. Starting from a representation of actors as *actor machines* [4], we structure the problem of generating an efficient software implementation into three parts: (1) a process called *reduction* that yields a sequential algorithm for executing an actor, (2) a process called *composition* which computes an actor that implements a network of actors, and finally (3) code generation, which

emits code (in this case, a C program) that can be compiled to the target. We take an initial stab at each of these problems, and evaluate the resulting code quality using the decoder in Figure C.1.

This paper is structured as follows. After reviewing some related work in the next section, we summarize the basic machine model in section C.3. Then we first discuss a high-level notion of code generation related to a transformation of those machines in section C.4. In section C.5 we apply the same model to the problem of composing a very general class of dataflow actors. In section C.6, we describe a compiler based on that machine model, and in section C.7 we evaluate the effects of composition. In section C.8 we discuss the results and possible directions for future work.

C.2 RELATED WORK

One of the earliest attempts to statically schedule dataflow programs focused on a class called *Synchronous Data Flow*, or SDF [1]. SDF actors are limited to constant token rates and a program consisting only of these actors permits extensive analysis, can be shown to execute in bounded memory, and can be scheduled entirely at compile-time. Cyclo-static Data Flow (CSDF) [51] retains those benefits while slightly extending expressiveness by allowing actors whose token rates vary periodically in fixed cycles. Interestingly, neither class is strictly speaking closed under composition, i.e. composing a network of SDF actors will either lead to an actor that is itself not SDF, or that behaves differently from the original program.

Since then, a variety of models (amounting to larger classes of dataflow programs) have been devised with the goal of maintaining the benefits of static analysis and scheduling while increasing the expressiveness. These include Parametric SDF (PSDF) [46], Scenario-Aware Data-Flow (SADF) [52], Variable-Rate Data-Flow (VRDF) [53], Schedulable Parametric Data-Flow (SPDF) [54]. We view these efforts as complementary to our work in that they provide concrete techniques that could be applied to important subclasses of the dataflow programs we want to represent. A more extensive discussion of approaches to dealing with more dynamic dataflow behavior can be found in [55].

Other work has aimed at overcoming the composition problem of SDF programs, such as Cluster Finite State Machines (CFSM) [2] (itself not hierarchically compositional) and Deterministic SDF with Shared FIFOs (DSSF) [56], the latter essentially solving the composition problem for SDF. Our work is focused on a much more general class of dataflow programs, and for that reason

```
actor Foo () X ⟹ Y, Z:

A1: action X: [v] ⟹ Y: [v]
    guard v ≥ 0
    end

A2: action X: [v, w] ⟹ Z: [v + w]
    end

    priority A1 > A2; end
end
```

Listing C.1: A simple actor, written in Cal.

we also need to consider issues such as efficiency and overhead that do not apply to programs that can be completely scheduled at compile time, and which cannot, in general, be measured without executing the program.

Finally, our work is motivated by the desire to efficiently implement actor languages such as Cal [12]. The basic machine model it builds on is described in [4] (and more formally in [57]), while the translation of Cal into this model is outlined in [44]. We will summarize the salient points in the next section.

C.3 ACTORS AND ACTOR MACHINES

The language we use to represent (atomic) actors is Cal [12]—Listing C.1 shows an example. The actor has one input port and two output ports (X and Y, Z, respectively), and two actions, A1 and A2. A1 consumes one input token and produces one output token on Y, and it has a guard that specifies that it can only fire if the value of the input token is not negative. A2 consumes two input tokens and produces one output token on Z, and since it is priority-ordered below A1, A2 can only fire when A1 cannot.

The details of the actor language matter little here—the important point is that part of the specification of the actions in an actor are the *conditions* that need to be satisfied for an action to be able to fire, and that there are two kinds of conditions: *input conditions* and *guards*. Input conditions in Cal are denoted by the input clauses of actions—e.g., X: [v, w] states there be two tokens available on the input port X. Guards are boolean expressions that may depend

on input tokens and actor state. If all input conditions and guards of an action are satisfied we say that it is *enabled* if no higher-priority action is enabled.

The execution of an actor can then be seen as alternating between two phases: *action selection*, i.e. the determination of an enabled action (there might be more than one, in which case any one of them may be chosen), involving the evaluation of the conditions, i.e. input conditions and guards), and *action execution* of an enabled action.

We shall codify this process of selecting actions by determining truth values for the conditions and then executing an enabled action in an *actor machine*. An actor machine implements an actor description, and its core is the *controller*, a finite state automaton with three kinds of transitions (*instructions*) between its states. The TEST(c, σ_1, σ_2) instruction tests a condition c (either input condition or guard) and transitions to state σ_1 if the condition was satisfied, and to σ_2 otherwise. The EXEC(a, σ) instruction executes an action a and then transitions to state σ. Finally, the WAIT(σ) instruction simply transitions to state σ. The purpose of the WAIT instruction is to indicate that some condition (e.g. the availability of tokens) must be changed to select an action.

Intuitively, each controller state represents the current knowledge of the actor machine about its environment, as ascertained by the testing of conditions. For example, if we enumerate the conditions of Foo as c_1 : (Foo.X, 1), c_2 : (Foo.X, 2) and $c_3 : \langle v >= 0 : v = \mathsf{X}[0] \rangle$, denoting, respectively, that there be one input token at X, two input tokens at X, and that $v >= 0$ be true for v being the first token on X, then each controller state represents a three-tuple on the set $\{\mathrm{X}, 0, 1\}$, denoting that the corresponding condition is either unknown, or known to be false or true, respectively. For instance, the triple 1X1 would represent the fact that we know that there is one input token, we do not know whether there are two input tokens, and we know the guard of A1 to be satisfied.

Note that conditions are not always unrelated. For example, c_2 implies c_1, since the presence of two tokens implies the presence of one on the same input port. Also, since c_3 requires an input token to be testable, it is said to *depend* on c_1, i.e. it can only be tested when we know c_1 to be satisfied.

Figure C.2 shows a graphical depiction of the controller for Foo. The ellipses represent controller states, the diamonds are TEST instructions (true branch solid, false branch dashed), the boxes EXEC instructions, and the rings are WAITs. The truth of (Foo.X, 2) implies that of (Foo.X, 1), so when we find condition c_2 true in state XXX (the initial state, which represents minimal information about the actor's environment) we transition to 11X. Conversely, when we find condition c_1 to be false in XXX, we transition to 00X.

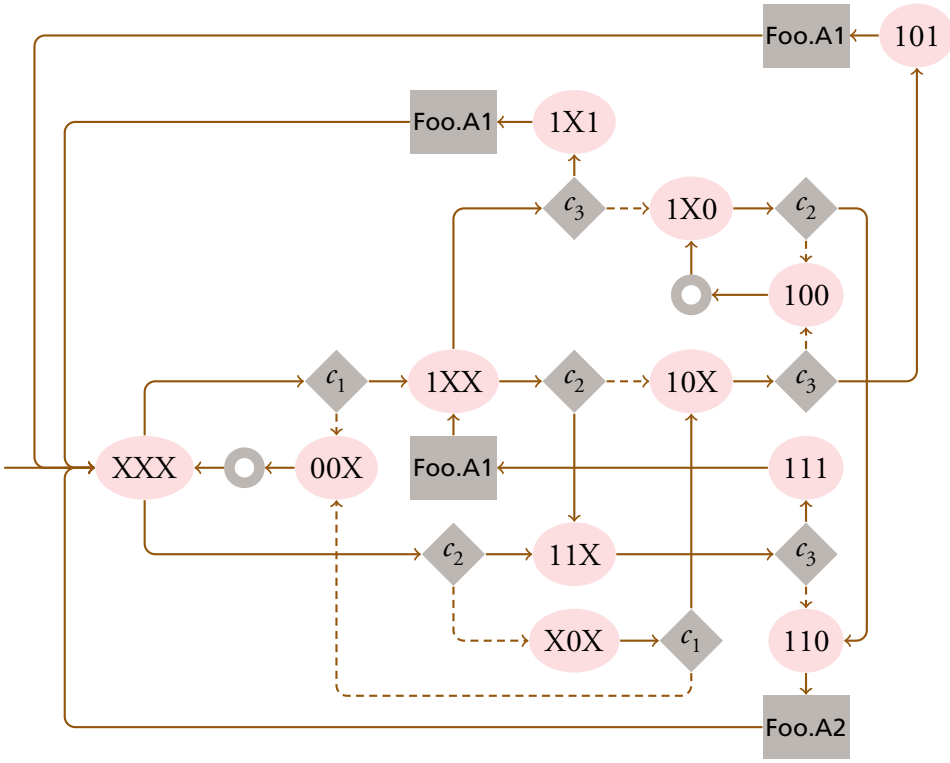


Figure C.2: Basic actor machine for the **Foo** actor in listing C.1.

Notice that in state 00X nothing can be done but wait for more input tokens, and re-test the input conditions. That is the purpose of the WAIT instruction, which essentially throws away information about the environment in order to re-test input conditions that have turned out to be false, but may have become true in the meantime.

In state 111 (two tokens are available in the input port, and the guard is true) we can fire A1. However, in that case we do not, as in the case of the other EXEC instructions, go back to state XXX, but to state 1XX. The reason is that we know that at least two input tokens are available before the firing, but as we consume only one of them, at least one will remain, so (Foo.X, 1) will still be true.

An actor machine can be seen as an *implementation* of an actor in a language like Cal, one of possibly many ways of representing that actor's behavior. The

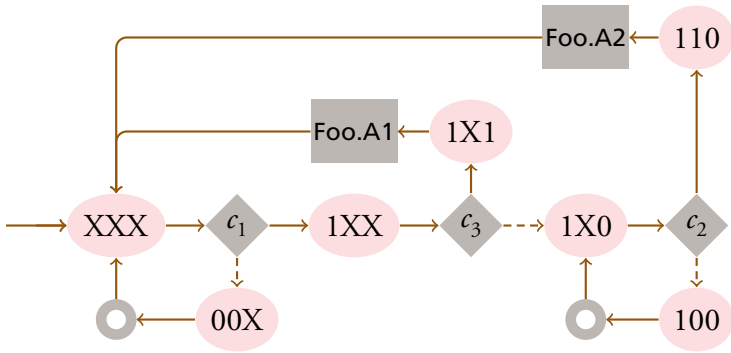


Figure C.3: A reduction of the actor machine in Figure C.2.

machine in Figure C.2 is a realization of a controller for Foo, and we will now look at a few alternative controllers that we derive from this fairly general form.

C.4 REDUCTION AND CODE GENERATION

The controller in Figure C.2 might appear unnecessarily complex, since it permits various ways in which to determine the next action to fire by testing the three conditions on which this choice depends. In particular, it contains several states that allow more than one instruction, for instance the initial state XXX. We call such a machine a *multi-instruction actor machine* (MIAM), it is used to represent a more general form of the controller. By contrast, a *single-instruction actor machine* (SIAM) allows at most one instruction in each controller state. SIAMs are an important subclass of actor machines because they directly correspond to a sequential implementation of an actor. In fact, in order to implement an actor on a sequential processor, we in essence need to construct a SIAM implementing it, which is the first step toward code generation.

The process of constructing a single-instruction machine from a multi-instruction machine, which we call *reduction*, is very straightforward in principle—in any state with more than one instruction, any one of them may be chosen, so we can arbitrarily pick one among them, remove all others, and also delete all states and their associated instructions which can no longer be reached. For instance, the machine in Figure C.3 results by testing condition c_1 in state XXX and then condition c_3 in state 1XX.

Reduction not only makes the controller sequential, it can also significantly

Table C.1: Number of states in the controllers for actors in MPEG4 SP decoder, for a MIAM and a SIAM generated from it.

Instance name	Number of states	
	MIAM	SIAM
add	25	22
addressing (U and V)	12	12
addressing (Y)	61	22
blkexp	41	21
DCsplit	3	3
FrameBuff	318	90
IAP	31	18
idct2d	8	5
interpolation	29	16
invpred (U and V)	126	42
invpred (Y)	406	66
IQ	11	8
IS	33	33
Merger420	9	9
mvrecon	75	69
mvseq	596184	524
parseheaders	43981	1232
serialize	14	12
splitter_420_B	54	31
splitter_BTYPE	612	68
splitter_MV	42	42

reduce its size. Table C.1 shows controller sizes (measured in number of states) for the MIAM and a SIAM generated from it, for the actors in the MPEG SP decoder from Figure C.1. For instance, note the very large MIAM controller of the mvseq actor, which corresponds to a controller with many conditions that can be checked in many different orders, and which reduces by more than three orders of magnitude into a SIAM.

Since these choices can be made arbitrarily, we could also produce the machine in Figure C.4 as a reduction of the one in Figure C.2, by choosing to test condition c_2 in state XXX. Even though it turns out to be bigger, it is not necessarily the inferior choice.

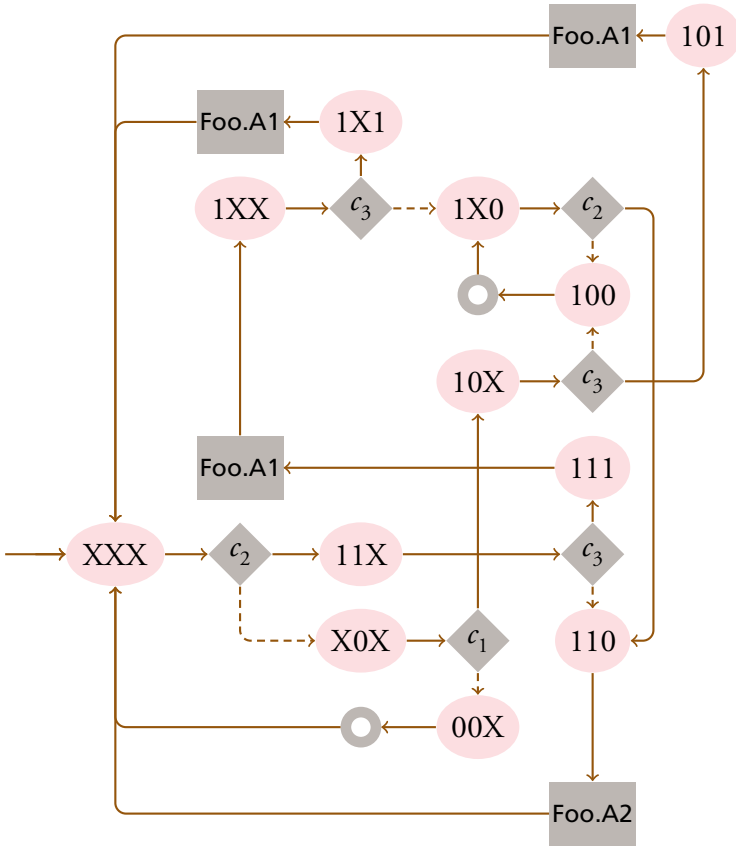


Figure C.4: Another reduction of the actor machine in Figure C.2.

The reason is that in generating sequential code we are not only concerned with the size of the controllers, but also with their *speed*. A measure for the speed of a controller is the number of tests it needs to perform in order to select the actions an actor needs to fire for a given input.[1] For statically schedulable dataflow programs, that number is zero, since no run-time tests need to be performed. For more general dataflow programs, however, some conditions will need to be tested at run time. Reduction can be viewed as a generalization of static scheduling for programs with data-dependent behavior.[2]

[1] Of course, in general we will need to weight these tests according to the amount of time required to execute them.

[2] In using this metric, we assume that the *actor* is deterministic and monotonic, and that for a given input the actor machine will always execute the same sequence of actions.

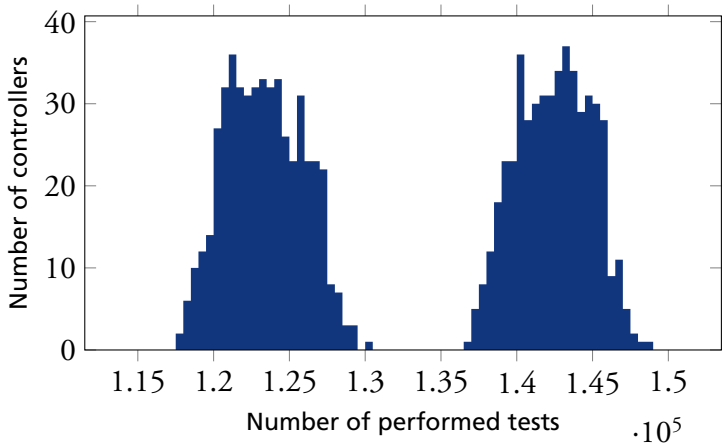


Figure C.5: Histogram over number of tests performed in parseheaders for 1000 random reductions for the same video stream.

Different reductions of the same multi-instruction machine can differ substantially with respect to run-time efficiency, albeit in ways that will, in general, depend heavily on the input data. Consider, e.g., the behavior of the two reductions above for (a) a slowly-arriving input stream of mostly positive numbers and (b) a fast-arriving stream of negative values. (The difference in arrival speed affecting the probability of the test for two input tokens succeeding right at the start.)

While it is easy to compute *some* reduction of a given MIAM, it is in general not trivial to compute a *good* one. In order to determine how much the choices made during reduction matter in terms of execution speed, we have randomly generated a population of 1000 reductions from a version of the parseheaders actor in Figure C.1, which has the most complex SIAM controllers in the program. Figure C.5 depicts the distribution of the number of tests the reductions performed for a given reference video stream, ranging from 117,000 to 149,000, a difference of over 20%.

In the presence of good reference input data, this approach of randomizing reductions and picking the best one after profiling could even be considered a code generation technique, albeit a costly and crude one. Reliable heuristics, however, that consistently point toward good choices during reduction have so far proved elusive.

```
actor Bar () X ⟹ Y:

Do: action X: [v] ⟹ Y: [f(v)]
    end
end
```

Listing C.2: Another very simple actor.

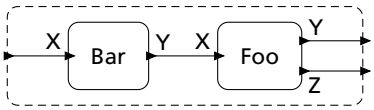


Figure C.6: A composition of two actors.

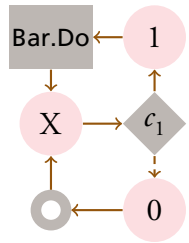


Figure C.7: The controller of the actor in Listing C.2.

C.5 COMPOSITION

A typical scenario when implementing streaming applications asks for a program consisting of n actors to be mapped to a network of $k < n$ processing elements, such that several actors will end up sharing the same processing element. This means that the instruction sequences that correspond to the execution of the actor machines on a processing element need to be interleaved, and one way of doing this is to construct a *composite actor machine* whose controller represents the interleaved execution of all the controllers of the component machines.

To make this more concrete, suppose we want to execute the dataflow program in Figure C.6, which includes our actor `Foo` from Listing C.1 and the actor `Bar` from Listing C.2. The latter has the controller depicted in Figure C.7, with the sole condition $c_1 : (\texttt{Bar.X}, 1)$.

One way of constructing the composite controller would be to build the product automaton. This would be tantamount to an arbitrary interleaving of the individual machines, but we can create much more efficient compositions that eliminate many of the redundant tests of such an interleaving.

First of all note that our composite controller is in charge of executing all actions that read to and write from the internal connections (i.e. connections between two actors that are to be composed), so if we kept track of how many tokens have been written to and read from each of them, there is no need to test any input conditions on internal connections. The idea, then, is to make

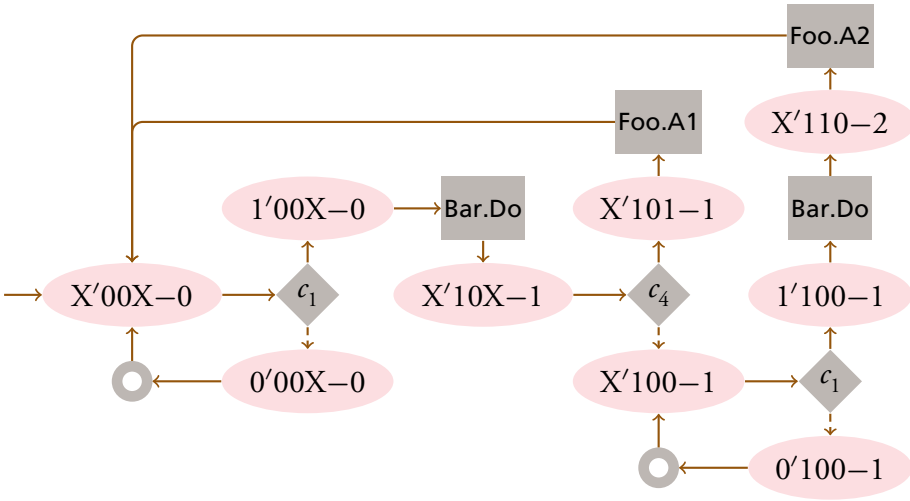


Figure C.8: A controller for the composite in Figure C.6.

the number of tokens waiting on the internal connections part of the controller state. For example, the states of composite controller of the program in Figure C.6 would include the knowledge about the one condition of Bar, about the three conditions of Foo, and in addition they would contain a natural number denoting the number of tokens in the single internal connection between those actors. Our initial state, therefore, might be written as X′XXX−0: we separate the condition of Bar and those of Foo by a ′ symbol for readability, and the token count follows the dash.

