Checking for Circular Dependencies in Distributed Stream Programs

Dai Bui
Hiren Patel
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-97
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-97.html

August 29, 2011

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: PTIDES) and #0931843 (ActionWebs)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MySyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

Checking for Circular Dependencies in Distributed Stream Programs

Dai N. Bui, Hiren D. Patel, Edward A. Lee
{daib,eal}@eecs.berkeley.edu, h.patel@ece.uwaterloo.ca
University of California, Berkeley

Abstract

This work presents a cyclic dependency analysis for stream-based programs. Specifically, we focus on the cyclo-static dataflow (CSDF) programming model with control messages through teleport messaging as implemented in the StreamIt framework. Unlike existing cyclic dependency analyses, we allow overlapped teleport messages. An overlapped teleport message is one that traverses actors that themselves transmit teleport messages, which can complicate the stream graph topology with teleport messages. Therefore, the challenge in this work is to decide whether such stream graphs are feasible in the presence of such complex teleport messages. Our analysis addresses this challenge by first ensuring that the stream graph with teleport messages is feasible, and then computing an execution schedule for the CSDF graph in the presence of complex overlapped teleport messaging constraints. Consequently, our analysis accepts a larger class of CSDF stream graphs with complex teleport messaging topologies for execution.

General Terms Languages, Semantics, Design

Keywords Streaming, Dependency analysis, Scheduling, Deadlock detection.

1. Introduction

Streaming applications are an important class of applications common in today's electronic systems. Examples of streaming applications constitute image, video and voice processing. To facilitate precise and natural expression of streaming applications, research proposes several streaming languages such as [1, 5, 7, 15, 21] that allow programmers to faithfully model streaming applications. These languages employ high-level domain abstractions instead of low-level languages such as C to enable portability and automatic optimizations for a variety of target architectures including novel multicore platforms. For example, static dataflow (SDF) [19], cyclo-static dataflow (CSDF) [3], multidimensional synchronous dataflow (MDSDF) [22], and Kahn process network(KPN) [14] are models of computation well-suited for modeling and synthesis of streaming applications.

In those above models of computations for streaming applications, SDF and its derivatives, such as CSDF and MDSDF, are more static than KPN, however, compilers could optimize applications in those domains for buffer space, scheduling, balanced mapping on multicore more easily than for general KPN applications. However, the static nature of SDF and its derivatives makes it difficult to implement more dynamic streaming applications such as changing rates of tokens that a task can produced or parameters used to compute with data. Several efforts have tried to address the issue such as parameterized dataflow [2] used in modeling image processing systems [28].

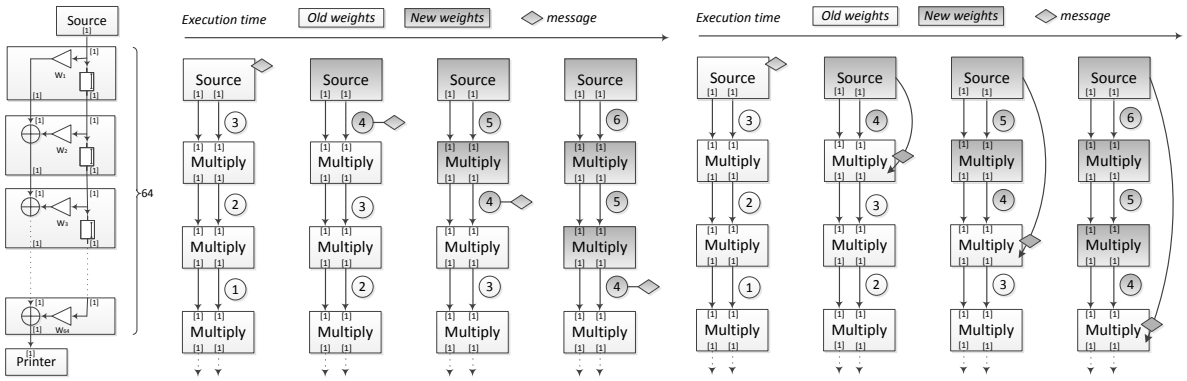
However, as streaming algorithms become more and more dynamic and complicated to enable higher quality of service while keeping underlying hardware systems at a reasonable price and power-efficient, streaming languages' original abstractions may no longer be able to capture all new complexities in a natural way. To solve the above problem, language designers should come up with new language extensions to express new complexities more convenient. Teleport messaging (TMG) in the StreamIt language [30] is an example.