Note, however, that the first two conditions of Foo are input conditions, dependent only on the number of input tokens available to Foo on X. Therefore, if we know the connection to not contain any tokens, we also know those conditions to be false, and thus we can start with initial state X′00X−0 instead.

Starting from this initial state, we can build up the composite controller in a process called *abstract simulation* [4], whereby we successively elaborate states by following one (or more) of the instructions that can be executed in them, leading us either to old states, or new ones. If any of the instructions changes the number of tokens in any of the internal connections, we keep track of this in the new states we transition to, including updates to any conditions that might be affected.

A possible composite controller resulting from this process is shown in Figure C.8. In general there may be many composite controllers, depending on

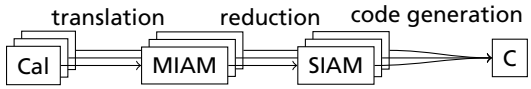


Figure C.9: A basic compiler work flow for translating networks of Cal actors to a C programs using actor machines as intermediate representation.

which, and how many, instructions we choose in those composite states that allow more than one to be executed. The resulting composite machine in this example is a SIAM, because we elaborated every state by following only one instruction. In this case, the heuristic that decides which instruction to choose will need to be built directly into the elaboration. Alternatively, composition could follow several, even all, possible instructions in each state, leading to a larger controller which, for sequential implementation, will need to be reduced afterward. The trade-offs involved, as well as the heuristics used during composition, are the subject of our current investigations.

C.6 ACTOR MACHINE COMPILER

To evaluate the compilation techniques described in section C.4 and C.5, we built a compiler and a composer for actor machines with a front-end for Cal and a back-end producing C programs. The basic work flow of the compiler is depicted in Figure C.9.

The first step in the compilation is to read a network of Cal actors and translate them to multi-instruction actor machines. The result of this translation is a network of actor machines that is semantically equivalent to the source network. To simplify the translation, in the compiler, the Cal actors use the same representation for statements and expressions as the actor machines do. The actor machines in this network are reduced to single-instruction actor machines, and since good reduction heuristics are still to be found, we chose to reduce the controller by simply selecting an arbitrary instruction in each controller state. The network of actor machines is then passed to the C back-end that generates a C program for that network.

C.6.1 Code Generator

Given a network of single-instruction actor machines, the code generator generates a single file C program that executes the actors in a round robin schedule, letting each actor run until the next WAIT instruction before switching to the

next actor. The channels that connect the ports of the actors are implemented with circular buffers of fixed size. For every input connection to the network, there is a special actor that reads data from a file and writes the content to the buffer, and output connections are handled similarly. These input and output actors are scheduled with the same schedule as the other actors in the program. The input and output files are specified as command line arguments to the program.

Apart from the input and output ports that the channels connect, an actor machine also contains a set of actions, a set of conditions (port conditions and guards), a set of state variables that are grouped into scopes, and a controller that performs the action selection. Actions are translated to functions in the C program, conditions are translated to expressions, state variables become static variables in C program and for each scope, a function that initializes its variables is generated.

The actor machine controller, that drives the action selection, is arguably the part of this compiler that differs the most from other Cal compilers such as Orcc [58]. It is implemented with a function where each state in the controller starts with a label after which the code for the instruction is put, and the transitions between the states are implemented with goto-statements. Listing C.3 shows the controller function for the actor in Listing C.2. There are three kinds of controller instructions: EXEC, TEST, and WAIT. An $\textsc{exec}(a, \sigma)$ instruction start with function calls to the initializers of the scopes that need to be initialized, and continues with a call to the function representing action a, and a goto-statement to the label of σ. A $\textsc{test}(c, \sigma_1, \sigma_2)$ is implemented with an **if**-statement on the expression representing c where the two branches are goto-statements to the labels of σ_1 and σ_2. If c is a guard that needs to initialize a scope, then a call to the appropriate initializer is placed before the **if**-statement. On the $\textsc{wait}(\sigma)$ instruction, the execution should continue with another actor. This is implemented by saving σ to a variable and returning from the function, letting the scheduler switch to another actor. In the beginning of the controller function, there is a switch-statement on that variable where each case has a goto-statement to the appropriate label.

C.6.2 Compiler Evaluation

In order to quantify the quality of the resulting code, we compiled the reference implementation of the MPEG-4 Part 2 Simple Profile decoder, written in RVC-CAL, with the actor machine compiler and with the Open RVC-CAL Compiler (Orcc) [58]. Figure C.1 shows a structural view of this video decoder that

```
static _Bool actor_0(void) {
   _Bool progress = false;
   static int state = -1;
   switch (state) {
   case -1: break;
   case 0: goto S0;
   case 2: goto S2;
   }

   S0: // test(tokens(X,1), S2, S1)
   if (tokens_b0 ≥ 1) goto S2;
   else goto S1;

   S1: // wait(S0)
   state = 0;
   return progress;

   S2: // test(space(Y,1), S4, S3)
   if (tokens_b1 + 1 ≤ size_b1) goto S4;
   else goto S3;

   S3: // wait(S2)
   state = 2;
   return progress;

   S4: // exec(Do, S0)
   init_scope_0_0();
   action_0_0();
   progress = true;
   goto S0;
}
```

Listing C.3: C function representing the controller of the actor in Listing C.2.

Table C.2: Execution time of the decoder in Figure C.1, compiled to C with two Cal compilers, and from C using Clang with four different optimization levels, decoding a 300 frames QCIF video.

Opt. flag	Actor Machine Compiler	Open RVC-CAL Compiler
-O0	1599 ms	2841 ms
-O1	923 ms	1158 ms
-O2	623 ms	682 ms
-O3	626 ms	667 ms

consists of 39 actors and approximately 5000 lines of code. We compared the two generated programs by their average execution time for decoding a given video sequence. Orcc is currently the best-maintained compiler tool chain for Cal (more precisely, for RVC-CAL), and is generating the most efficient code for general dynamic dataflow programs. In this comparison, no actor machines were composed. The actor machines were reduced to single-instruction actor machines by arbitrarily choosing one of the instructions in each controller state. The two C programs generated by the two Cal compilers were compiled using Clang version 4.2.1 with the optimization levels -O0 to -O3. These programs were then executed on a computer with a 2 GHz Intel Core i7 processor using the UNIX time command to measure the time. In this comparison, we look at the reported *user time* to compare the CPU time spent in the program.

The actor machine version was slightly faster than the Orcc version on the highest optimization level for this example. Table C.2 shows the time to decode a 300 frame QCIF video with the programs generated by the two compilers for different optimization levels, where the times are averages over 1000 executions. These numbers suggest that the actor machine is in its basic form more efficient than the Orcc representation, but also that a good optimizer can compensate for this inefficiency.

C.7 ACTOR MACHINE COMPOSER

We implemented an actor machine composer for the actor machine compiler described in the previous section. Figure C.10 shows the compilation flow of a dataflow program where the composition step is included. After the multi-instruction actor machines have been reduced to single-instruction actor machines, the network of actors is composed to a single actor, and the result of the

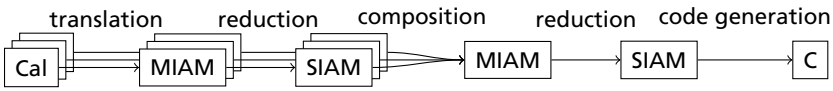


Figure C.10: Compiler work flow for compiling a network of Cal actors to a C program, where actors are composed using actor machines.

composition that is a multi-instruction actor machine is reduced to a single-instruction actor machine using the same method as mentioned earlier.

C.7.1 Composition Evaluation

To see how composition affects performance, we isolated a few pairs of directly connected actors from the rest of the MPEG-4 decoder, and for each pair we compared the execution times with and without composition. The input stimulus to these pairs of actors was the same as the input they got when decoding the video stream in the previous section.

The pairs of actors we looked at are runnable with bounded buffer sizes. For the composed version of each pair, we used the smallest possible buffer sizes. Buffer minimization was done manually for these examples. For the run-time scheduled versions, minimizing buffer sizes may affect the performance negatively because it introduces more tests at run time in the following way. If an actor is scheduled to run as long as it makes progress, it will stop on a WAIT instruction, usually after a failing TEST. The next time the actor is scheduled, the condition will be tested again to see if its truth value has changed. By using larger buffers, the port conditions will be true for a longer time, the actors can run with fewer context switches, and the number of reevaluated conditions is reduced. Because of this effect, we ran the run-time scheduled versions not only with minimal buffers, but also with the buffer sizes multiplied by 2, 4, 8, 16 and 32.

To measure the difference between the run-time scheduled and the composed versions, we used the actor machine compiler in both cases and compiled the generated C programs with Clang on optimization levels -O0 to -O3. To do the actual measurement, we added timers to the generated code that measures the time spent in the actors, not including the special file reading and writing actors.

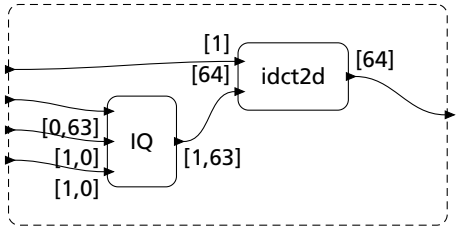


Figure C.11: Composition example A. The input and output ports are annotated with the cyclic token rates of the actor.

Table C.3: Execution times of example A, composed and run-time scheduled with different buffer sizes, compiled on different optimization levels. Measurements are normalized against the scheduled -O0 version with minimal buffers (36.4 ms).

Opt. flag	Composed	Scheduled, buffer size factor					
		1	2	4	8	16	32
-O0	0.90	1.00	0.97	0.92	0.91	0.94	0.93
-O1	0.36	0.47	0.43	0.40	0.38	0.38	0.38
-O2	0.17	0.28	0.24	0.21	0.19	0.19	0.18
-O3	0.17	0.28	0.24	0.21	0.19	0.19	0.18

Example A: Cyclo-Static Data Flow. The first pair of actors we composed and evaluated is the cyclo-static data flow network in Figure C.11 consisting of the actors IQ and idct2d from the texture part of the MPEG-4 decoder in Figure C.1. The IQ actor has a cyclic token consumption and production pattern — first it consumes a token on two input ports each and produces one token on the output port, then it consumes 63 tokens on the third input port and produces 63 tokens on the output port. The idct2d actor always consumes one token on one port, 64 tokens on the port connected to the IQ actor, and produces 64 tokens on the output port. Creating a static schedule for this kind of dataflow programs has been done before [51], but in this paper we show a new way of building such schedule by composing actor machines.

Table C.3 shows the execution time for the composed and the scheduled versions of this network, compiled with different optimization levels. For easier comparison, the times in the table are normalized against the scheduled version with minimal buffers and without optimizations. One thing to note about the

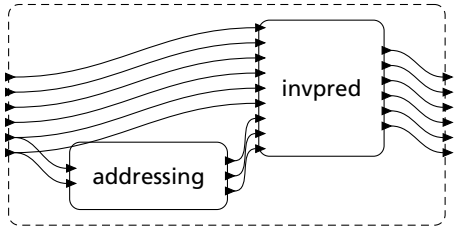


Figure C.12: Composition example B.

Table C.4: Execution times of example B, composed and run-time scheduled with different buffer sizes, compiled on different optimization levels. Measurements are normalized against the scheduled -O0 version with minimal buffers (4.1 ms).

Opt. flag	Composed	Scheduled, buffer size factor					
		1	2	4	8	16	32
-O0	0.74	1.00	0.90	0.74	0.70	0.66	0.65
-O1	0.49	0.74	0.61	0.52	0.47	0.44	0.43
-O2	0.27	0.54	0.41	0.33	0.29	0.26	0.24
-O3	0.26	0.55	0.41	0.33	0.28	0.26	0.24

results is that doubling the buffer sizes increases the performance.[3] We believe this is due to fewer conditions being retested if the same actor runs for a longer time before switching. Note also that the composed version is faster than the scheduled one, even when the buffer sizes of the scheduled network are 32 times larger.

Example B: Dynamic program. The second example is a pair of actors where the actions can not be scheduled statically, because the sequence of firings depends on the values of the input tokens. Figure C.12 shows a structural view of this example, which is part of the texture network in the MPEG-4 decoder in Figure C.1. This example is runnable with the three buffers connecting addressing and invpred having space for only a single token each.

Table C.4 shows the execution times for the different versions on different optimization levels. For easier comparison, the times are normalized against the

[3] An exception to this appears to be the lowest optimization level where we observe a slight slowdown for larger buffers. As we use a general-purpose CPU for our experiments, we might be observing cache effects here, but more experiments would be required to isolate the cause.

run-time scheduled version with optimization level -O0 and minimal buffers between the two actors. This example also shows a speedup when using composition instead of run-time scheduling. On the high optimization levels, this speedup is the same as for the run-time scheduled with 16 times larger buffers. The scheduled version even performs better for very large buffers. This, too, might be attributable to the higher locality of the use of data, which would again suggest an effect of the processor cache.

C.8 CONCLUSION

In this paper we have structured the problem of software code generation for dynamic dataflow programs into reduction, composition, and code emission, based on the actor machine model. We implemented some basic solutions for each of those stages, and evaluated them quantitatively with respect to the speed of the generated code. Our basic code generation improves on the reference (by Orcc) even with randomly chosen reduction and no composition. Composition, again without any heuristic to guide it toward better results, further improves on this by between 10 and 50 percent for the same buffer size, even though uncomposed (scheduled) actor networks can "catch up" if given much larger buffers. As embedded targets are common for many typical applications, we need to gather more data for these platforms, and also for other applications than video coding.

These results are most encouraging, and we believe there is still potential for improvements based on more sophisticated approaches to reduction, composition, and code emission. This opens large areas of investigation that we have so far only begun to explore. Good solutions to the problems will be enabling techniques for effectively and efficiently implementing general dataflow programs. As we have shown, different reductions can vary considerably with respect to their run-time efficiency, so one area of future work will be robust heuristics that help us produce better reductions. These might require knowledge, such as in the form of statistics, about the nature of the input streams.

Composition of realistic dataflow programs will always have to deal with the possibility of combinatorial explosion of the controller state space. In practice, it will often do this by trading off controller size against run-time performance (i.e. more tests). The challenge is to identify good trade-offs, and help designers navigate the huge design space resulting from the many ways in which actors may be composed. The idea of extending the controller state to include other

information about the program could be expanded by the use of abstract interpretation, which would allow us to track not only token counts in the controller state, but also maintain information about the values of tokens. This, in turn, could be used to eliminate the testing for some of the guards.

Finally, actor machines admit a variety of styles for generating code, of which we have only begun to explore one. In addition to run time, code size (which we did not discuss in this paper) can be important especially for embedded targets, and we expect different code generation techniques to vary widely in both speed and code size, likely leading to another dimension of trade-offs.

ACKNOWLEDGMENT

This work has been supported by the strategic research area ELLIIT.

Paper D

Finding Fast Action Selectors for Dataflow Actors

Gustav Cedersjö, Jörn W. Janneck and Jonas Skeppstedt
Department of Computer Science, Lund University

ABSTRACT The parallel structure of dataflow programs and their support for processing streams of data make dataflow programming an interesting tool for doing stream processing on parallel processing architectures. The computational kernels, the actors, of a dataflow program communicate with other actors via FIFO channels. The actors in the dataflow model used in this paper may perform different actions depending on the state of the actor and on the data that has been sent to the actor that is present on its ingoing channels.

For this kind of dataflow programs, decisions on what to do in an actor in a given state has to be made at runtime in a process called *action selection*. Each action is associated with a set of conditions on the state and the input channels. All conditions must be fulfilled for the action to be selected, and the task of the action selector is to test different conditions to select an action.

This paper builds upon previous work on the *actor machine*—a machine model for dataflow actors where the action selection is central. We present two heuristics that based on profiling data creates fast action selectors using the actor machine. The heuristics are implemented in the Tycho Dataflow Compiler and are evaluated using a video decoder written in Cal.

D.1 INTRODUCTION

Recent shift towards more parallel computing platforms and the popularization of stream applications such as signal processing, video encoding and cryptography has renewed the interest in dataflow programming. Dataflow programs are graph structured, where the nodes are computational kernels, called actors, and the edges are unidirectional channels through which the actors communicate. There are several popular tools and programming languages for different variants of this general model, including Simulink from MathWorks, LabVIEW and G from National Instruments, Cal [12] and its dialect RVC-CAL [48] that is used in video coding and is standardized by ISO/IEC, and Esterel [59] that is used in safety critical control systems.

Dataflow programs are often classified by properties that make them amendable for some efficient implementation techniques. Synchronous dataflow [1] and cyclo-static dataflow [7] are examples of classes that enable efficient scheduling of the actors at compile time. The dataflow programs in this paper are written in RVC-CAL, and contains actors from a broad class of dataflow where many of the efficient implementations techniques for the more restricted classes can not be applied. Throughout the rest of the paper we use the name Cal to mean both Cal and its dialect RVC-CAL.

Listing D.1 shows a simple dataflow actor written in Cal. It has three ports to which channels are to be connected—one input port `X` and two output ports `Y` and `Z`. The actor defines two actions `A1` and `A2`. If at least one token on port `X` is available and its value is positive, then action `A1` can be executed. This actor also specifies a priority between the actions, meaning actions of lower priority can only be executed when the higher prioritized actions cannot. The effect of the priority in this example is that action `A2` can only be executed if two or more tokens are available on `X` and the value of the first token is negative.

Software implementations of this kind of actor usually alternates between *action selection* and *action execution* where the action selection is done by testing the conditions of the actions until an action is found for which all conditions are fulfilled. The order in which these conditions are tested can affect the execution time of the actor. Traditionally, software implementations of Cal actors have been testing the condition of the actions in action priority order, in example `Foo` starting with action `A1` by testing if there is a token on port `X`, and in that case test if the value is positive, and if any of those two conditions are not fulfilled continue with action `A2` by testing if there are two tokens on port `X`.

A weakness with this approach is that it does not remember what is has

```
actor Foo () X ⟹ Y, Z:

A1: action X: [v] ⟹ Y: [v]
    guard v ≥ 0
    end

A2: action X: [v, w] ⟹ Z: [v + w]
    end

    priority A1 > A2; end

end
```

Listing D.1: A simple actor, written in Cal.

tested. Assuming the input port is empty, this action selector will first test if there is one token (for action A1), and then immediately test if there are two tokens (for action A2) which can never be true. Also, if the input is a dense stream of mostly negative values, it might be more efficient to start by testing if there are two tokens available, and then test if the value of the first token is positive.

In this paper we develop heuristics, based on execution statistics, to find efficient action selectors for the actors of a video decoder. The heuristics are described on the *actor machine*—an abstract machine for dataflow actors—and are implemented in the Tÿcho Dataflow Compiler. They are evaluated by the number of tested conditions, and the number of clock cycles on a Power processor.

The paper continues with related work in section D.2 and D.3, and then describes our work in section D.4, D.5 and D.6, and finally concludes with section D.7.

D.2 RELATED WORK

To our knowledge, the most used compiler for RVC-CAL is the Open RVC-CAL Compiler (Orcc) [15]. This is also the compiler with the most complete support for RVC-CAL. The C and C++ code generators for this compiler use the traditional approach to action selection, briefly described in section D.1.

Apart from the action priority, mentioned earlier, Cal actors can also define a schedule that restricts when actions can be selected. The schedule is described as a finite state machine, where each transition is associated with a set of actions that can be selected using that transition. The C and C++ backends of Orcc takes care of both the schedule and the priorities at compile time. The schedule transitions are implemented by unconditional branches, and the priorities are handles by testing the actions in an order that satisfies the action priorities.

The problem of testing conditions whose value can be inferred from previous tests is solved by generating code that makes it possible for an optimizing compiler to remove redundant tests in many cases.

One difference to the actor machine that is relevant in this paper is that the actor machine can reason about the order of tests in ways that are not possible here, where conditions are more tightly associated with specific actions.

The profiling technique and design flow used in this paper is inspired by the work of Casale Brunet et al. [60] where a high-level platform-independent profiled simulation is used to derive a causation trace of the execution. The causation trace is used to guide mapping, scheduling and buffer sizing when compiling the program.

D.3 ACTOR MACHINE

This paper builds upon previous work on the actor machine, most notably on [44], where the traditional approach to action selection, used for example by Orcc, is compared to action selectors based on the actor machine.

The actor machine [4] is an abstract machine for dataflow actors, that can represent a broad class of dataflow actors, even broader than that of Cal.[1] The main focus of the actor machine is on the action selection process which is represented by a *controller*. One controller can represent several possible action selectors, and can be *reduced* to a single action selector using a simple selection process described in section D.3.2.

D.3.1 Controller

The controller is a finite state machine where the state represents knowledge about the conditions. A condition is either known to be true (1), known to be false (0) or unknown (_). Example `Foo` in Listing D.1 contains three conditions:

[1] Actor machines can represent actors with actions that require more tokens than they consume, which is not possible in Cal but is possible in for example computation graphs.

- $c_1 = \text{tokens}(\mathsf{X}) \geq 1$, the availability of one token on input port X,
- $c_2 = \text{peek}(\mathsf{X}, 1) \geq 0$, the value of the first token on port X being positive, and
- $c_3 = \text{tokens}(\mathsf{X}) \geq 2$, the availability of two tokens on input port X.

The state of the controller for this actor can be represented by a 3-tuple of knowledge (c_1, c_2, c_3), where for example $(1, 0, \_)$ means that the first condition is true, the second is false and the third is unknown. This representation of knowledge enables the controller to avoid unnecessary tests.

There are three kinds of transitions between controller states: TEST, EXEC and WAIT. The TEST transitions test a condition and proceed to a state that "knows" about result of the test. The EXEC transitions execute an action and proceed to a state where knowledge about conditions that might have changed have been removed. If the controller reaches a state where it knows that there are too few tokens available to select any action, it has to wait and remove the knowledge about the absent tokens and try again later. This is done with the WAIT transition.

There are two relations between conditions that are important for the construction of an actor machine. The first relation is a dependency $c_1 \rightarrow c_2$, meaning c_1 can not be tested unless c_2 is true. Dependencies can be used to describe that there need to be a token available to be able to test the value of that token. The other relation is implication $c_1 \implies c_2$, meaning if c_1 is true, then c_2 is also true. In the example above, we have three relations on the conditions:

- The dependency $c_2 \rightarrow c_1$ means that a token must be available to test if its value is positive.
- The implication $\neg c_1 \implies \neg c_3$ describes that two tokens can not be available if one token is not.
- The last implication $c_3 \implies c_1$ is the dual of the previous: two available tokens implies one available token.

Starting with the state vector $(\_, \_, \_)$ and using the relations described above, we can derive a controller for actor Foo that describes several possible action selectors for this actor. Figure D.1 shows a visual representation of the resulting controller. The ellipses represent the states, the rectangles represent the EXEC transitions, the diamonds represent the TEST transitions and the rings

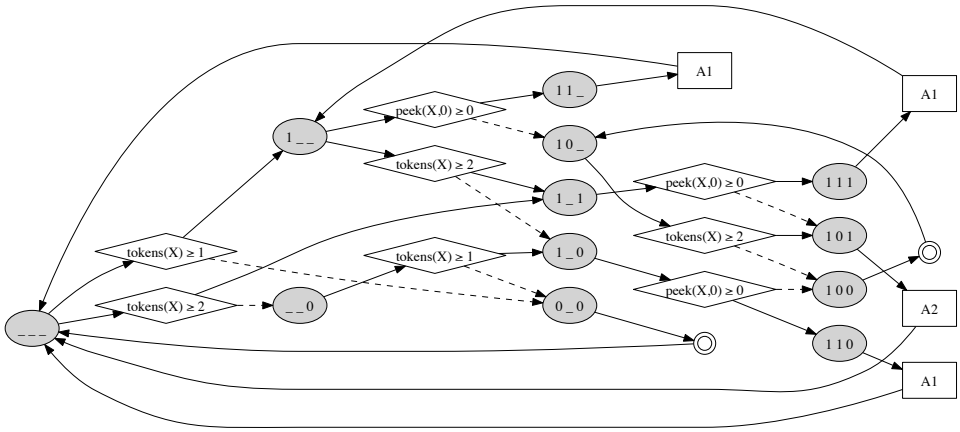


Figure D.1: Actor machine controller of actor **Foo**. The gray nodes represent states, and the white nodes represent transitions.

represent the WAIT transitions. The dashed edges should be followed if the condition is false.

From two of the states in the controller in Figure D.1, state $(\_, \_, \_)$ and $(1, \_, \_)$, more than one transition is possible. In state $(\_, \_, \_)$, either c_1 or c_3 can be tested, because both are unknown and neither of them depends on another condition. In state $(1, \_, \_)$ either c_2 or c_3 can be tested, because both are unknown, $c_2 \rightarrow c_1$ is satisfied and c_3 does not depend on another condition.

D.3.2 Controller reduction

To execute a controller, only one of the transitions from a state can be taken. The choice of transition can be made arbitrarily, but it affects the performance in general. The choice can be made at compile time, where the time to make a decision is not as critical as in runtime. (On the other hand, the decision could be a better informed decision at runtime.) By reducing the controller at compile time, the size of the program code is also reduced.

The process of selecting transition can be made by successive *reductions* of the controller. Reduction is the process of selecting a non-empty subset of the transitions in each state and pruning all unreachable states. A fully reduced controller is a controller where all states contain exactly one transition.

D.4 REDUCTION HEURISTICS

We have developed two heuristics for reducing the actor machine controller. Neither of the reducers produce a fully reduced controller in general, but it might do it in some cases. Both heuristics relies on profiling information from the execution of the actors. The first is based on action firing frequencies and the second is based on the probability of conditions to be true. The reducers are implemented in the Tÿcho Dataflow Compiler.

D.4.1 Profiling

The Tÿcho Dataflow Compiler can compile Cal programs to C programs with facilities for producing execution traces of the actor machine controllers. By changing the preprocessor definition TRACE_INSTRUCTION to true when compiling the C program, the compiled program will report the sequence steps that the controllers in the program take. The trace is a text file with one line per controller transition. Each line describes which actor that performed a controller transition and what kind of transition it was. In addition to this information, EXEC transitions also report which action it executed, and TEST transitions which condition it tested and the result of that test.

To gather statistics about the transitions and conditions of a dataflow program the following steps are taken. First, the dataflow program is compiled to a C program. This C program is then compiled with the tracing facilities enabled. The resulting executable is then executed with typical input stimulus. The resulting trace file is later analyzed to calculate statistics as input to the reduction heuristics.

The usefulness of the statistics as a basis for a reducer depends on the input being typical for that application. It is possibly also affected by how the actor machine controller is reduced. Assuming condition z_1 and z_2 are correlated, and the controller is reduced such that condition z_2 is only ever tested if for example z_1 is true, then the resulting trace will not give any information about the probability of z_2 when z_1 is false.

D.4.2 Reduction based on action execution frequency

The first reduction heuristic is based on action execution frequency, and it tries to find the shortest hypothetical sequence of TESTs that leads to the most frequent action. The sequence is hypothetical in the sense that it does not have to be possible for real, it just have to be present in the controller. The rationale for this reducer is that the most frequent action has the highest probability to be selected. The reason for selecting the shortest sequence is that all other se-

quences contain TESTs for conditions that are not needed in order to select this action.

The reducer calculates the shortest sequence of transitions up to and including the next EXEC or WAIT, where the length of this sequence is a pair (f^-, t) that is compared in lexicographical order, first comparing f^- and then breaking ties with t. If the sequence ends with an EXEC then f^- is the negation of its action frequency, and if the sequence ends with a WAIT then $f^- = 1$. This gives the EXEC transitions of the most frequent action the smallest value and all WAIT transitions a value larger than any EXEC. The second part of the length, t, is the number of TEST transition on the path.

D.4.3 Reduction based on condition probabilities

The second reduction heuristic is arguably much simpler in its description and implementation. It is based on the fact that when actor machines are constructed in the compiler, the conditions of the actions are most often constructed such that they have to be true for the action to be selected. The heuristic therefore selects the conditions with the highest probability to be true.

If there is an EXEC transition from the current state, then all EXEC transitions are selected. Otherwise, if there is a TEST transition in the current state, then p_{max} is calculated as the highest condition probability among the TEST transitions from this state, and all TESTs with a probability $p \geq p_{max} - d$ are selected, where $d = \frac{1}{10}$ is a threshold to be able to select more than one transition for later reductions. If neither an EXEC or a TEST is available, then the WAIT transition is selected, if there is one.

D.5 EXPERIMENTAL SETUP

To evaluate how the reduction heuristics affect the performance, we implemented the reducers in the Tÿcho Dataflow Compiler and evaluated them using a simple video decoder. The program that we used for evaluation is the reference implementation of the MPEG-4 Part 2 Simple Profile decoder in the Reconfigurable Video Coding framework. This program was compiled to C using the Tÿcho Dataflow Compiler and the resulting C program was compiled with gcc 4.8.2 with optimization flag -O2.

We executed the programs and counted how many TESTs the different controllers needed to decode a given video sequence. We also measured the execution time using the ASIM Power Architecture simulator [61]. The simulation was performed without cache to emulate an embedded system.

To estimate the space of possible controllers, we created 100 instances of the program where the reduction was made by random choice. We then measured number of TESTs and the execution time of these programs to use as a reference when comparing the other reducers. Since the reduction heuristics we developed does not produce fully reduced controllers, we combined them with a random reducer to to see how the space of possible controllers after the reduction was reduced.

D.6 RESULTS

This work builds upon previous work by Cedersjö and Janneck [44] where random reductions of actor machine controllers are compared to the traditional action selection approach. The two approaches were compared using the parser actor of a similar video decoder—the most complicated actor regarding action selection—by counting the number of performed tests. Because none of the randomly reduced controllers retest known conditions, all of the 1000 randomly reduced controllers in that paper required fewer tests than the traditional approach did.

In this paper we compare controllers from two reduction heuristics to random reductions for a whole dataflow program rather than just a single actor. Figure D.2 shows a histogram over the number of TEST transitions that instances of the example program need to decode the example video. The plot is divided into three parts by the following reduction strategies:

random reduce the controller by selecting one transition at random in each state.

action-frequency first reduce the controller with using the heuristic based on action frequency, described in section D.4.2, and then apply random reduction.

condition-probability first reduce the controller using the heuristic based on condition probabilities, described in section D.4.3, and then apply random reduction.

Figure D.2 shows that the controllers produced by the two heuristics require fewer transitions than the randomly reduced controllers. This also shows that 100 random reductions is not enough to find the extremes in the space of possible reductions.

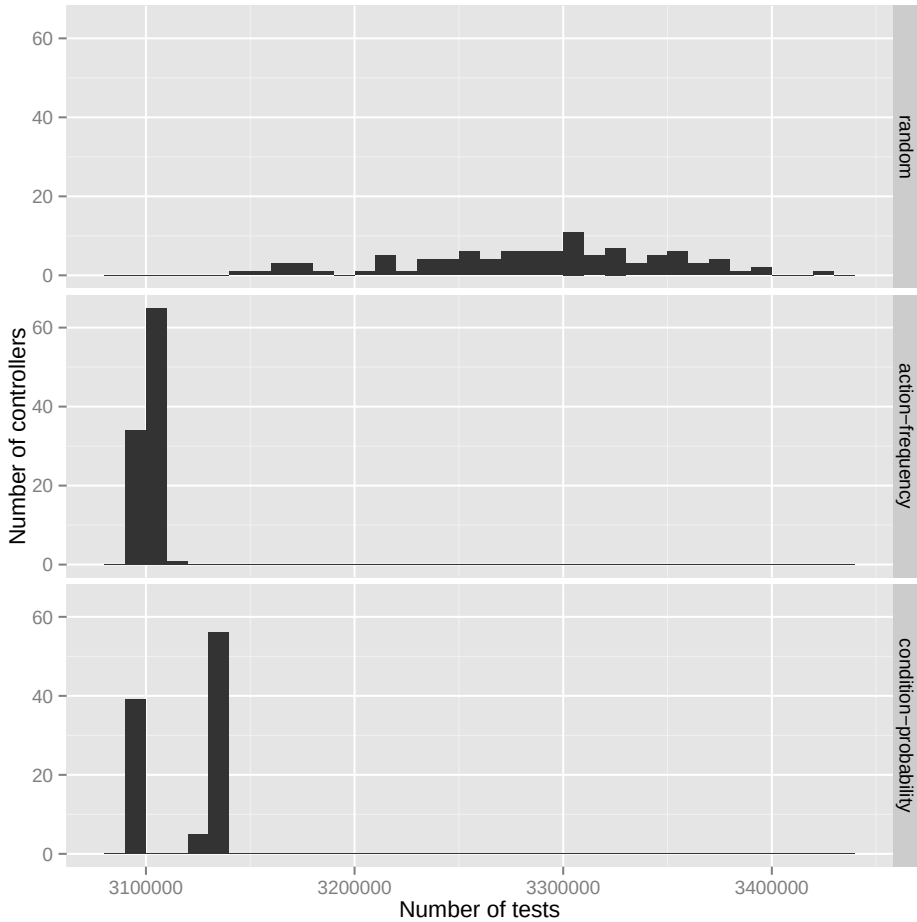


Figure D.2: Histograms over the number of tests required to decode a video sequence (foreman, 5 frames, QCIF). The histograms are grouped by the reducer that was used when compiling the program.

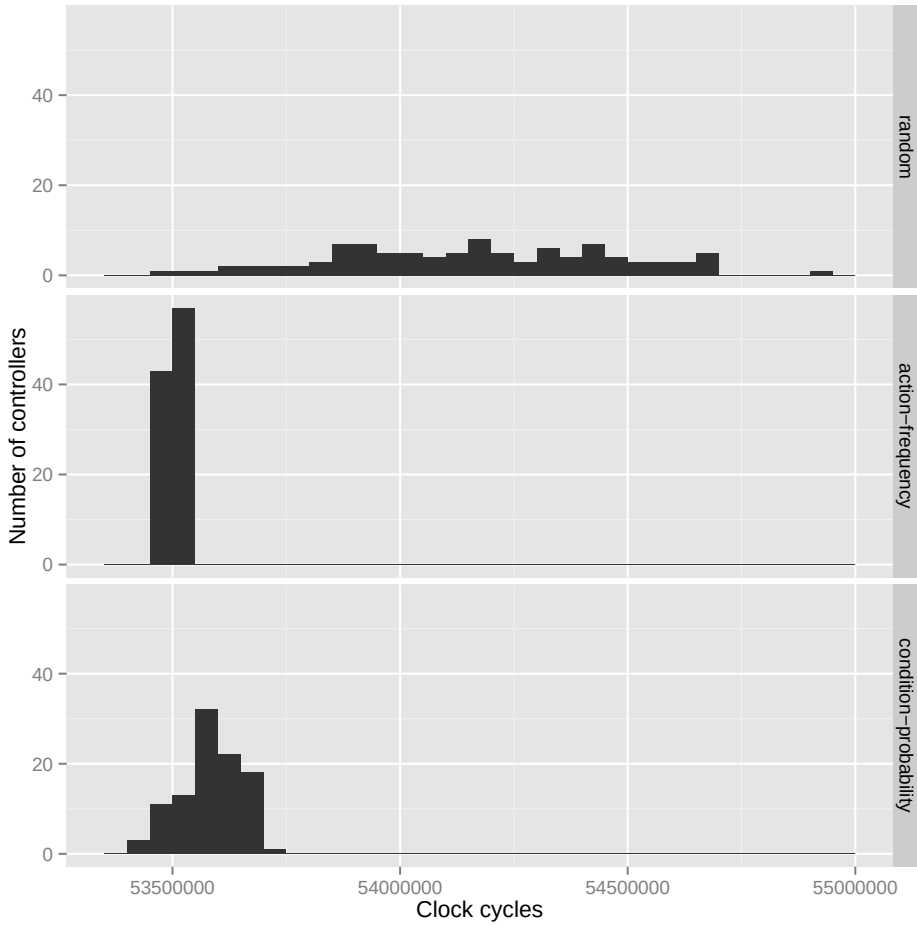


Figure D.3: Histograms over the number of clock cycles required by a Power processor to decode a video sequence (foreman, 5 frames, QCIF). The histograms are grouped by the reducer that was used when compiling the program.

Since different tests take different amount of time to execute, and also because an optimizing C compiler has the possibility to reduce the number of tests, we measured the execution time of the optimized and compiled C programs. Figure D.3 shows a histogram over the number of clock cycles on a Power processor using the same programs as in Figure D.2. The figure shows that the fastest of the randomly reduced programs are approximately as fast as programs where the heuristics were used.

D.7 CONCLUSIONS

In this paper we presented two heuristics for creating fast action selectors for dynamic dataflow actors. We compared the performance of the action selectors that these heuristics produce, both in terms of the amount of tests they perform and the execution time of the whole dataflow program.

Paper E

Processes and Actors: Translating Kahn Processes to Dataflow with Firing

Gustav Cedersjö and Jörn W. Janneck
Department of Computer Science, Lund University

ABSTRACT Dataflow programming is a paradigm for describing stream processing algorithms in a manner that naturally exposes their concurrency and makes the resulting programs readily implementable on highly parallel architectures.

Dataflow programs are graph structured, with nodes representing computational kernels that process the data flowing over the edges. There are two major families of languages for the kernels: process languages and languages for dataflow with firing. While processes tend to be easier to write, the additional structure provided by the dataflow-with-firing style increases the analyzability of dataflow programs and supports more efficient implementation techniques.

This paper seeks to combine these benefits in a principled manner by constructing a family of translations from a process language to dataflow with firing. In order to formally relate these descriptions, we first introduce a notion of firing to the semantics of Kahn processes, which allows us to give a precise

definition of equivalence between programs written in these different styles. Then we introduce a family of translations between them and and show that they retain meaning of a program. The presented language and its translation has been implemented in a compiler for the dataflow programming language Cal.

E.1 INTRODUCTION

Dataflow programming is a graph-based programming model, where the nodes perform computation on the data that flow over the edges. In the dataflow models we use in this paper, the edges represent buffered, lossless and order-preserving channels. Nodes may have local state variables that are updated throughout the execution, but there are no mutable state variables that are shared between nodes. All communication is done by sending data items (*tokens*) over the channels.

Dataflow programs exhibit a lot of concurrency, because each node can execute independently of the other nodes whenever it has data to process. It also tends to create small modules with few dependencies, which is good for modularity.

There are two major families of languages for expressing the computation in the nodes—process languages and languages for dataflow with firing. Process languages, such as that of Kahn [17], describe the computation in the nodes as sequential programs that explicitly read and write on the channels. Languages for dataflow with firing are instead structured around the concept of a firing, typically describing a set of actions that can be fired upon given conditions, where each action has a known number of tokens it consumes and produces. The difference in how the languages are structured affects how common idioms are expressed—where process languages can use if and while statements to control the execution, languages for dataflow with firing use the firing conditions of the actions to achieve the same control flow.

Figure E.1 and E.2 show a program that computes the sum of n tokens, both as a Kahn process and as dataflow with firing. The process version is arguably much simpler, because the control flow better follows the text of the program.

Dataflow programming languages are often associated with a particular execution model or a few different models. Kahn processes, for example, are typically executed using threads or with demand-driven cooperative scheduling, as described in [62], and this paper introduces another execution model for Kahn

```
process SumN() X, N ⟹ Sum :
  n; sum; x;

  repeat
    N ⟶ n;
    sum := 0;
    while n > 0 do
      X ⟶ x;
      sum := sum + x;
      n := n − 1;
    end
    Sum ⟵ sum;
  end
end
```

Figure E.1: A process that computes the sum of *n* tokens.

processes. For dataflow with firing, there are several execution models that take advantage of the firings to create an efficient implementation.

Synchronous dataflow [1] is a model where the number of tokens an actor consumes and produces is the same in every firing. For programs written in this model and for a slightly more general model called cyclo-static dataflow [7], a schedule can be completely determined at compile-time, removing the need for scheduling decisions at runtime. However, not all parts of a program need to be cyclo-static or synchronous dataflow to take advantage of these implementation techniques, as demonstrated in [63] for StreamIt, and in [3] for Cal. Also, [64] and [65] show in two different ways that actors with dynamic token rates can be composed, effectively creating a semi-static schedule of the composed actors. Section E.6.1 discusses a few implementations of dataflow with firing, comparing them to a traditional process implementation.

There are also other benefits of the firing semantics. One is the possibility to record traces of the action firings and create a dependency graph between the firings of a particular execution. In [60], such traces are used to guide the choice of implementation parameters in a design space exploration of a dataflow program.