1.1 StreamIt Language and Compiler

StreamIt [29] is a language for streaming applications based on the CSDF [19] programming model, a generalization of the Synchronous Dataflow (SDF) [19] model of computation. The language exploits inherent task-level parallelism and predictable communication properties of CSDF to partition and mapping a stream program onto different multicore architectures [8, 11]. A StreamIt application is a directed graph of autonomous actors connected via FIFO buffers of predictable sizes due to static production and consumption rates of actors. This static feature of CSDF enables compilers to optimize, and transform programs to efficiently deploy them onto different architectures. The CSDF model of computation is suitable for expressing regular and repetitive computations. However, CSDF makes it difficult to express dynamic streaming algorithms because of its requirement to enforce periodic and static schedules. As a result, dynamic streaming algorithms require substantial modifications to the streaming program structures itself, which makes using CSDF a complex task for such applications.

1.2 Control Messages

In contrast to high-frequency regular data messages such as those typically modeled in CSDF, infrequent control messages sent between actors. Control messages are necessary to enable implementing more dynamic streaming application algorithms, e.g. they could be used to adjust the employed protocol, and the compression rate. Manually integrating infrequent low rate control messages into frequent high-rate streams would complicate the overall structure of an application, which then makes it difficult for users to maintain and debug such an application, and also potentially reduce its efficiency. It is useful to separate control from data computation, as in [2], so that compiler optimization methods can be applied to generate more efficient code.



(a) FIR example

(b) New weights are attached with data tokens and processed at actors before processing data

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

Figure 1. Adding dynamicities to an FIR computation

Thies et al. in [30] give a TMG model for distributed stream programs. TMG is a mechanism that implements control messages for stream graphs. The TMG mechanism is designed not to interfere with original dataflow graphs' structures and scheduling, therefore a key advantage of TMG is that it incorporates control messages without requiring the designer to restructure the stream graph, recompute production and consumption rates, and further complicate the program code for the actors. However, it still needs to be precise relatively to data messages. For instance, a set of new parameters specified in a control message should only take effect on some designated data messages. This requires synchronization methods between data messages and control messages. Moreover, the structure of stream graphs with TMG exposes dependencies that allow automated analytical techniques to reason about timing of control messages. Naturally, stream graph compilers can implement these analysis techniques. Users can also change latency of messages without changing the structures of stream graphs.

Let us illustrate the problem with an example. Consider the Finite Impulse Response (FIR) example from [30], shown in Figure 1(a). FIR is a common kind of filter in digital signal processing, where the output of the filter is a convolution of the input sequence with a finite sequence of coefficients. The FIR example is composed of a `Source`, a `Printer`, and 64 `Multiply` and `Add` actors. Each `Multiply` actor has a single coefficient w, called a *tap weight*, of a FIR filter.

Now suppose that during the execution at some iteration, the `Source` actor detects some condition, and it decides to change the tap weights of the `Multiply` actors. The new set of weights should only be used to compute with data produced by the `Source` actor during and after its current execution.

One way to ensure that the new coefficients are only used with the appropriate data is to attach the new set of tap weights with data packets sent from the source, as in Figure 1(b). In the figure, each time a data token attached to a message with new weights arrives, each `Multiply` actor will detach the message and update its weight and use the new weight to compute with the just arrived data token. However, this approach is not efficient; it changes the structure and data packets, and would require an aggressive compiler analysis to optimize a program as well as minimize communication buffer sizes between actors in a stream graph.

In contrast, with the teleport approach, illustrated in Figure 1(c), the `Source` actor could send teleport messages (TM) containing new weights to each `Multiply` actor before data tokens that need to be computed with new weights arrive. This approach provides a clean separation between control execution and data computation. This separation makes it easier to maintain and debug programs, it also helps avoid error-prone task of manual embedding and processing control information within data tokens.