In this paper we combine the simplicity of processes with the benefits of

```
actor SumN() X, N ⟹ Sum :
   n; sum;

   start: action N:[nbr] ⟹
   do
      sum := 0;
      n := nbr;
   end

   add: action X:[x] ⟹
   guard n > 0
   do
      sum := sum + x;
      n := n − 1;
   end

   done: action ⟹ Sum:[sum]
   guard n ≤ 0 end

   schedule Start:
      Start (start) ⟶ Sum;
      Sum (add) ⟶ Sum;
      Sum (done) ⟶ Start;
   end
end
```

Figure E.2: An actor that computes the sum of *n* tokens.

having them represented as dataflow with firing by designing a Kahn process language with a translation to dataflow with firing. The main contributions of this paper are the translation from the Kahn process language to dataflow with firing, and a way of expressing action firings in the denotational semantics of Kahn processes. Additionally, we elaborate on how the Kahn process source program can be transformed to the recursive functions of its denotational semantics—a detail that is only sketched by Kahn [17].

This paper continues in section E.2 and E.3 with some background on Kahn processes, the process model that we have chosen to implement, and then a short introduction to Cal, the target language of the translation. Section E.4 introduces a process language whose translation to Cal is presented in section E.5. Section E.6 discusses the language design and the translation to dataflow with firing. Related work is discussed in section E.7, and finally, section E.8 concludes the paper.

E.2 PROCESS MODEL

The process model we use in this paper is the one of Kahn [17], often referred to as Kahn process networks. A process is described as a sequential program that can communicate with other processes via blocking reads and non-blocking writes on channels. Kahn showed that a network of such processes always produces the same values on the channels, irrespective of how their executions are interleaved.

E.2.1 Semantics

The semantics of Kahn processes have been described in [17], and its details are beyond the scope of this paper. We will, however, discuss some of its building blocks to show the correctness of the translations we are presenting in this paper.

The semantics is denotational rather than operational, and the processes are described as functions on sequences of values. The sequences may be of finite or denumerably infinite length. There is a complete partial order on sequences called the prefix order $\sqsubseteq$, with $a \sqsubseteq b$ if and only if a is the initial segment of b. The functions that describe the processes must be monotonic on this partial order, meaning $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$. Another way to describe prefix monotonicity is that if such a function applied to sequence a yields the result r, and the same function is applied to sequence b that starts with a, then the result will start with r. The monotonicity describes an important property of the execution of processes, viz. that a process cannot change the output it has

```
actor Merge() X, Y ⟹ Z :
  action X:[v] ⟹ Z:[v] end
  action Y:[v] ⟹ Z:[v] end
end
```

Figure E.3: Non-deterministic merge actor.

already produced. The functions must also be Scott-continuous, which on the prefix order means that in addition to being monotonic, a function may not depend on whether an input sequence is finite or infinite. Kahn processes are by construction Scott-continuous functions.

A network of processes is described as an equation system where the data on the communication channels are variables and the processes are continuous functions over these variables. The semantics of the program is the smallest solution to the equation system with respect to the prefix order. This solution is unique and can be computed, or if infinite, arbitrarily well approximated, with fixed point iteration. A consequence of the solution being unique is that the execution order of the processes cannot affect the data they produce.

E.3 CAL

Cal [12] is a language for describing actors of dataflow with firing, originally developed as part of the Ptolemy project [66]. A variant of Cal has been standardized by MPEG in ISO/IEC 23001-4:2014 for describing video codecs.

An actor consists of *ports*, *state*, *actions* and additional constraints on when actions can be fired. Figure E.2 shows an actor written in Cal. It has three actions, tagged with `start`, `add` and `done`. Two of the actions also have a guard, i.e. a boolean expression that must be true for the action to fire. The actions are also, in this example, constrained by an action schedule—a finite state machine that controls which actions can be fired.

Cal can express computation that is not possible using Kahn processes, namely actors that are not monotonic on the prefix order or not even deterministic. The `Merge` actor in figure E.3 is an example of a non-deterministic actor. It takes a token either from port `X` or port `Y` and puts its value on `Z`.

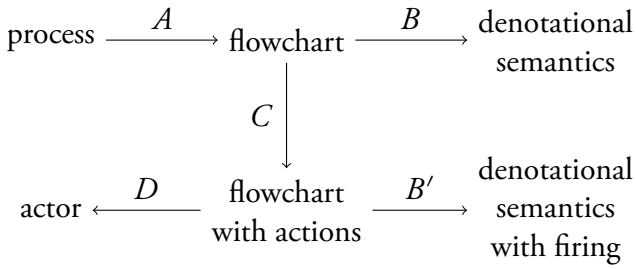


Figure E.4: An overview of the transformations presented in this paper.

E.4 PROCESS LANGUAGE

In this section we describe a small language for writing Kahn processes, which we in the next section translate to dataflow with firing. Figure E.4 shows an overview of the transformations that are performed on the processes. We will refer to this figure for each transformation we introduce.

E.4.1 Language Grammar

The grammar of the language is shown in figure E.5. Some parts of the grammar are not described in this paper, namely *expression*, *identifier* and *type*, and for those productions we use the corresponding productions from the Cal language report [12]. The process example in figure E.1 is written in this language, and later in the paper, there are a few more examples.

A process begins with the keyword **process** followed by the name of the process. Then follows a list of formal parameters in parentheses, and the actual parameters for these are bound at compile time, when instantiating the dataflow graph. After the parameter list comes the input and output port declarations. These are also bound to the communication channels when instantiating the graph.

The body of the process starts with a list of variable declarations, followed by the process description. The process description is either repeated, indicated by the **repeat** keyword, or just executed once, in case the **begin** keyword is used. The statements in the process description is what the process executes at run-time.

There are five kinds of statements of which three of them are known from many other imperative programming languages: if statements, while loops and assignments. The read and write statements are central for the semantics of

Kahn processes. The read statements look like this

Port ⟶ variable

and reads one token from Port, which must be an input port, and assigns the value to variable. The read is blocking, which means that the execution will not proceed unless there is a token available. The write statements, however, are non-blocking and look like this:

Port ⟵ expression

The expression is evaluated and the value is written to Port, which must be an output port.

E.4.2 Semantics

The semantics of this language is the semantics of Kahn processes, as described in [17], where the processes are viewed as functions on sequences. We use $[v_1, v_2, \ldots, v_n]$ to denote a sequence of n elements, and $[v_1, v_2, \ldots]$ to denote a sequence of infinite length. Concatenation, denoted $X.Y$, is the sequence that starts with X and continues with Y, or just X if X is infinite. We use $[\,]$ to denote an empty sequence.

The first step towards describing a process as a function on sequences is to describe it as a flowchart with one node in the flowchart per statement. This step is labelled A in the overview in figure E.4.

The read and write statements are represented by input/output nodes (parallelograms). Assignments are represented by process nodes (rectangles). The branching statements **if** and **while** are represented by groups of nodes: one decision node (diamond) for the condition test, one subchart for each alternative execution path, and a connection node (small circle) where the control flow converges. Entry points and exit points are represented by terminal nodes (rounded rectangles), labeled start and stop, respectively.

The flowchart is then translated to a function on sequences. This is transformation B in the overview in figure E.4. The denotational semantics of Kahn processes is defined on this form, and Kahn refers to the methods of McCarthy [67] on how to do the translation.

Each node in the flowchart is translated to a function that refers to its successor nodes for the continued execution. The functions are parameterized by the state variables and input streams. For simplicity, we only consider one state variable v, one input sequence X and one output sequence, but later we gen-

entity = "process" *identifier* "(" *parameters* ")" *ports* "==>" *ports* ":"
  {*declaration*}
  process
  "end".

parameter = [*type*] *identifier*.
parameters = [*parameter* {"," *parameter*}].

port = [*type*] *identifier*.
ports = [*port* {"," *port*}].

declaration = [*type*] *identifier* [":=" *expression*] ";".

process = ("repeat" | "begin") {*statement*} "end".

statement = *if* | *while* | *read* | *write* | *assignment*.

if = "if" *expression* "then" {*statement*} ["else" {*statement*}] "end".

while = "while" *expression* "do" {*statement*} "end".

read = *identifier* "—>" *identifier* ";".

write = *identifier* "<—" *expression* ";".

assignment = *identifier* ":=" *expression* ";".

Figure E.5: Simplified grammar of a process language.

eralize this to tuples. A start node *start* initializes the state of the process,

$$start(v, X) = next(v', X), \tag{E.1}$$

where v' is the initial value of v and *next* represents the successor node in the flowchart. A stop node *stop* represents the end of the execution and is therefore the empty sequence:

$$stop(v, X) = [\,].$$

A connection node *conn* is defined as its successor *next* and can be omitted. An assignment node is defined as

$$assign(v, X) = next(f(v), X),$$

where f computes the new value of v and *next* is the successor node. A conditional node, *cond* has two possible successor nodes, *true* and *false*, and is defined as

$$cond(v, X) = \begin{cases} true(v, X) & \text{if } p(v), \\ false(v, X) & \text{otherwise,} \end{cases}$$

where p is the predicate on the state variables that is the condition of the node. A write node *write* that writes a value $f(v)$ derived from the state to the output port is defined as follows:

$$write(v, X) = [f(v)].next(v, X). \tag{E.2}$$

Finally, a read node that reads a token and assigns it to v is defined as

$$read(v, X) = \begin{cases} next(h, T) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases} \tag{E.3}$$

where the execution stops if there is no token available. The description of a process is completed by a function that hides the state variables and is defined as the start function with any values of the variables:

$$process(X) = start(\bot, X).$$

If a process function is constructed using only the functions described above, the process will be monotonic on the prefix relation, because extension of an input to the function can only affect extensions of its output. It will also be continuous, because the function cannot depend on the finiteness of the sequences. The function therefore represents a Kahn process.

```
process Delay () X ⟹ Y :
   v := 0;
   repeat
      Y ⟵ v;
      X ⟶ v;
   end
end
```

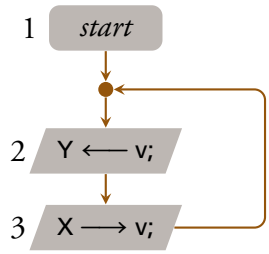


Figure E.6: A process definition and its corresponding flowchart with numbered nodes.

To generalize this to handle more state variables and input ports and output ports, the variables can be represented by a tuple, and the input sequences by a tuple of sequences and the output by a tuple of sequences as well. We extend the prefix order to tuples of sequences with $(X_1,\ldots,X_n) \sqsubseteq (Y_1,\ldots,Y_n) \iff X_1 \sqsubseteq Y_1 \wedge \ldots \wedge X_n \sqsubseteq Y_n$ and define the concatenation operation $(X_1,\ldots,X_n).(Y_1,\ldots,Y_n)$ as elementwise concatenation $(X_1.Y_1,\ldots,X_n.Y_n)$. The *assign* function will only change one of the elements of the variable tuple, *read* will only read from one of the streams to one of the variables, and *write* will only add to one of the elements of the output sequence tuple.

As an example of this translation, the process in figure E.6 is first transformed to the flowchart in the figure. The nodes of this flowchart are numbered 1 to 3, which corresponds to the functions $Delay_1$ to $Delay_3$. These functions are defined as described above:

$$
\begin{aligned}
&Delay_1(v,X) = Delay_2(0,X) && \text{by (E.1)},\\
&Delay_2(v,X) = [v].Delay_3(v,X) && \text{by (E.2)},\\
&Delay_3(v,X) = \begin{cases} Delay_2(h,T) & \text{if } X = [h].T,\\ [\,] & \text{if } X = [\,] \end{cases} && \text{by (E.3)}.
\end{aligned}
$$

The process is defined as follows:

$$Delay(X) = Delay_1(\bot, X).$$

We can now simplify function $Delay_3$ by substituting $Delay_2$:

$$Delay_3(v,X) = \begin{cases} [h].Delay_3(h,T) & \text{if } X = [h].T,\\ [\,] & \text{if } X = [\,]. \end{cases}$$

Figure E.7: A cyclic dataflow program with the **Delay** process.

Now we can see that $Delay_3$ is equivalent to its X argument and further simplify it to

$$Delay_3(v, X) = X.$$

Similarly we can simplify *Delay* by substituting $Delay_1$ and then $Delay_2$ and then finally $Delay_3$ and get the following:

$$Delay(X) = [0].X.$$

This definition is a very compact description of the process that captures the essence of the unit delay.

Figure E.7 shows a small cyclic network with one **Delay** process. The corresponding equation system in the semantics of Kahn processes is

$$\begin{cases} Y = Delay(X), \\ X = Y. \end{cases} \tag{E.4}$$

In general, there are several solutions to the equation systems of Kahn process networks, but the smallest solution with respect to the prefix order is what defines the semantics. In this case, the smallest solution to the equation system in (E.4) has an output sequence that is an infinite sequence of zeros: $Y = [0, 0, 0, \ldots]$.

E.5 TRANSLATION TO DATAFLOW WITH FIRING

The translation from process to dataflow with firing is done in three steps. (The corresponding labels from figure E.4 are shown in parentheses.)

1. Construct a flowchart of the process. (*A*)
2. Group nodes in the flowchart into actions. (*C*)
3. Create a dataflow actor from the actions. (*D*)

The first step makes it easier to reason about the statements in the code. The second step cannot be done arbitrarily without the risk of changing the semantics of the program. We therefore show how this step can be performed without changing what the process computes by making sure that the result of B and B' in figure E.4 are the same. In the third step, we take a process with actions and create a Cal actor with the same control flow.

E.5.1 Grouping nodes into actions

A dataflow actor is executed in atomic steps, called action firings. An action can only be fired if its conditions are fulfilled. We call these *firing conditions*. The tokens that an action reads, for example, must be present in order to fire. It also means that the number of tokens that an action requires must be known before firing that action. Typically, but not necessarily, it is even known at compile-time.

An example of a possible action is a chain of read, write and assignment nodes in the flowchart. Because a chain only has one possible control flow, where each statement is executed exactly once, the number of tokens that will be consumed and produced by executing a chain is known at compile-time. However, an action grouping is not correct just because it contains valid actions. It must also represent the same sequence-function.

To determine if a process with its statements grouped to actions is equivalent to the original process, we add action firing to the functions on sequences and check if the function is still the same. Referring to figure E.4, we check that the results of B and B' are the same.

If an action a is a sequence of statements that start with s and reads n tokens from X, then the atomicity is modeled as

$$a(v,X) = \begin{cases} s(v,X) & \text{if } X = [h_1,\ldots,h_n].T, \\ [\,] & \text{otherwise.} \end{cases} \tag{E.5}$$

The execution "continues" only when the input sequence X is long enough to execute the whole sequence of statements. The definition in (E.5) can be generalized to handle more than one input sequence by adding more firing conditions.

As an example, we use the Delay process in figure E.6 and its corresponding flowchart in figure E.6. Let the write statement of node 2 together with the read statement of node 3 be an action. The requirement for this action to fire is that X has at least one element, because of the read statement. The effect of

making an action of statement 2 and 3 is that the write will not be executed unless there is a token available for the read. To show that this translation is incorrect, we construct the function that corresponds to this translated process:

$$
\begin{aligned}
Delay'(X) &= Delay'_1(\bot, X), \\
Delay'_1(v, X) &= Delay'_a(0, X) && \text{by (E.1)}, \\
Delay'_a(v, X) &= \begin{cases} Delay'_2(v, X) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,] \end{cases} && \text{by (E.5)}, \\
Delay'_2(v, X) &= [v].Delay'_3(v, X) && \text{by (E.2)}, \\
Delay'_3(v, X) &= \begin{cases} Delay'_a(h, T) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,] \end{cases} && \text{by (E.3)}.
\end{aligned}
$$

This set of functions differs from the *Delay* functions in the following two ways. $Delay'$ contains a function $Delay'_a$ that describes the atomicity of $Delay'_2$ and $Delay'_3$, and all references to $Delay_2$ are instead references to $Delay'_a$.

A translation is correct only if its function is equivalent to the original function. If we apply *Delay* and $Delay'$ to the empty sequence

$$
\begin{aligned}
Delay([\,]) &= [0], \\
Delay'([\,]) &= Delay'_1(\bot, [\,]) = Delay'_a(0, [\,]) = [\,],
\end{aligned}
$$

we see that the translation is incorrect.

E.5.2 Action grouping schemes

We will present a series of action grouping schemes that yield correct translations from processes to dataflow with firing. We only consider grouping schemes that group chains of statement nodes, i.e. sequences of read, write and assignment statements. The condition nodes are translated to their own actions without any firing conditions that only designate its successor action in the execution.

There is a trivial action grouping that is always correct, that is the grouping where each statement becomes its own action. In this grouping, only the actions with a read statement will have a firing condition. This condition is also the same as the condition of the read statement itself. Let a be the action with one read statement r,

X ⟶ v;

whose successor is n:

$$a(v,X) = \begin{cases} r(v,X) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases}$$
$$r(v,X) = \begin{cases} n(h,T) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,]. \end{cases}$$

We can see that $a = r$. For an action a with a write or assignment statement s, the action is equal to the statement, because the action has no firing condition:

$$a(v,X) = \begin{cases} s(v,X) & \text{always}, \\ [\,] & \text{never}. \end{cases}$$

This grouping results in very fine grained actions, but dataflow actors are usually not written in this way, firstly because it is very tedious to write that many actions, secondly because it is usually not necessary, and thirdly because it is usually more efficient to write fewer and larger actions. The actions need to be scheduled, either at compile time or at runtime, and the more actions to schedule, the more time it tends to take. For these reasons, we study some grouping schemes with larger actions as well.

Regular expressions on sequences of statements. To describe different grouping schemes, we use regular expressions over sequences of statements. Let r, w and a denote the set of read, write or assignment statement, respectively. Alternation is denoted $x|y$ which is the union of all sequences in x and y. Concatenation xy is the set of sequences that are concatenations of a sequence in x with a sequence in y. The Kleene star x^* represents concatenation of any number of sequences in x, including zero sequences. As an example, the expression $(r|a)^*$ denotes any sequence of read and assignment statements.

Actions with only writes and assignments. If sequences of write and assignment statements, $(w|a)^*$, are grouped to actions, the resulting function of the action will be identical to the function of the statements. Write and assignment statements do not read any tokens, and the action will therefore have no conditions. Let s be the first statement in a sequence in $(w|a)^*$, followed by the rest

of the process, and a be an action with these statements:

$$a(v,X) = \begin{cases} s(v,X) & \text{always,} \\ [\,] & \text{never.} \end{cases}$$

Action a is trivially equal to statements s.

Actions with only reads and assignments. If sequences of read and assignment statements, $(r|a)^*$, are grouped to actions, the resulting function will have conditions on the lengths of the sequences corresponding to the number of read statements for each stream. Assume the read statements of an action a reads n tokens from X, the action will then be defined as

$$a(v,X) = \begin{cases} s(v,X) & \text{if } X = [h_1,\ldots,h_n].T, \\ [\,] & \text{otherwise,} \end{cases}$$

where s represents the execution starting at the first statement in the action. Since the statements of the action doesn't produce any output, we know that when the sequence is shorter than what all statements read, the output is empty:

$$\forall k < n.\, s(v,[x_1,\ldots,x_k]) = [\,].$$

That is also true for the action:

$$\forall k < n.\, a(v,[x_1,\ldots,x_k]) = [\,].$$

When the sequences, however, are long enough for all the read statements of the action, we get the following:

$$\forall k \geq n.\, a(v,[x_1,\ldots,x_k]) = s(v,[x_1,\ldots,x_k]).$$

Both when the inputs are sufficiently long and when they are not, the action represents the same function as its statements.

Actions with reads and assignments and then writes and assignments. The two regular expressions above can be combined to $(r|a)^*(w|a)^*$, accepting first reads and assignments and then writes and assignments. To study what affect this action has on the function, we first study the case with two consecutive

actions a_1 with statements from $(r|a)^*$ and a_2 with statements from $(w|a)^*$. As we have seen earlier, action a_1 requires all tokens that it reads to be present before proceeding, and action a_2 does not require anything to continue. An action a that contains the statements of a_1 followed by a_2, will have the same conditions as a_1, because a_2 does not have any conditions. The function that represents a will therefore be identical to the function that represents a_1 followed by a_2. To illustrate this, assume the simple case with one read statement r followed by one write statement w followed by another statement n. First, we make actions a_r of r and a_w of w. The functions for this now looks like the following:

$$
\begin{aligned}
a_r(v,X) &= \begin{cases} r(v,X) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases} \\
r(v,X) &= \begin{cases} a_w(h,T) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases} \\
a_w(v,X) &= w(v,X), \\
w(v,X) &= [v].s(v,X).
\end{aligned}
$$

The important part is that a_w will be the same for all groupings in $(w|a)^*$, making it possible to use w directly wherever a_w is used and eliminate a_w:

$$
\begin{aligned}
a_r(v,X) &= \begin{cases} r(v,X) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases} \\
r(v,X) &= \begin{cases} w(h,T) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases} \\
w(v,X) &= [v].s(v,X).
\end{aligned}
$$

If instead we make one action a of both r and w we get the same functions:

$$
\begin{aligned}
a(v,X) &= \begin{cases} r(v,X) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases} \\
r(v,X) &= \begin{cases} w(h,T) & \text{if } X = [h].T, \\ [\,] & \text{if } X = [\,], \end{cases} \\
w(v,X) &= [v].s(v,X).
\end{aligned}
$$

Grouping statements by $(r|a)^*(w|a)^*$ to actions preserves the semantics of the process.

E.5.3 Translation to dataflow actor

From the flowcharts with actions, the translation to dataflow with firing (transformation D in figure E.4) is quite straightforward. We use Cal as the target language for this last step, and we explain the translation with the SumN process in figure E.1 and the actions of figure E.8

When the actions of a process are identified, they are translated to Cal actions that perform the same task, and each action gets a unique action tag. We call these actions *statement actions*. From the grouping in figure E.8, the following actions are constructed.

```
a: action N:[temp] ⟹
do
  n := temp;
  sum := 0;
end

b: action X:[temp] ⟹
do
  x := temp;
  sum := sum + x;
  n := n − 1;
end

c: action ⟹ Sum:[sum] end
```

From the decision node, two actions are constructed, one that can be fired when the condition is true, and one when the condition is false. We call these pairs of *condition actions*. Note that these actions neither consume nor produce any tokens.

```
d_true: action ⟹
guard
  n > 0
end

d_false: action ⟹
guard
  not (n > 0)
end
```

Finally, an action schedule is created to ensure that the actor has the same control flow as the original process. The schedule will have one state for each statement action, and one state for each pair of condition actions. The transitions are then constructed to reflect the control flow of the process.

```
schedule A :
  A (a) ⟶ D;
  D (d_true) ⟶ B;
  D (d_false) ⟶ C;
  B (b) ⟶ D;
  C (c) ⟶ A;
end
```

The execution starts in state A with action a that reads how many numbers should be summed, and continues to state D. In D, there are two actions, one that can be executed when $n > 0$ and another action otherwise. Depending on which of the two is fired, the execution proceeds to either B or C. State B represents the body of the loop and when action b is fired the execution proceeds with the loop condition in state D again. After the loop, in state C, action c is executed an the output is produced. The execution then continues in state A to start a new summation.

The start node of the flowchart corresponds to the variable initialization of the process, and this is just copied to the resulting actor. This example does not have any stop node, but stop nodes are represented by states in the schedule without any transitions that leaves them.

The end result is not as polished as the handwritten Cal version in figure E.2, but it is not far from it.

E.6 DISCUSSION

E.6.1 Implementation efficiency

Even though this paper is not about efficient implementations of dataflow with firing, its existence is highly relevant for this work. Some restricted classes of dataflow with firing, such as synchronous dataflow and cyclo-static dataflow are well known for their efficient implementation techniques, but even dynamic dataflow can be efficienly implemented. To demonstrate the efficiency compared to a traditional process implementation, we implemented the proposed language and the translation to Cal in the Tÿcho dataflow compiler and designed a small program with four processes, depicted in figure E.9. We made

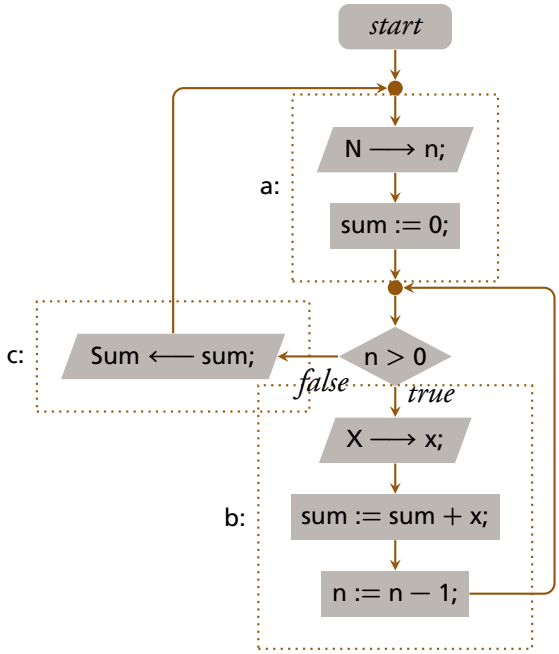


Figure E.8: Flowchart of the **SumN** process in figure E.1 with an example grouping.

Figure E.9: A program with four processes. **LFSR** is a linear feedback shift register, producing the $n = 10^7$ first numbers of a maximum length sequence of 16 bits, followed by a 0. **Even** filters out all odd values, forwarding only the even numbers to **Inc**, that increments the numbers by 1. **Sum** computes the sum of all numbers until it gets a 1 (the incremented 0 at the end of the stream) and prints the sum.

two process implementations; one in our proposed language that we compiled using the Tÿcho dataflow compiler, and one with goroutines in Go that we compiled using gc[1]. To get an indication of how much overhead the parallel descriptions induces in terms of scheduling and bookkeeping, we also made a sequential implementation in C that performs the same computation using a single loop.

In the Go implementation, the size of the channels that connects the goroutines affects the performance significantly. Figure E.10 shows the execution

[1] The other official Go compiler—gccgo—produced slower programs for this example.

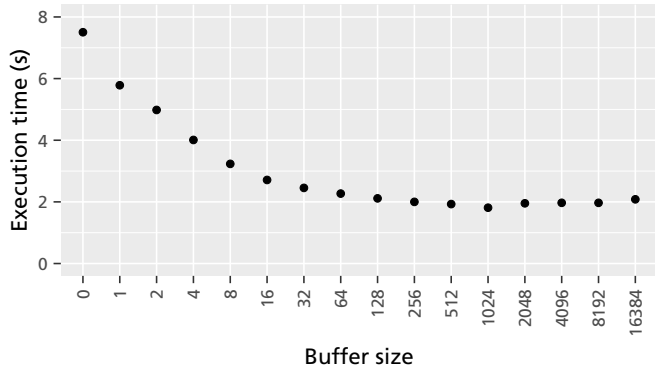


Figure E.10: Execution time of the Go program with different buffer sizes, where 0 indicates rendezvous communication.

Table E.1: Execution Time of the Program in Figure E.9

	Mean (ms)	Standard deviation (ms)
Go (gc)	1811.4	13.6
Kahn processes (Tÿcho)	203.0	3.7
Kahn processes (Tÿcho)[a]	158.3	2.8
Kahn processes (Tÿcho)[b]	64.9	2.5
C (Clang)[c]	64.6	2.8

[a] compiled with actor merging

[b] compiled with actor merging and buffer to variable conversion

[c] a sequential program that computes the same result

time for buffer sizes ranging from 0 to 2048, where 0 means rendezvous communication. Using large buffers in the program implemented with Tÿcho did not result in any significant performance difference.

The Tÿcho dataflow compiler can merge actors using actor machine composition, as described in [64], similar to what is done with RVC-CAL actors using the Open RVC-CAL Compiler (Orcc) in [65]. The resulting merged actor does not need to check that state of the channels that are connected to itself. If these self-loop channels are of size 1, they can be replaced by variables, which is also done by Tÿcho. Table E.1 shows the execution time of five different implementations:

1. the Go implementation with channel sizes set to 1024,
2. the Kahn process implementation compiled with Tÿcho,
3. the Kahn process implementation compiled with Tÿcho with actor merging,
4. the Kahn process implementation compiled with Tÿcho with actor merging and with buffers converted to variables, and
5. the sequential C program.

The Kahn process implementations that are transformed to dataflow with firing are clearly faster than the goroutine implementation, even without actor merging. With actor merging and the buffer to variable transformation, the programs is as fast as the sequential C program.

All measurements were performed on a computer with a quad-core 2 Ghz Intel Core i7 processor running OS X, and the sequential C program and the C programs produced by Tÿcho were compiled with Clang using the flag -O2.

E.6.2 Program simplicity

We believe that many dataflow actors could be much simpler described as processes, most importantly because the control flow of a process better follows the text flow of the source program than it would do in a dataflow-with-firing style. To get an indication on what kinds of actors would benefit from being expressed as processes, we have identified some patterns commonly used in Cal actors by studying the example applications from *orc-apps*[2]—a collection of Cal applications. The patterns we identified are the following:

1. Actors with only one action.
2. Fixed sequences of actions in the sense that action B always follows action A.
3. Iterative token production or consumption with an unknown number of tokens.
4. Actors that select different actions depending on the value of a particular token or state variable.

[2] https://github.com/orcc/orc-apps

Table E.2: Number of tokens in the source code of the actors and their respective processes.

	Actor	Process	Ratio	Patterns
SyntaxParser[a]	682	412	60%	2, 3, 4
Mgnt_DCSplit[b]	97	107	110%	1
Algo_Byte2bit[b]	122	81	66%	2, 3
Algo_SelectMB_4[c]	491	200	41%	2, 3, 4
Algo_PictureRecon…Sat…[b]	355	255	72%	2, 4
KeySchedule[e]	757	629	83%	2

[a] from JPEG decoder
[b] from MPEG-4 Part 2 decoder
[c] from MPEG-4 AVC decoder
[d] from AES cipher

In our opinion, pattern 1 is often clearly expressed in Cal and we see no point in rewriting such actors in a process style. However, expressing pattern 2, 3 and 4 is, in our opinion, more complex in Cal than it is in our process language. Fixed sequences of actions (pattern 2) involve at least two actions and an action schedule. When trying to follow the control flow of such an actor, the reader must consult the action schedule to see in which order the actions are fired. When expressed as a process, this pattern is simply a sequence of statements. Pattern 3 and 4 can in our process language be expressed by (respectively) a while-loop and if-statement, but encoding the same pattern in a Cal actor requires at least two actions whose firing conditions only differ in a guard expression, representing the branching condition.

A review of existing Cal applications and the feasibility of translating them to processes is out of the scope for this paper. We have, however, collected a few actors that use the patterns described above and translated them to our process language. We have not measured how readable they are, or how well the program text follows the control flow, but since the simplified control flow also results in smaller programs, we have measured the size of the programs in number of source code tokens. As the result in table E.2 shows, the actor with only one action (pattern 1) uses more tokens in the process version than the actor version, but the other patterns are expressed using fewer tokens in their process versions.

E.6.3 Language Design

The first objective of the language design is to enable programmers to write processes and have them executed as actors. The second objective is to make it easy to use processes in Cal programs. We have made some design choices motivated by these objectives.

The look and feel of the language resembles Cal to a large extent. The reason for this decision is to make it easier for Cal programmers to start using the language. Not only does it look like Cal, the types, statements and expressions are borrowed from Cal as well. By representing and computing values in the same way, the interaction between the two languages becomes straightforward. In the larger version of the language that is implemented in the Tÿcho dataflow compiler, the process description is actually implemented as a new language construct for Cal actors, rather than a separate language of its own.

E.6.4 More general groupings

We saw in the *Delay′* example that actions that start with writes and continue with reads are not equivalent to just the statements themselves. There are, however, other correct grouping schemes that recognize larger actions. The schemes we have looked at all consist of sequences of statement nodes, but there are examples where loops and conditionals can be part of actions as well. If, for example, the two execution paths of an if-statement have the same read and write pattern, they could be considered for inclusion in the same action. The same is true for loops where no communication happens. The grouping schemes that we have shown in this paper are therefore not the most coarse-grained.

E.7 RELATED WORK

E.7.1 Language

The process language we introduced makes it possible to write dataflow programs as combinations of processes and dataflow with firing, but it is not the first example of such combination. StreamIt [10] is a language for Synchronous dataflow—a flavor of dataflow with firing where the number of tokens consumed and produced by an actor is the same on every invocation. In version 2.1 of StreamIt the restriction of a fixed input and output rate was lifted, effectively making StreamIt a language for expressing both synchronous dataflow and Kahn processes. The implementation in [63] also treats them separately, heavily optimizing the parts with fixed token rates and dynamically scheduling

all interaction that deals with dynamic token rates. Our approach is instead to unify the two by translating the process to dataflow with firing. In StreamIt, however, the filters with dynamic token rates can not in general be translated to their current model for dataflow with firing, synchronous dataflow, because that model is not expressive enough.

E.7.2 Translation

In [30], Falk et al. presents a translation from Kahn processes expressed in SystemC to dataflow with firing by constructing a control flow graph in which all read statements precede the write statements in the basic blocks. This is equivalent to the most general action grouping scheme presented in this paper. The novelty of our work is the connection to the semantics of Kahn processes to show that the action grouping is correct.

One aspect of this work is that it enables programmers to write programs in an imperative style and have it executed with a different execution model that we believe is better in many ways. Related to this, Capriccio [68] is a threading package for C that enables programmers to write threaded code with blocking operations, have it executed using cooperatively scheduled coroutines and event-based operations instead of blocking ones. Similarly, Tame [69] and the work of Adya et al. [70] both try to unify the models of threaded with event-driven programs, partly using program transformations not unlike those we do.

Even though there are similarities between these systems and ours, we believe that the challenges differ very much. One of their main challenge is memory management for automatic variables. That is not an issue for our language, because the processes are essentially stack-free at the points where they need to be. Our main challenge, on the other hand, is to make sure that the exact semantics of Kahn processes is retained in the translation.

In [38], Lee shows a translation from dataflow with firing to Kahn processes, which is the reverse of our translation, and gives conditions on when this translation is possible.

E.7.3 Process model

In this work, we use Kahn process networks as our process model. There are other well-known process models, such as Communicating Sequential Processes [71] or pi calculus [72]. We chose Kahn process networks over these calculi because the communication model of Kahn process networks closely resembles the one of dataflow with firing. Because of this similarity, several

aspects of combining Kahn process networks and dataflow with firing have already been studied. In [38], Lee shows that a certain class of dataflow with firing is determinate by transformation to Kahn process networks. In [21], Kienhuis and Deprettere introduce a model for dataflow with firing that is able to describe Kahn process networks.

When proving the equivalence of the programs before and after action grouping, we extended the Kahn process model with a notion of actions. There is a related model, called Stream Based Functions [73] that instead extends a model for dataflow with firing with the expressiveness of Kahn processes. This model is used in Compaan [22] to transform nested loop programs to Kahn process networks.

E.8 CONCLUSIONS AND FUTURE WORK

We have introduced a Kahn process language and a family of translations from this language to dataflow with firing and Cal. It enables programmers to write their dataflow programs as processes and still use the efficient implementation techniques and analysis tools that available for Cal. For software implementations, if the number of processes exceeds the number of processors, which is the most common case, their corresponding actors can be merged to reduce communication overhead using actor machine composition [64] or by merging Cal actors [65]. For programmable hardware, Xronos have been shown to synthesize hardware directly from the RVC-CAL reference implementation of an MPEG-4 decoder [74]. On the analysis side, the profiling infrastructure of TURNUS uses execution traces of Cal programs to explore the design space of their implementations [60].

We have also elaborated on the translation from a Kahn process language to its denotational semantics, introducing a way of reasoning about process languages, building upon the works of Kahn [17] and in turn McCarthy [67]. This way of reasoning enabled us to introduce the concept of an action to the semantics of Kahn processes.

To conclude the paper, we have shown a way of giving programmers the simplicity of writing processes and at the same time treat them as Cal actors with all of its benefits. As future work, on the language side, we would like to investigate ways of letting programmers affect the action grouping. We would also like to integrate more features of Cal into the process language and vice versa. On the translation side, we would like to study action grouping schemes that can identify even larger actions.

Paper F

Tÿcho: A Framework for Compiling Stream Programs

Gustav Cedersjö and Jörn W. Janneck
Department of Computer Science, Lund University

ABSTRACT Many application areas for embedded systems, such as DSP, media coding and image processing, are based on stream processing. Stream programs in these areas are often naturally described as a graphs, where nodes are computational kernels that send data over the edges. This structure also exhibits large amounts of concurrency, because the kernels can execute independently as long as there is data to process on the edges. The explicit data dependencies also help making efficient sequential implementations of such programs, allowing programs to be more portable between platforms with various degrees of parallelism.

The kernels can be expressed in many different ways; for example, as sequential programs with read and write statements for the communication, or as a set of state transitions that can be performed and conditions for when these transitions can be made. Traditionally, there has been a tension between how the kernels are expressed and how efficiently they can be implemented. There are very efficient implementation techniques for stream programs with restricted expressiveness, such as synchronous dataflow.

In this paper we present a framework for compiling stream programs, that we call Tÿcho. It handles kernels of different styles and with a high degree of expressiveness, using a common intermediate representation. It also provides

Submitted to *ACM Transactions on Embedded Computing Systems*.

efficient implementation, especially for, but not limited to, the restricted forms of stream programs.

F.1 INTRODUCTION

F.1.1 Compiler Framework

GCC and Clang are well known compilers for C and C-like languages such as C++, Objective-C and OpenMP. Both compilers use language independent intermediate representations of programs, on which most optimizations are performed, i.e. GIMPLE [75] in GCC and LLVM IR [76] in Clang. The benefits of having a common intermediate representation is that optimizations that operate on that form become available to all input languages, and code generators for specific targets can generate code from all input languages.

These compilers are great for C-like languages, but not all languages are like C. For example, Haskell [77] is very different from C, and the most well known Haskell compiler, GHC, uses an intermediate representation, called Core [78], that is very different from GIMPLE and LLVM IR. Core captures the essence of Haskell programs and GHC performs optimizations and program transformations on that form. However, the LLVM IR is still used in the backend of GHC, where complementary optimizations are performed.

In this paper, we present a compiler framework for stream programs called Tÿcho. The core of its intermediate representation is language-independent, specific for stream programs, and supports optimizations that are important for stream programs. Similar to how GHC handles different optimizations on different representations, this framework performs optimizations related to stream programs and generates code that is further optimized by other compilers, such as GCC or Clang.

F.1.2 Stream Programs

A stream program is a parallel program that operates on streams of data. It is structured as a network of concurrently executing components that communicate by streaming data over channels, see Figure F.1. The components of these networks are computational kernels that read from and write to the channels and perform computation. Kernels may also have internal mutable state that influences the computation, but no state variables are shared between kernels.

A simple way to describe a kernel is as a *process*—a sequential program with read and write operations on the channels. In [17], Kahn describes the seman-

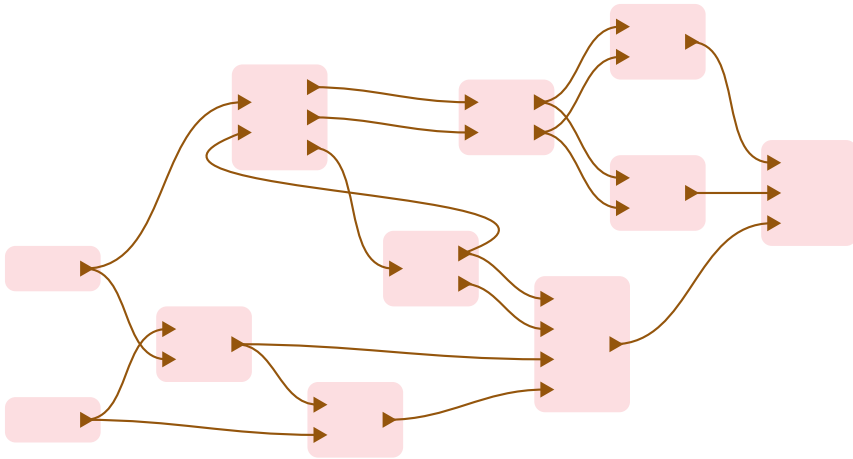


Figure F.1: High-level view of a stream program, where the boxes are computational kernels that communicate via its ports, the triangles, over the channels, represented by lines between the ports.

tics of a process language, often referred to as Kahn process networks, and shows that programs written in this language or style are deterministic—the output will always be the same for a given input—irrespective of how the processes are scheduled.

Dataflow with firing [38] suggests a different way of describing kernels, centered around the concept of *firing*. An actor—a dataflow kernel with firing—has a set of *transitions* that perform the computation, and each transition has a set of *conditions* that describe the requirements for a transition to be fired. If, for example, a transition reads a token from a particular channel, the conditions for that transition include "the presence of at least one token on that channel."

There is a trade-off between expressiveness and efficient implementations for stream programs, and many restricted forms of dataflow with firing has been developed to reach different points in this tradeoff. Synchronous dataflow [1] is subclass of dataflow with firing that can be scheduled at compiletime. Cyclo-static dataflow [7] is a slightly more general class with the same property. Boolean-controlled dataflow [9] is an even more general class that in some cases can be scheduled at compiletime. In this paper, we are primarily interested in very general dataflow with firing.

F.1.3 Problem

Stream programs can be realized on platforms with various degrees of parallelism, ranging from software implementations for a single CPU to processor arrays [32], GPUs [79] and FPGAs [80]. Different platforms have their own sets of challenges and optimization opportunities.

In this paper, we take a look at three aspects of efficient software code generation of stream programs for CPU-based platforms, with a focus on dataflow with firing:

- efficient transition selection for kernels with multiple transitions,
- efficient scheduling through kernel fusion, and
- efficient variable sharing between conditions and transitions.

The problem we are addressing is how these aspects can be handled with the additional requirements that

- the solution should be applicable to a general class of stream programs, and
- the solution should be language-independent.

An example for which the additional requirements do not hold is static scheduling of synchronous or cyclo-static dataflow programs. Even though it is language-independent, it is not applicable to a very general class of stream programs.

The proposed solution is a framework for compiling stream programs called Tÿcho. To meet the generality requirement, the kernel representation in Tÿcho is based on a machine model for dataflow with firing, called actor machine, that supports a broad class of stream program kernels. The actor machine is presented in Section F.2. Section F.3 describes the actor machine based kernel representation in Tÿcho, and presents frontends for two kernel languages of different styles, to illustrate the language-independence. Section F.4 addresses the problem of efficient transition selection in the presence of multiple transitions. The solution to efficient scheduling is presented in Section F.5, where a new actor machine fusion algorithm is presented and evaluated. Section F.6 describes two techniques to optimize variable initialization when variables can be shared between conditions and transitions. Section F.7 discusses related work and Section F.8 concludes the paper.

F.2 ACTOR MACHINE

F.2.1 Dataflow with Firing

Dataflow with firing is a family of stream programs with a concept, introduced by Dennis [6], called *firing*. The kernels are called *actors*, and a firing is an execution step of an actor. A set of *transitions* describes what steps an actor can take, and the act of making a transition is a firing. In a transition, an actor may (1) read input data items, or *tokens*, (2) update its internal state, and (3) produce output tokens. The number of tokens that are consumed and produced in a transition is predefined, but may vary among the transitions. Each transition is associated with a set of *conditions* that are the prerequisites for firing the transition. The conditions include, for example, the availability of the tokens that the transition consumes.

Examples of dataflow with firing include synchronous dataflow [1], cyclo-static dataflow [7] and Cal [12].

F.2.2 Actor Machine Controller

The actor machine [4] is a simple machine model for dataflow with firing. It is focused on the process of selecting transitions by testing conditions. It is permitting the expression of a wide range of stream program kernels. It is, for example, not limited to the statically schedulable programs, such as synchronous dataflow, or even deterministic programs. Nonetheless it supports important stream program optimizations, such as kernel fusion [4, 64], for creating efficient implementations. Tÿcho uses the actor machine as a basis for its kernel representation, and the main reasons are the generality in what kernels it can describe, combined with the efficient implementation techniques it enables.

The central component of the actor machine is its controller. It is a state machine where the set of states S encode knowledge about the conditions of a kernel. Each state $s \in S$ is associated with a set of instructions that lead to other states. There are three kinds of instructions: EXEC, TEST, and WAIT. An $\textsc{exec}(t, s')$ has two parameters: one transition $t \in T$, and one target state $s' \in S$, that when executed performs transition t and proceeds to state s'. A $\textsc{test}(c, s_t, s_f)$ has three parameters: a condition $c \in C$, and two states $s_t, s_f \in S$, and when executed it tests the condition and proceeds to s_t if true and to s_f otherwise. The target states s_t and s_f keep track of the knowledge whether c is true or false, to make sure that condition c is not retested needlessly. Some conditions, however, can change by an external event, e.g. the condition of there being tokens available for input could change because of some other kernel

```
actor Merge () A, B ⟹ C :
    a: action A:[v] ⟹ C:[v] end
    b: action B:[v] ⟹ C:[v] end
end
```

Listing F.1: A non-deterministic merge kernel written in Cal.

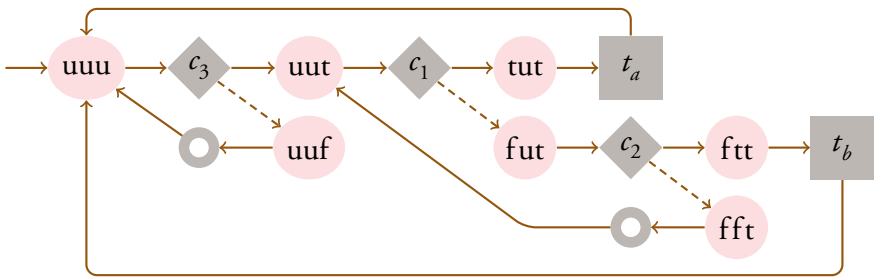


Figure F.2: A graphical representation of an actor machine controller that implements the merge actor in Listing F.1.

producing that input. To discard knowledge about such conditions, a WAIT(s') instruction makes the actor machine proceed to a state s' where those conditions are unknown. The WAIT instruction is therefore also a signal that this actor machine cannot make progress unless an external event occurs.

Listing F.1 shows a small Merge kernel written in the actor language Cal. It has two actions, labeled a and b, that describe two transitions t_a and t_b, respectively. Action a reads one token v from port A and writes its value to the output port C. The other action does the same thing, but with port B.

Figure F.2 shows a graphical representation of an actor machine that implements the Merge actor in Listing F.1. The ellipses represent states, the diamonds, rectangles and rings represent TEST, EXEC and WAIT instructions, respectively. The dashed edges point to the target state of the TEST when the condition is false. The two actions of the Merge actor have three conditions in total: condition c_1 is the availability of a token on port A, condition c_2 is the availability of a token on port B, and condition c_3 is the availability of a space for a token on port C. The knowledge about these three conditions is represented by a triple $\langle k_1, k_2, k_3 \rangle$ of values from the set $\{\mathrm{t}, \mathrm{f}, \mathrm{u}\}$, where $k_i = \mathrm{t}$ and $k_i = \mathrm{f}$, means the condition c_i is known to be true and false, respectively, and $k_i = \mathrm{u}$ means that condition c_i is unknown. For instance, in state $\langle \mathrm{u}, \mathrm{u}, \mathrm{t} \rangle$, it is

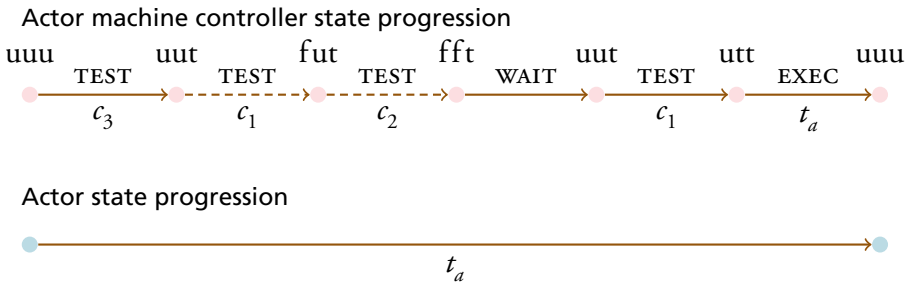


Figure F.3: State progression of an actor machine controller and the actor it implements.

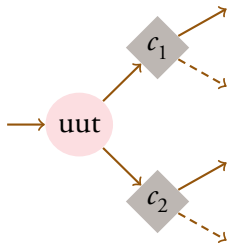


Figure F.4: A fragment of a multi-instruction actor machine with a state where condition c_3 is known to be true, but condition c_1 and c_2 are unknown.

not known whether there are any tokens on the input ports, but it is known to have space for output.

It is important to distinguish the state of the actor machine controller from the state of the actor. The controller state changes with every actor machine instruction, but the actor state is only changed by transitions, which are fired by the EXEC instruction. Figure F.3 illustrates the state progression of the actor machine controller and the actor state for a particular execution of the Merge actor and its implementation in Figure F.2. When the execution starts, there is no token on any of the input ports but there is space for an output token. From there, condition c_3 is tested, followed by c_1 and c_2 and a WAIT, because no transition could be selected. So far, the actor state is unchanged and only the controller has made progress. At some point after c_1 was tested, a token arrived, and now c_1 is tested again, this time with a successful result. The last step of the controller progression is the firing of transition t_a, which also affects the actor state, shown in the progression in Figure F.3.

The example in Figure F.2 is a single-instruction actor machine, which means it has at most one instruction in each state. Actor machines with more than one instruction per state are called multi-instruction actor machines. In state $\langle u, u, t \rangle$, for instance, any of c_1 and c_2 could have been tested, and the state could have TEST instructions for both conditions, as depicted in Figure F.4.

To create an efficient implementation of a stream program, the translation of kernels to single-instruction actor machines can be done in two steps, by first creating multi-instruction actor machines and then *reducing* the controllers to single-instruction actor machines. The multi-instruction actor machines describe many possible single-instruction actor machines, and the purpose of reduction is to find efficient single-instruction actor machines among those. By separating the translation to actor machine and the instruction selection (through reduction), different source languages can use the same instruction selection strategies by using the same reducers.

F.3 THE TŸCHO COMPILATION FRAMEWORK

F.3.1 Overview

Tÿcho is an open source framework for building compilers for stream programs. The framework is built upon an intermediate representation for kernels based on actor machines, together with a simple network description of how the kernels are connected. The Tÿcho code base includes an example compiler, the Tÿcho compiler (tychoc), that reads Cal programs and Kahn process networks and generates optimized C code from the intermediate representation.

Compilers are built as a sequence of compilation phases, starting with a phase that reads the source code of the kernels and the network descriptions of how the kernels are connected. After all kernels and network code is parsed follows phases that perform name analysis, type analysis and other analyses. If the program is valid, the following phases evaluate the network descriptions to build a concrete network of kernels, and kernels are translated to actor machines. At this stage of the compilation, the language specific parts have been replaced by a network of actor machines. Optimization phases are performed on this representation, before finally handing over the program to the backend phase.

To build a new compiler for a different backend or a different kernel language, some phases need to be implemented while others from the framework can be reused. Different compilers will have different sequences of phases with

different sequences of optimizations, based on what is good for the target platform and depending on what is required by the source languages.

F.3.2 Actor Machine Intermediate Representation

The intermediate representation of the stream program kernels is based on actor machines. Just like in the formal model, the actor machines of the intermediate representation contains input and output ports for communication, conditions, transitions and state.

Conditions and Transitions. The transitions are described in a procedural language with statements for consuming input and producing output through the ports. An important aspect of dataflow with firing is that the internal state of an actor does not change *between* firings, and since conditions are tested between the firings, they cannot change the state. Conditions are therefore written in a purely functional language that cannot change state variables or consume input tokens or produce output tokens. The order in which conditions are evaluated cannot affect the result of the evaluation.

The procedural and the functional languages are tightly integrated, with the functional language being the expression part of the procedural language. Functions and procedures are first class values, and their definitions are expressions in the functional language.

Controller. The controller is represented as a bipartite graph where the vertices are states or instructions. It is constructed lazily, to allow large multi-instruction controllers to be described without requiring the complete graph to fit in memory.

A reduced controller is a controller where some instructions are removed, typically making some other states and instructions unreachable and implicitly removed. Transformed controllers can be constructed as a layer on top of an existing controller, referring to states and instructions of the controller beneath. Reductions, for example, can be implemented as such.

Variable Scopes. The state of an actor machine is represented by variables, and the variables are grouped into scopes. An initialization expression of a variable in a scope may refer to any variable in that scope, similar to `let rec` in ML. Recursive functions and procedures, and even mutually recursive ones,

can therefore simply refer to each other as long as they are declared in the same scope.

Scopes can have two kinds of lifespans—they are either *persistent* and live as long as the actor machine executes, or *transient* and become invalid right after each firing. Persistent scopes are initialized before the actor machine starts executing. A transient scope is initialized before its variables are used, if it is not already initialized. For example, if two conditions refer to variables in the same scope, the scope does not need to be reinitialized when evaluating the second condition, given that no transition has been fired in between. More generally, if it can be determined statically that the variables of a transient scope will have the same values if it is initialized again, it does not need to be reinitialized.

The decoupling of the variable scopes from the conditions and transitions enables some interesting optimizations. Common subexpression elimination between conditions can be implemented by moving the common subexpression to a variable in a transient scope that both conditions refer to. The actor language Cal supports sharing variables between conditions and transitions, which naturally translates to a transient scopes that both the conditions and the transition refer to.

F.3.3 Frontend: Cal and RVC-CAL

Cal [12] is a language for dataflow actors with firing. Listing F.1 on page 118 shows an example actor written in Cal. Tÿcho includes frontends for two versions of Cal, one that is based on the Cal Language Report [12] and one that is based on RVC-CAL [48]. Since RVC-CAL is a subset of Cal, the RVC-CAL actors are represented internally by the abstract syntax tree for Cal actors.

A Cal actor has internal state variables and ports through which it communicates. The computation is described by a set of *actions*, where each action is a transition combined with its associated conditions. An action may also declare variables in an *action scope* that is visible both to the transition and the conditions. An action scope needs to be initialized before testing a condition that requires it, or before firing the transition. After an action has fired, all action scopes are invalidated and needs to be reinitialized the next time they are required.

The translator from Cal actor to actor machine translates each action to a transition, a set of conditions, and a transient scope with the variables of the action scope. Then, a multi-instruction controller is created that implements the action selection according to the semantics of Cal. There are a few other constructs in Cal actors that affects the action selection (action priorities, action

```
process SumN () uint N, int X ⟹ int Sum :
  int sum;
  uint n;
  int x;
  repeat
    sum := 0;
    N ⟶ n;      // reads a value from port N to variable n
    while n > 0 do
      X ⟶ x;    // reads a value from port X to variable x
      sum := sum + x;
      n := n − 1;
    end
    Sum ⟵ sum;  // writes the value sum to port Sum
  end
end
```

Listing F.2: An example process that computes the sum of a given number of values. The read and write statements are marked with comments.

schedules and initialization actions), all of which are taken into account in the creation of the controller.

F.3.4 Frontend: Kahn Processes

Tÿcho also has a frontend for a Kahn process language that, syntactically, is influenced by Cal, but is very different in the way the computation is structured. Listing F.2 shows an example kernel in this process language. The process language is designed to be similar to Cal, and borrows the expression language and types directly from Cal. The statements are extended to include a read operation and a write operation, which do not collide with any of the current syntax of Cal. The process declaration is also similar to the actor declaration of Cal.

Another requirement on the language design is that the processes should be implementable without requiring to maintain a stack between the read and write operations. As a consequence of this requirement, read and write operations are only allowed in the process body and not in procedures, because procedures can be recursive.

In [81], a translation from Kahn processes to dataflow actors is presented, which is the basis for the Kahn process frontend in Tÿcho. The idea is to iden-

tify segments in the code that can be expressed as transitions, i.e. producing and consuming a fixed number of tokens. The translator in [81] starts by translating the process to a flowchart and then groups the nodes of the flowchart to sequences of statements with the goal of finding long sequences that can be grouped to a transition without changing the behavior of the kernel. The reason for finding long sequences of statements rather than short is that it reduces the overhead, both in the rest of the compilation and at runtime. Instead of creating a flowchart, the translator in Tÿcho starts with a control flow graph where the basic blocks are split such that every basic block becomes a valid transition and with the same requirement that the behavior of the process is retained. Variables of a Kahn process are translated to variables in a persistent scope of the actor machine intermediate representation.

F.3.5 Backend: C

Tÿcho has a backend that generates C programs. The generated code is then compiled using a C compiler for the target platform. The state variables of a kernel, together with the state of the actor machine controller and references to the buffers that it communicates through, is represented by a struct. Values of this struct are called state objects. The conditions and transitions are represented by functions that take a reference to a state object as parameter. The controller is also implemented by a function that takes a reference to a state object as parameter, and the controller function returns true if it has fired a transition, and false otherwise. Listing F.3 illustrates the code that is generated for the controller in Figure F.2.

Functions and procedures of the intermediate representation may refer to variables declared in enclosing scopes. The C backend implements functions and procedures as fat pointers—objects with a pointer to a function and a pointer to an object describing the envorinment, where the environment object contains pointers to the variables of the enclosing scopes that the function or procedure uses.

F.4 TRANSITION SELECTION

F.4.1 Background

Dataflow actors with firing are executed as a sequence of transition firings, and before each firing the actor needs to select which transition to fire. The ex-

```
_Bool Merge_run(Merge_state *self) {
    _Bool progress = false;

    switch (self->controller_state) {
    case Merge_uuu: goto uuu;
    case Merge_uut: goto uut;
    }

    uuu: // Test(c3, uut, uuf)
    if (Merge_c3(self)) goto uut; else goto uuf;

    uut: // Test(c1, tut, fut)
    if (Merge_c1(self)) goto tut; else goto fut;

    tut: // Exec(ta, uuu)
    progress = true; Merge_ta(self); goto uuu;

    uuf: // Wait(uuu)
    self->controller_state = Merge_uuu; return progress;

    fut: // Test(c2, ftt, fft)
    if (Merge_c2(self)) goto ftt; else goto fft;

    ftt: // Exec(tb, uuu)
    progress = true; Merge_tb(self); goto uuu;

    fft: // Wait(uut)
    self->controller_state = Merge_uut; return progress;
}
```

Listing F.3: Simple implementation of an actor machine controller in C that executes until it cannot make any progress.

ecution can therefore be seen as a process that alternates between selecting a transition by testing conditions and executing the selected transition.

The actor machine controller describes this process where TEST instructions test conditions and EXEC instructions execute transitions. It also describes the situations when no transition can be selected using the WAIT instruction. One important aspect of the actor machine is the ability to encode knowledge in the states of the controller, which enables an actor machine to avoid testing conditions that it already has tested.

Given the knowledge that is encoded in a controller state, there are in general several things that could be done. If two conditions, c_1 and c_2, are unknown in controller state s, testing either c_1 or c_2 are candidates for a next step of execution. However, if transition t can be executed in the same state, s, it is probably better to execute t immediately, because the selection process is all about selecting a transition. A multi-instruction actor machine controller may have all three instructions in state s but only one of them can be executed each time the controller is in that state.

A multi-instruction controller can be seen as a family of single-instruction controllers, i.e. all controllers that can be generated by selecting one of the instructions in each state. The process of selecting a non-empty subset of the instructions in each state is called *reduction*. If the reduction yields a single-instruction controller, it is called a *full reduction*. Different reductions yield different performance when executing the actor machine.

The space of possible single-instruction controllers that a multi-instruction controller represents can be explored by randomly choosing one instruction in each state. By repeating this process to get different implementations of the same program, the implementations can then be benchmarked to find which one is the fastest. This process can, however, be very time consuming, and needs to be repeated if the program changes. An alternative approach is to use reduction heuristics that select instructions.

F.4.2 Controller Reducers

To support the claim that Tÿcho provides efficient transition selection, we present a few reduction heuristics that optimize the transition selection. Some of the heuristics have been presented in [82]. We categorize the reducers using two metrics: whether the reducer uses profiling information and whether the reducer looks at more than one state when making a decision. The reducers that only consider one state (or possibly a fixed number of states) to make a reduction decision are called *local reducers*, and the ones that consider arbitrarily many

states are called *regional reducers*. Reducers that look at the whole controller at the same time are called *global reducers*. This paper does not include any global reducers, but the regional reducers might consider all states depending on the actual controller they reduce.

Reducer: Random. The random reducer selects an instruction in each controller state at random and yields a full reduction of the controller. The pseudorandom number generator can optionally be given a seed, to be able to make reproducible builds of a program, otherwise it will start with a seed based on the current time. This is a local reducer that does not use any profiling information.

Reducer: Instruction Priority. The instruction priority reducer prioritizes instructions in the following order: EXEC then WAIT and finally TEST. If a state contains an EXEC instruction, then all EXEC instructions are selected. Otherwise, if a state contains a WAIT instruction, then all WAIT instruction are selected. Finally, if there are only TEST instructions, then all instructions are selected.

The reason for prioritizing EXEC instructions is that the process is all about finding a transition to execute. The reason for prioritizing WAIT instructions over TEST instructions is not as obvious. The WAIT instructions are only allowed in states where no transition can be executed until one of the known conditions are retested, and to retest a condition, the knowledge about it must first be discarded by a WAIT instruction. This means that from a state with a WAIT instruction, there exist no sequence of TEST instructions that lead to an EXEC instruction without first going through a WAIT. By taking the WAIT instruction as early as possible, the required conditions can be retested to reach an EXEC. It is also possible that results from TEST instructions might be discarded in the coming WAIT anyway.

This is a local reducer that does not use any profiling information, and because it selects all instructions of a kind, it does not, in general, yield a full reduction.

Reducer: Most Differentiating Test. A TEST is differentiating if the sets of transitions that are reachable from the target states are different. The more different the sets are, the more differentiating the TEST is.

Let $r : S \to \mathscr{P}(T)$ be a function from a state s to the set of transitions of EXEC instructions that are reachable from s going through only TEST instruc-

tions. Given a TEST(c, s_t, s_f), the size of the symmetric set difference[1] between $r(s_t)$ and $r(s_f)$ is the metric of how differentiating the TEST is. In a state with TEST instructions, this reducer selects all tests with the highest value of the differentiation metric.

The rationale behind this reducer is that the most differentiating TEST instructions brings the decision process closer to an EXEC or a WAIT instruction, while the least differentiating TEST instructions does not matter for most transitions and should be postponed to states where it matters more.

This reducer is a regional reducer, because it traverses the controller to see which transitions are reachable. It does not, in general, yield a fully reduced controller, because several TEST instructions could get the same value of the differentiation metric. The reducer does not use any profiling information.

Reducer: Most Probable Condition. This reducer is introduced in [82] and is based on a platform-independent profiling metric. The program is first instrumented to record the truth values of every tested condition in the controller, and then executed with some input. The empirical probabilities of the conditions are then calculated and used as input for this reducer which selects the TEST instructions with the conditions that have the highest probability to be true.

Let $\hat{P}(c)$ be the empirical probability that c is true, and $\delta = 0.1$ be a probability interval. Given the set of conditions C_s of all TEST instructions in a state s, all TEST instructions with a condition $c \in C_s$ such that $\hat{P}(c) \geq \max_{c' \in C_s} \hat{P}(c') - \delta$ are selected.

The idea behind this reducer is that testing the most probable conditions will quickly lead to a transition that is likely to be selected. It might, however, be the case that an unlikely transition requires many conditions that are very likely, and few conditions that are unlikely.

This reducer is a local reducer that uses profiling information and does not, in general, yield a fully reduced controller.

Reducer: Most Probable Transition. This reducer is also introduced in [82] and is also based on a platform-independent profiling metric: the transition probabilities. The program is first instrumented to record how many times each transition is executed, and then executed with some input. The relative frequencies of the transitions then guide the reducer.

[1] The symmetric set difference between A and B is $(A \cup B) \setminus (A \cap B)$.

The reducer is a regional reducer that traverses the controller to see which transitions are reachable, i.e. the transitions for which there is an EXEC instruction that is reachable through a sequence of TEST instructions. Given a state, this reducer finds which transitions are reachable and selects the most probable transition as its target. Then it selects all instructions that are on a path no longer than the shortest path to an EXEC instruction with the target transition.

The rationale behind this reducer is that it will with as few tests as possible reach the most probable transition.

The reducer is a regional reducer that uses profiling information. Because there can be many paths of equal length that is the shortest path to the target transition, this reducer does not, in general, yield a fully reduced controller.

F.4.3 Evaluation

We evaluate how the reducers in the previous section affect the performance of a stream program. The evaluation is similar to [82], but in addition to the two profiling based reducers, we also evaluate the reducer that selects differentiating tests.

None of the three reducers that we evaluate yield fully reduced controllers. To evaluate the effect of these reducers, we sample the space of possible single-instruction controllers that these multi-instruction controllers describe, using the random reducer. We also sample the space of single-instruction controllers without first applying any of the three reducers. The instruction priority reduction is applied before all other reducers. In summary, there are four chains of reducers that are evaluated:

most-differentiating-tests *instruction priority*, and then *most differentiating tests*, and then *random*.

most-probable-transitions *instruction priority*, and then *most probable transitions*, and then *random*.

most-probable-conditions *instruction priority*, and then *most probable conditions*, and then *random*.

baseline *instruction priority*, and then *random*.

We use as benchmark the reference implementation of the MPEG-4 Part 2 SP decoder from the RVC framework, consisting of 39 Cal actors of various degrees of complexity. We generate 30 variants of each chain of reducers with different random seeds and generate C programs from each of them. The C

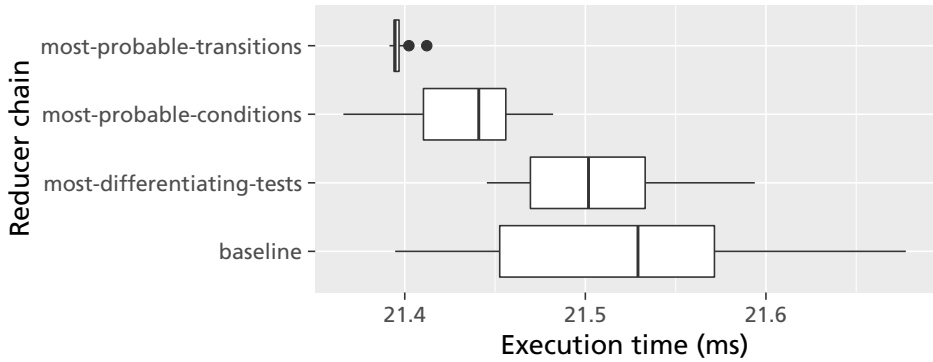


Figure F.5: Box plot of execution time of the benchmark grouped by which reducers are used. The multi-instruction actor machines that are the result of the reductions are sampled using a random reducer.

programs are compiled using GCC 4.8.2 for a 32-bit PowerPC system and are simulated using the cycle-accurate ASIM Power Architecture simulator [61].

The execution times of the benchmark, compiled with the four chains of reducers, are presented in Figure F.5 as a box plot. The reducer that aims for the most probable transitions resulted in the smallest median and quartiles and the smallest variance. The reducer that selects the most probable conditions resulted in the smallest minimum. The reducer that selects the most differentiating test resulted in a slightly lower median than the baseline, but did not yield as fast programs as the baseline in the best case. The results show that Tÿcho can optimize the transition selection by controller reduction.

F.5 KERNEL FUSION

F.5.1 Background

When more than one kernel is executed on the same processor, they need to be scheduled. Programs with thread based concurrency are often implemented using preemptive scheduling, but stream programs, and especially dataflow with firing, can often be scheduled more efficiently. Some stream programs can be executed with a static schedule that is computed at compiletime, e.g. synchronous dataflow [1]. Other stream programs, for example Boolean-controlled dataflow [9] can be scheduled statically to some extent, and only

requires some scheduling decisions to be done at runtime. Stream programs that cannot be scheduled using these efficient techniques could still contain subprograms that can.

One way of implementing a schedule for a set of kernels is by fusing the kernels. The fused kernel performs the computation of the original kernels in the order that the schedule describes. Actor machine fusion (that is sometimes called actor machine composition) implements kernel fusion for actor machines and is one of the motivations for the machine model [4]. Cedersjö and Janneck presented an algorithm for actor machine fusion in [64], but this algorithm was never shown to be scalable.

In this section we review the algorithm in [64] and present a new, improved algorithm. We compare the new actor machine fusion algorithm to a Cal actor fuser by Boutellier et al. [65].

F.5.2 Actor Machine Fusion

As discussed earlier, an actor machine consists of one controller and sets of communication ports, transitions, conditions and scopes. A fused actor machine is still an actor machine and consists of one controller and the disjoint unions of, respectively, the communication ports, transitions, conditions and scopes of the original actor machines. The controller is created by merging the controllers of the original actor machines and injecting additional knowledge about the state of the system to avoid TEST instructions.

Controller State. A controller that is fused from n actor machines keeps track of the controller states of its original actor machines in a tuple $\mathbf{s} \in S_1 \times \cdots \times S_n$. Each element $s_i \in S_i$ in the tuple $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$ is the current controller state of original actor machine i.

An important aspect of actor machine fusion is the ability to remove TEST instructions that are not needed. Similar to how actor machines translated from Cal keeps track of result of tested conditions by encoding the knowledge in its states, a fused actor machine can keep track of how its original actor machines communicate. The states of a fused controller keeps track not only of the states of its original actors, $S_1 \times \cdots \times S_n$, but also of additional knowledge K about the program state, such as the number of tokens on a specific buffer. The complete state of a fused controller is denoted $\langle \mathbf{s}, k \rangle \in S_1 \times \cdots \times S_n \times K$, where K is an abstraction of the program state.

Program State Knowledge. An actor machine condition evaluator $e : \Sigma \times C \to \{\mathrm{t}, \mathrm{f}\}$ is a function that evaluates an actor machine condition for a given program state. We use Σ to denote the set of program states and C for the set of conditions. A transition executor is described by a partial function $u : \Sigma \times T \nrightarrow \Sigma$, where T is the set of transitions. The transition executor is defined for all legal transitions in the given state—u is for example undefined for transitions that read tokens in states where the tokens are not present.

The additional knowledge that a fused controller encodes is an abstraction $\mathscr{K}$ of the program state Σ. The abstraction $\mathscr{K} : \Sigma \to K$ maps program state to a simpler representation K that captures some aspects of the program state, e.g. the number of tokens on a particular buffer.

At compiletime, when the actor machine fusion is performed, the full program state Σ is not known, which makes it impossible to evaluate $\mathscr{K}$. Instead, the knowledge abstraction is represented by a 4-tuple $(K, k_0, \hat{e}, \hat{u})$, where K is the set of abstract knowledge, $k_0 = \mathscr{K}(\sigma_0)$ is the knowledge about the initial state, σ_0, of the program. The functions $\hat{e} : K \times C \nrightarrow \{\mathrm{t}, \mathrm{f}\}$ and $\hat{u} : K \times T \nrightarrow K$ are the duals of e and u that operate on the abstract state K instead of the program state Σ. The function $\hat{e}$ is consistent with e iff $\hat{e}(\mathscr{K}(\sigma), c) = e(\sigma, c)$ for all $\sigma \in \Sigma$ and $c \in C$ for which $\hat{e}(\mathscr{K}(\sigma), c)$ is defined. Similarly, $\hat{u}$ is consistent with u iff $\mathscr{K}(u(\sigma, t)) = \hat{u}(\mathscr{K}(\sigma), t)$ for all $\sigma \in \Sigma$ and $t \in T$ for which $u(\sigma, t)$ is defined.

In this article, we use an abstraction that captures the number of tokens on buffers that are internal to the fused actor machine. In this abstraction, $K = B_1 \times \cdots \times B_r$, where B_i is the set of integers from 0 to the buffer size of buffer i, the initial knowledge is $k_0 = \langle 0, \ldots, 0 \rangle$. The evaluation function $\hat{e}$ is defined for all conditions that test for tokens or for space on the internal buffers. The transition function $\hat{u}$ is defined for all transitions that do not consume tokens that are not present or produce tokens that do not fit on the internal buffers, and it returns a new element of K where the number of tokens are updated according to what the transition does.

Controller Fusion Algorithm. For each controller state $\langle \mathbf{s}, k \rangle \in S_1 \times \cdots \times S_n \times K$ in the fused controller, the fusion algorithm generates instructions by selecting which instructions of the original actor machines should be present in the fused state. The target states of the instructions is then adapted to the state space of the fused controller. The initial state of the fused controller $\langle \mathbf{s}_0, k_0 \rangle$ consists of the tuple of initial states of the original controllers together with the initial

knowledge state. The fused controller can be created by generating instructions for the initial state and all states that are targets of the generated instructions.

Given a state $\langle \mathbf{s}, k \rangle$, the fusion algorithm calls a function SELECT INSTRUCTIONS $(\mathbf{s}(i), k)$ for each original actor machine i to get the instructions of each actor machine controller. A simple implementation of SELECT INSTRUCTIONS returns all instructions from the controller state of that actor machine. However, to remove TEST instructions whose conditions are known by k, SELECT INSTRUCTIONS needs to consider the additional knowledge represented by k and select instructions accordingly. This article presents two implementations of SELECT INSTRUCTIONS that use the knowledge k.

If the set of instructions returned by the calls to SELECT INSTRUCTIONS contains an EXEC instruction, then there is no need to test any more conditions, therefore only EXEC instructions are selected in that case. Similarly, if there are no EXEC instructions, but there are TEST instructions, they are prioritized over WAIT instructions, because WAIT instructions indicate that the actor machine cannot execute a transition and a TEST could potentially lead to an EXEC.

If an $\text{EXEC}(t, s')$ instruction is selected from the controller of actor machine i to be included in the fused controller state $\langle \mathbf{s}, k \rangle$, then the instruction is adapted with a new target state $\langle \mathbf{s}[i \mapsto s'], \hat{u}(k, t) \rangle$, i.e. element i of the state tuple $\mathbf{s}$ is updated to s', and the knowledge is updated using $\hat{u}$. Similarly, when a $\text{TEST}(c, s', s'')$ instruction is selected from actor machine i for the fused controller state $\langle \mathbf{s}, k \rangle$, it is transformed to $\text{TEST}(c, \langle \mathbf{s}[i \mapsto s'], k \rangle, \langle \mathbf{s}[i \mapsto s''], k \rangle)$. When WAIT instructions are selected, they are merged to a single WAIT instruction. Given a controller state $\langle \mathbf{s}, k \rangle$, for each i from 1 to n, let w_i be either the target state of the WAIT instruction that is selected from controller $\mathbf{s}(i)$ if such instruction is selected, or $\mathbf{s}(i)$ otherwise. The merged instructions is $\text{WAIT}(\langle \langle w_1, \ldots, w_n \rangle, k \rangle)$.

Instruction Selector 1. As mentioned in the previous section, a simple implementation of SELECT INSTRUCTION selects all instructions from the given actor machine state. This selector does not eliminate any TEST transitions and the state space of the resulting controller can therefore be very large.

In this section we present an instruction selector that is based on the algorithm in [64], but is generalized to handle additional knowledge of any knowledge abstraction and not just the number of tokens on the internal buffers. The idea behind Algorithm F.1 is to check if the conditions of the TEST instructions are known by the additional knowledge, and in that case perform the selection from the target state instead.

The algorithm in [64] was successfully applied to small examples of Cal programs with performance improvements as result, but was not tested on larger examples.

Algorithm F.1 Instruction selector 1.

```
function SELECT INSTRUCTIONS(state s, knowledge k)
    X ← all instructions in s
    return ⋃_{x∈X} APPLY KNOWLEDGE(x, k)

function APPLY KNOWLEDGE(instruction x, knowledge k)
    match x with
        case TEST(c, s', s'') do
            match ê(k, c) with
                case true do
                    return SELECT INSTRUCTIONS(s', k)
                case false do
                    return SELECT INSTRUCTIONS(s'', k)
                case undefined do
                    return {x}
        case EXEC(_, _) or WAIT(_) do
            return {x}
```

Instruction Selector 2. When scheduling two actor machines A and B, if it is known that A cannot reach an EXEC before B has performed one of its EXEC instructions, then there is no need to perform any TEST instruction of A before the EXEC instruction in B has been executed. For example, if B produces a token that A requires, then there is no need to check the other conditions of A before B has produced that token.

This observation is used in the second instruction selector, Algorithm F.2. Before selecting instructions, liveness analysis is performed on the actor machine controller, checking if there exist a sequence of TEST instructions that could lead to an EXEC instruction, taking the additional knowledge in the current state into account. If there is a possibility to reach an EXEC instruction, then instructions are selected as in the previous algorithm, otherwise, only WAIT instructions are selected. The WAIT instructions are needed for the fused state

where all original actor machines have stalled to generate a combined WAIT instruction.

Algorithm F.2 Instruction selector 2.

```
function SELECT INSTRUCTIONS(state s, knowledge k)
    X ← all instructions in s
    X ← ⋃_{x∈X} APPLY KNOWLEDGE(x, k)
    if STALLED(s, k) then
        return the WAIT instructions in X
    else
        return X

function APPLY KNOWLEDGE(instruction x, knowledge k)
    ▷ see Algorithm F.1

function STALLED(state s, knowledge k)
    X ← all instructions in s
    return ⋀_{x∈X} STALLED AT INSTRUCTION(x, k)

function STALLED AT INSTRUCTION(instruction x, knowledge k)
    match x with
        case EXEC(_, _) do
            return false
        case WAIT(_) do
            return true
        case TEST(c, t, f) do
            match ê(k, c) with
                case true do
                    return STALLED(t, k)
                case false do
                    return STALLED(f, k)
                case undefined do
                    return STALLED(t, k) ∧ STALLED(f, k)
```

F.5.3 Evaluation

We evaluate the efficiency of the actor machine fusion algorithm by measuring the execution time of stream programs with and without actor machine fusion. To compare to prior art, we also reproduce the experiments of Boutellier et al. [65].

We generate two C programs from each of the four RVC-CAL programs in [65] using the Tÿcho compiler, one with actor machine fusion and one without. These eight programs, together with the corresponding eight C programs of [65], that were generated using Orcc, are then compiled to three different target architectures. All programs are executed ten times on each platform, and we present the mean and the standard deviation of the execution times.

Boutellier et al. [65], uses the number of clock cycles per processed sample as a platform independent performance metric. The numbers are achieved by subtracting an estimate of the time spent on I/O operations from the total execution time and dividing it by the number of processed samples. The estimated time spent on I/O is computed by timing a different program that only performs the I/O operations. In this paper, we instead compare the whole program performance *including* the time spent on I/O operations. The main reason is that, on our platforms, the execution time of one program doing a series of I/O operations is not an accurate estimate of the time spent doing the same I/O operations in a different program.

The result is presented in Table F.1. The applications Adaptive LMS and Digital predistortion became much faster after kernel fusion on all three platforms and both stream program compilers. Both applications were very fine-grained with small and simple kernels, and were completely scheduled in both kernel fusion algorithms. For the ZigBee transmitter baseband, the kernel fusion did not make any measurable difference, which is not surprising because the application only contains four kernels that are much more complex than the ones in the previous examples. The MPEG-4 decoder became slightly faster after kernel fusion on all architectures except on the PowerPC system with the Tÿcho compiler.

We believe the main reasons why Orcc, in most of these examples, produces faster code than Tÿcho, are the difference in how closures are implemented and how actor state is handled. Orcc supports the kinds of closures that are permitted in RVC-CAL, which can be implemented without pointers, while Tÿcho handles closures more generally. Tÿcho includes some optimizations for closures, but does not detect all cases that are permitted in RVC-CAL. The actor

Table F.1: Number of clock cycles per input sample.

	ARM		PowerPC		Intel x86	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
Application: Adaptive LMS (39 kernels)						
Tÿcho, baseline	4522	41.4	3029	25.1	1548	33.4
Tÿcho, with fusion	172	1.9	181	0.2	107	4.3
Orcc, baseline	13773	46.1	11679	39.8	3359	97.2
Orcc, with fusion	108	0.2	80	0.1	70	14.5
Application: Digital predistortion (80 kernels)						
Tÿcho, baseline	6765	52.8	3451	13.2	1974	40.1
Tÿcho, with fusion	1569	55.9	1544	1.2	823	18.7
Orcc, baseline	3050	52.9	1358	1.1	1245	32.7
Orcc, with fusion	383	0.8	471	1.4	594	19.4
Application: Zigbee transmitter baseband (4 kernels)						
Tÿcho, baseline	33216	1266.2	10900	530.0	36366	201.3
Tÿcho, with fusion	30127	715.5	10997	604.5	36276	219.1
Orcc, baseline	29814	514.8	9317	293.5	35541	160.5
Orcc, with fusion	30159	1209.5	9319	204.8	35999	337.6
Application: MPEG-4 part 2 SP decoder (34 kernels)						
Tÿcho, baseline	90115	921.7	53392	106.5	31810	229.0
Tÿcho, with fusion	76284	1096.9	56362	289.8	29127	175.7
Orcc, baseline	51306	1370.2	29131	128.9	16886	212.1
Orcc, with fusion	34184	138.4	25120	81.6	14492	181.6

state is also handled differently; Orcc translates actor variables to static variables of the C program, while Tÿcho puts all variables in a state object. The reason for using a state object is for code reuse and to makes it easier to move kernels in memory or between computers.

F.6 ACTOR MACHINE SCOPE OPTIMIZATIONS

F.6.1 Background

The variable scopes of the actor machine based kernel representation in Tÿcho offers some optimization opportunities. In this section, we discuss two opti-

mizations; one that is applicable when several conditions or transitions use the same variable scopes, and one that is applicable when only one condition or transition use a particular variable scope.

F.6.2 Scope Initialization

Problem. Actor machine scopes have, as discussed in Section F.3.2, two different lifespans—persistent, that live as long as the actor, and transient, that live until and including the next transition. A scope needs to be initialized before being used by a condition, a transition or by the initializer of another scope. Transient scopes, therefore needs to be initialized repeatedly throughout the execution of the actor machine.

Since the lifespan of transient scopes end after each transition, and only the transitions can change the values of their variables, transient scopes can be initialize before every time it is used. All conditions between two transitions will see the same values in the transient scopes, irrespective of how many times it is reinitialized.

It is, however, unnecessary to reinitialize a scope if its variables will get the same values. For example, if first a condition uses a transient scope, and then a transition uses the same transient scope, the scope does not need to be initialized before the second use (unless another transition has been performed in between).

Some transient scopes are unaffected by some transitions. If a transient scope will always be initialized to the same values before and after a transition has executed, then it is unaffected by that transition. Even though, the lifespan of such scope has ended, it will be initialized to the same values on its next use. This is a situation that occurs more often when kernels are fused, because the transitions cannot update the variables that originate from other kernels.

The problem we are addressing here is the unnecessary reinitialization of transient scopes that takes time that is better spent on actual computation.

Solution. Our solution is to perform static analysis on the actor machine controller, the transitions and the scopes to reduce the amount of reinitialization.

The first thing that is analyzed is how controller instructions use the actor machine scopes. Two functions are defined; $\mathbf{req} : I \to \mathscr{P}(Z)$ which gives the transitive dependencies on scopes of an instruction, and $\mathbf{kill} : I \to \mathscr{P}(Z)$ which is a mapping from instructions to the set of scopes that are no longer alive after the instructions has executed.

The function **req** is the transitive closure of the scope dependencies, meaning for example if $i \in I$ uses a variable in scope $z_1 \in Z$ and an initializer in z_1 in turn requires a variable in $z_2 \in Z$, then $\{z_1, z_2\} \subseteq \mathbf{req}(i)$. Regarding **kill**, if a transition that is executed by instruction $i \in I$ consumes a token from a port q, and $Z_q \subseteq Z$ is the set of scopes that are initialized with a token from port q, then $Z_q \subseteq \mathbf{kill}(i)$, because the scopes in Z_q might get initialized to different values after the transition.

To describe how states and instructions relate, we define a relation $\mathbf{tgt} \subseteq I \times S$ that is the relation between instructions and their target states, and a function $\mathbf{src} : I \to S$ that maps instructions to their source state.

We define two functions $\mathbf{alive}_I : I \to \mathscr{P}(Z)$ and $\mathbf{alive}_S : S \to \mathscr{P}(Z)$ in terms of each other. The first, $\mathbf{alive}_I$, maps instructions to the set of scopes that are alive after that instruction. The second, $\mathbf{alive}_S$, maps states to the set of scopes that are alive in that state. Let Z_p be the set of persistent scopes,

$$\mathbf{alive}_S(s) = Z_p \cup \bigcap_{(i,s') \in \mathbf{tgt}, s = s'} \mathbf{alive}_I(i), \text{ and}$$

$$\mathbf{alive}_I(i) = \big(\mathbf{alive}_S(\mathbf{src}(i)) \cup \mathbf{req}(\mathbf{i})\big) \setminus \mathbf{kill}(i).$$

The set of scopes that needs to be initialized before executing instruction i is its required scopes, except the persistent scopes and the ones that are already alive: $(\mathbf{req}(i) \setminus Z_p) \setminus \mathbf{alive}_I(i)$.

F.6.3 Scope Lifting

Problem. Input patterns of an action in Cal declare variables for the input tokens that are visible both to its guards and its action body. An action may also declare other variables that are visible in both places. When a Cal actor is translated to an actor machine in Tÿcho, these variables are put into an actor machine scope with a transient lifespan (see Section F.3.2). Transient actor machine scopes, just like action scopes, end their lifespan after each transition or action firing.

We have found that declaring local variables inside a condition or a transition often yields faster programs than referring to variables that are in actor machine scopes. One reason that makes it faster is that the C compilers can do more optimizations on local variables than variables in the state object, because it has more information about what happens to the variable if it is local to the function.

In this section, we address the problem of lifting variable declarations of transient scopes to conditions and transitions to increase program performance.

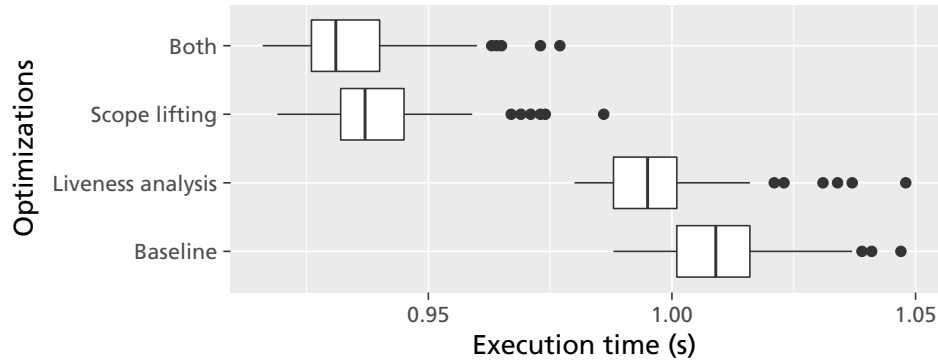


Figure F.6: Execution time of MPEG-4 Part 2 SP decoder, compiled using different optimizations

Solution. Our solution is to lift all transient scopes that are used by exactly one condition or one transition. It could be beneficial to lift other scopes as well, for example if the optimizations that are enabled by lifting a variable reduces the execution time more than the repeated initialization of the variable increases the execution time, but that is beyond the scope of this paper.

F.6.4 Evaluation

We have implemented scope liveness analysis and scope lifting in the Tÿcho compiler, and measured the execution time of the reference implementation of the MPEG-4 part 2 SP decoder from the RVC framework with and without the two optimizations. The programs were tested on an Intel Core i7, running macOS, which introduces some randomness from other processes running on the system. The generated C programs were compiled using Clang with optimizaiton flag -O3. Figure F.6 shows that scope lifting was more effective than liveness analysis on this example, and that the combination was the most effective optimizaiton.

F.7 RELATED WORK

Tÿcho uses a language independent kernel representation as its intermediate representation. There are also other tools for stream programs with that property, for example SDF^3 [83] and DIF [25].

SDF³ is a set of tools for analyzing, transforming and scheduling synchronous dataflow graphs. A kernel of a synchronous dataflow program has the same token production and consumption rates on every firing [1]. Many interesting operations on these programs can therefor be performed without knowing what computation is actually performed in the kernels. SDF³ is made language-independent by ignoring the computation in the kernels. It also includes a tool for generating random graphs representing valid programs.

The Dataflow Interchange Format (DIF) is a language-independent text-based stream program description language. It is a vendor-agnostic format for tools and developers that can represent a wide variety of stream programs, with specific constructs for more restricted forms of dataflow, such as boolean-controlled dataflow [9], cyclo-static dataflow [7] and synchronous dataflow [1]. Similar to SDF³, the DIF package includes tools for computing properties and making transformations of the stream programs.

SDF³ and DIF recognizes specific classes of dataflow with firing that can be efficiently implemented, for example using static scheduling. The approach to scheduling in Tÿcho starts in the other end, by providing a scheduling technique through kernel fusions, that is efficient for statically schedulable programs, but applicable to other programs as well, with reduced efficiency.

In Section F.5.3 we compare the performance of the kernel fusion performed by Tÿcho to the Cal actor merger in [65]. The merger in [65] is based on the idea that one actor is leading the execution of a set of following actors that depend directly on the leader or some of the followers. The leader may have data dependent behavior but the behavior of the following actors should be determined by what the leader does. The Cal actor merger computes the variable dependency graph across all actors that are to be merged. From this graph, a subgraph is constructed containing only the nodes that could influence action selection. This information is later used to decide how the actions can be combined to larger actions. The result of this process is a single actor that performs the task of all the merged actors.

The kernel fusion based on actor machines that is presented in this paper, is arguably much simpler, because it only looks at the controllers, the buffer sizes and the token rates of the transitions and performs simple operations on them. On the other hand, the performance of the programs merged with [65] and compiled with Orcc was better than programs fused with the Tÿcho based solution presented in this paper.

Another problem addressed by this paper is transition selection, which is related to tree pattern matching; a language construct that is common in func-

tional programming languages such as ML. Both problems are about testing conditions to determine which part of the code to execute. A common optimization objective of both transition selection and of pattern matching is to test as few conditions as possible. In an early work in this area [84], the authors note that building the minimal decision tree is NP-hard, and describes some heuristics for finding small decision trees.

The most effective transition selectors were created using profiling data. This technique is often called profile-guided optimization, which is studied in [85]. We used two kinds of profiles for our heuristics, one counting how many times the conditions were true and false, respectively, and one counting how often the transitions were executed. These kinds of profiles are most similar to the control flow profiles, described in [85].

F.8 CONCLUSIONS

In this paper, we have presented a framework for compiling stream programs and how that framework solves three problems of stream program compilation. The first problem is about stream program kernels with several transitions—that can do different things in different situations—how to create efficient decision structures for transition selection. That problem was solved using the actor machine as the basis for the intermediate representation in Tÿcho, and further improved using actor machine reduction. We reviewed two reducers from one of our previous papers that use profiling data and presented one new reduction strategy that does not require profiling. An interesting direction for future work is towards more efficient reducers that does not use profiling data.

The second problem is also about making decisions; how to achieve efficient scheduling by fusing kernels to one. This problem was also solved in Tÿcho on the intermediate representation based on actor machines. We presented a new actor machine fusion algorithm and how it differs from our previous algorithm. We evaluated the new algorithm by comparing it to a Cal actor merger. An interesting extension of this work, inspired by the Cal actor merger, would be to fuse a sequence of transitions into a single transitions. We believe this extension could reduce the size of the controller, at the cost of changing scheduling properties and buffer size requirements.

The last problem is about how to efficiently implement variable sharing between conditions and transitions. First, we showed how to use liveness analysis for the variable scopes in the actor machine controller to remove some vari-

able initializations. We also showed how to lift variable declarations from the variable scopes of the actor machine to the conditions and transitions. These optimizations were also performed on the actor machine based intermediate representation in Tÿcho.

All three problems were solved on a language-independent representation of stream programs, and using techniques that are applicable to a broad class of stream programs. To demonstrate language independence and generality, we presented two compiler frontends; one for a Kahn process language and one for Cal—a language representing a very broad class of dataflow with firing.

Bibliography

[1] E. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications," in *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, (New York, NY, USA), pp. 189–198, ACM, 2008.

[3] R. Gu, J. W. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, vol. 63, no. 1, pp. 129–142, 2011.

[4] J. W. Janneck, "A machine model for dataflow actors and its applications," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pp. 756–760, IEEE, 2011.

[5] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[6] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, pp. 362–376, Springer, 1974.

[7] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397–408, 1996.

[8] T. M. Parks, J. L. Pino, and E. A. Lee, "A comparison of synchronous and cycle-static dataflow," in *Signals, Systems and Computers, 1995. 1995 Con-*

ference Record of the Twenty-Ninth Asilomar Conference on, vol. 1, pp. 204–210, IEEE, 1995.

[9] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, vol. 1, pp. 429–432, IEEE, 1993.

[10] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*, (Grenoble, France), Apr 2002.

[11] M. Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe, "MPEG-2 decoding in a stream programming language," in *International Parallel and Distributed Processing Symposium*, (Rhodes Island, Greece), Apr 2006.

[12] J. Eker and J. W. Janneck, "CAL language report: Specification of the CAL actor language," tech. rep., December 2003.

[13] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 29–35, 2008.

[14] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pp. 863–866, ACM, 2013.

[15] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software code generation for the RVC-CAL language," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 203–213, 2011.

[16] E. Gebrewahid, M. A. Arslan, A. Karlsson, and Z. Ul-Abdin, "Support for data parallelism in the CAL actor language," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '16, (New York, NY, USA), pp. 2:1–2:8, ACM, 2016.

[17] G. Kahn, "The semantics of a simple language for parallel programming," in *In Information Processing'74: Proceedings of the IFIP Congress*, vol. 74, pp. 471–475, 1974.

[18] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[19] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *J. ACM*, vol. 31, pp. 560–599, June 1984.

[20] D. May, "Occam," *SIGPLAN Not.*, vol. 18, pp. 69–79, Apr. 1983.

[21] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the SBF model of computation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 34, no. 3, pp. 291–300, 2003.

[22] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: Deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the eighth international workshop on Hardware/software codesign*, pp. 13–17, ACM, 2000.

[23] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic transformation techniques for efficient exploration of alternative application instances," in *Proceedings of the tenth international symposium on Hardware/software codesign*, pp. 7–12, ACM, 2002.

[24] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Rapid System Prototyping, 2008. RSP'08. The 19th IEEE/IFIP International Symposium on*, pp. 17–23, IEEE, 2008.

[25] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: An interchange format for dataflow-based design tools," in *International Workshop on Embedded Computer Systems*, pp. 423–432, Springer, 2004.

[26] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 111–116, European Design and Automation Association, 2009.

[27] M.-Y. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. F. Deprettere, "Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software

implementation," *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, 2007.

[28] J. Falk, C. Haubelt, and J. Teich, "Efficient representation and simulation of model-based designs in SystemC," in *Proc. of FDL*, vol. 6, 2006.

[29] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState—an internal design representation for codesign," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 524–544, 2001.

[30] J. Falk, C. Zebelein, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "Analysis of SystemC actor networks for efficient synthesis," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 2, p. 18, 2010.

[31] C. Zebelein, J. Falk, C. Haubelt, and J. Teich, "Classification of general data flow actors into known models of computation," in *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pp. 119–128, IEEE, 2008.

[32] E. Gebrewahid, M. Yang, G. Cedersjö, Z. U. Abdin, V. Gaspes, J. W. Janneck, and B. Svensson, "Realizing efficient execution of dataflow actors on manycores," in *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pp. 321–328, IEEE, 2014.

[33] N. Fors, G. Cedersjö, and G. Hedin, "JavaRAG: a Java library for reference attribute grammars," in *Proceedings of the 14th International Conference on Modularity*, pp. 55–67, ACM, 2015.

[34] D. E. Knuth, "Semantics of context-free languages," *Theory of Computing Systems*, vol. 2, no. 2, pp. 127–145, 1968.

[35] G. Hedin, "Reference attributed grammars," *Informatica (Slovenia)*, vol. 24, no. 3, pp. 301–317, 2000.

[36] J. W. Janneck, G. Cerdersjö, E. Bezati, and S. C. Brunet, "Dataflow machines," in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, pp. 1848–1852, IEEE, 2014.

[37] U. M. Mirza, M. A. Arslan, G. Cedersjö, S. M. Sulaman, and J. W. Janneck, "Mapping and scheduling of dataflow graphs—a systematic map," in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, pp. 1843–1847, IEEE, 2014.

[38] E. A. Lee, "A denotational semantics for dataflow with firing," Tech. Rep. UCB/ERL M97/3, EECS, University of California at Berkeley, Jan. 1997.

[39] J. Gorin, M. Wipliez, F. Prêteux, and M. Raulet, "LLVM-based and scalable MPEG-RVC decoder," *Journal of Real-Time Image Processing*, vol. 6, no. 1, pp. 59–70, 2011.

[40] L. Augustsson, "Compiling pattern matching," in *Functional Programming Languages and Computer Architecture*, pp. 368–381, Springer, 1985.

[41] S. Casale-Brunet, M. Mattavelli, and J. W. Janneck, "Profiling of dataflow programs using post mortem causation traces," in *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, pp. 220–225, IEEE, 2012.

[42] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. 100, no. 1, pp. 24–35, 1987.

[43] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow: model and implementation," in *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, vol. 1, pp. 503–507, IEEE, 1994.

[44] G. Cedersjö and J. W. Janneck, "Toward efficient execution of dataflow actors," in *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pp. 1465–1469, IEEE, 2012.

[45] M. Wipliez and M. Raulet, "Classification and transformation of dynamic dataflow programs," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pp. 303–310, IEEE, 2010.

[46] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *Signal Processing, IEEE Transactions on*, vol. 49, no. 10, pp. 2408–2421, 2001.

[47] K.-E. Årzén, A. Nilsson, and C. von Platen, "Model compiler," Tech. Rep. D1e 2.0 (M36 Release), Lund University, 2011.

[48] ISO/IEC, "Information technology – MPEG systems technologies – Part 4: Codec configuration representation." ISO/IEC 23001-4, 2nd Edition, 2011.

[49] S. Lee, T. Lim, E. Jang, J. H. Lee, and SeungwookLee, "MPEG Reconfigurable Graphics Coding framework: Overview and applications," in *Proc. 2011 IEEE Visual Communications and Image Processing Conference (VCIP 2011)*, IEEE, 2011.

[50] G. Cedersjö and J. W. Janneck, "Actor classification using actor machines," in *2013 Conference Record of the Forty Seventh Asilomar Conference on Signals, Systems and Computers*, IEEE, 2013.

[51] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Static scheduling of multi-rate and cyclo-static DSP applications," in *Workshop on VLSI Signal Processing*, IEEE Press, 1994.

[52] S. Stuijk, M. C. Geilen, B. D. Theelen, and T. Basten, "Scenario-Aware Dataflow: Modeling, analysis and implementation of dynamic applications," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, pp. 404–411, 2011.

[53] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Buffer capacity computation for throughput-constrained modal task graphs," *ACM Trans. Embed. Comput. Syst.*, vol. 10, pp. 17:1–17:59, Jan. 2011.

[54] P. Fradet, A. Girault, and P. Poplavko, "SPDF: A schedulable parametric data-flow MoC," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 769 –774, march 2012.

[55] S. S. Bhattacharyya, E. F. Deprettere, and B. D. Theelen, "Dynamic dataflow graphs," in *Handbook of Signal Processing Systems*, pp. 905–944, Springer, 2nd ed., 2012.

[56] S. Tripakis, D. Bui, B. Rodiers, and E. A. Lee, "Compositionality in Synchronous Data Flow: Modular code generation from hierarchical SDF graphs," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, no. UCB/EECS-2009-143, Oct 2009.

[57] J. W. Janneck, "Actor Machines — a machine model for dataflow actors and its applications," Tech. Rep. LTH 96-2011, LU-CS-TR 201-247, Department of Computer Science, Lund University, 2011.

[58] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software code generation for the RVC-CAL language," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 203–213, 2009. 10.1007/s11265-009-0390-z.

[59] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[60] S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, M. Mattavelli, and J. W. Janneck, "Methods to explore design space for MPEG RMC codec specifications," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1278–1294, 2013.

[61] J. Skeppstedt, "The ASIM Power Architecture simulator."

[62] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," research report, 1976.

[63] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel, "Dynamic expressivity with static optimization for streaming languages," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pp. 159–170, ACM, 2013.

[64] G. Cedersjö and J. W. Janneck, "Software code generation for dynamic dataflow programs," in *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pp. 31–39, ACM, 2014.

[65] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silven, "Actor merging for dataflow process networks," *Signal Processing, IEEE Transactions on*, vol. 63, no. 10, pp. 2496–2508, 2015.

[66] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, 1994.

[67] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[68] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: scalable threads for internet services," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 268–281, ACM, 2003.

[69] M. N. Krohn, E. Kohler, and M. F. Kaashoek, "Events can make sense.," in *USENIX Annual Technical Conference*, pp. 87–100, 2007.

[70] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management.," in *USENIX Annual Technical Conference, General Track*, pp. 289–302, 2002.

[71] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[72] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[73] A. C. J. Kienhuis, *Design space exploration of stream-based dataflow architectures*. TU Delft, Delft University of Technology, 1999.

[74] E. Bezati, S. C. Brunet, M. Mattavelli, and J. W. Janneck, "Synthesis and optimization of high-level stream programs," in *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pp. 1–6, Ieee, 2013.

[75] J. Merrill, "Generic and gimple: A new tree representation for entire functions," in *Proceedings of the 2003 GCC Developers' Summit*, pp. 171–179, 2003.

[76] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[77] S. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[78] S. Marlow and S. P. Jones, "The Glasgow Haskell compiler," tech. rep., 2012.

[79] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, "Scalable framework for mapping streaming applications onto multi-GPU systems," *SIGPLAN Not.*, vol. 47, pp. 1–10, Feb. 2012.

[80] E. Bezati, M. Mattavelli, and J. W. Janneck, "High-level synthesis of dataflow programs for signal processing systems," in *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, pp. 750–754, IEEE, 2013.

[81] G. Cedersjö and J. W. Janneck, "Processes and actors: Translating Kahn processes to dataflow with firing," in *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*, pp. 21–30, IEEE, 2016.

[82] G. Cedersjö, J. W. Janneck, and J. Skeppstedt, "Finding fast action selectors for dataflow actors," in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, pp. 1435–1439, IEEE, 2014.

[83] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF for free," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pp. 276–278, IEEE, 2006.

[84] M. Baudinet and D. MacQueen, "Tree pattern matching for ML." 1985.

[85] R. Gupta, E. Mehofer, and Y. Zhang, "Profile-guided compiler optimizations," in *The Compiler Design Handbook: Optimizations and Machine Code Generation*, CRC Press, 2002.