The TMG mechanism requires synchronization between data tokens and control messages, so that a control message is only handled just before the computation of the appropriate data token. The theory of StreamIt TMG synchronization method provides an `SDEP` function to accomplish this synchronization. However, the `SDEP` function alone is not powerful enough to enable the StreamIt compiler to handle the case where several TMs are *overlapped* in a stream graph. This limitation hinders the deployment of more complicated stream programs in StreamIt.

1.3 Circular Dependencies

Figure 1(c) uses a simplified notion of TMs, where the *latency* of the message is implicitly set to 0. Latency constraints are needed in order to specify which instance of an actor can receive TMs from which instance of another actor. A integer parameter called latency annotating a TM is used to achieve this. In Figure 1(c), all latencies are implicitly zero. This means that the n^{th} execution of `Source` may send a TM to the n^{th} (but no earlier nor later) execution of each of the `Multiply` actors.

In general, latencies can be different from zero, as shown in Figure 2. It is then possible that there does not exist a schedule that delivers TMs with the desired latencies. The left side of Figure 2 shows an example where the set of latency constraints is not satisfiable. Let us explain the example. Let us denote by A_m the m^{th} execution of actor A. Suppose that actor A is currently at its n^{th} execution. Then, we have the following constraints:

- The latency constraint imposed by TMs sent from actor D to actor A requires that A_{n+1} wait for possible TMs from D_n before it can execute. In other words, A_{n+1} depends on D_n and A_{n+1} has to execute after D_n. Let's denote this as $D_n \prec A_{n+1}$.
- As B_{n+1} consumes one token produced by A_{n+1}, we have $A_{n+1} \prec B_{n+1}$.

- We assume that multiple executions of a single actor must proceed in sequential order, so $B_{n+1} \prec B_{n+2}$, and hence, of course, $B_{n+1} \prec B_{n+10}$
- Again, the latency imposed by teleport messages sent from actor B to actor C constrains C_n to wait for a possible TM from B_{n+10}, therefore, $B_{n+10} \prec C_n$.
- Finally, D_n consumes one token produced by C_n, therefore $C_n \prec D_n$.

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

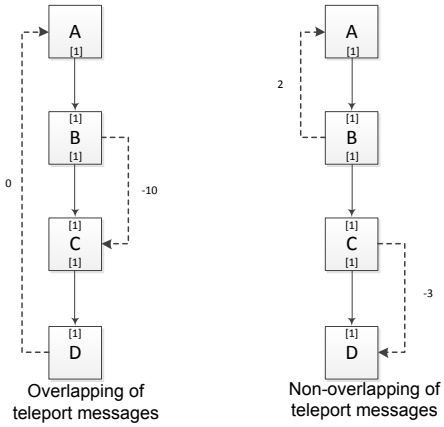


Figure 2. Overlapping and non-overlapping scenarios of teleport messages.

There are two factors creating cyclic dependencies of actor executions for a stream graph: i) the structure of the stream graph, and ii) the latencies of teleport messages. Let us take an example to illustrate the importance of the two factors.

Now, we keep the same graph structure in the left side of Figure 2, however, we suppose that the latency for TMs between actor B an actor C is 0. In this case, there exists a valid evaluation order: $\ldots \prec A_n \prec B_n \prec C_n \prec D_n \prec A_{n+1} \prec B_{n+1} \prec \ldots$. We can see that for the same stream graph *structure*, different TM latencies could result in completely different situations; the first one is a deadlocked while the second one has a valid schedule.

Thies et al. [30] call the graph structure on the left of Figure 2 "overlapping constraints," because the paths between actors involved in TMs have some actors in common. The compiler simply rejects graph structures that have overlapping TM situations regardless of message latencies even if the latencies could result in valid schedules, as is case if the latency between actor B and actor C is 0. The StreamIt compiler only allows non-overlapping TMs, as on the right side of Figure 2. This conservative approach reflects an underdeveloped theory of execution dependency in the StreamIt compiler.

The first contribution of our paper is a method for checking circular dependencies of distributed stream programs in the presence of overlapping of TMs by introducing static finite dependency graphs for infinite sequences of executions then proposing an algorithm for directly constructing such graphs from stream graphs. The second contribution of this paper is to show how to find an execution order for a stream graph when there is no circular dependencies by solving a linear program. We have implemented our checking and ordering methods as a backend for the StreamIt compiler.

2. Background

2.1 Programming Model

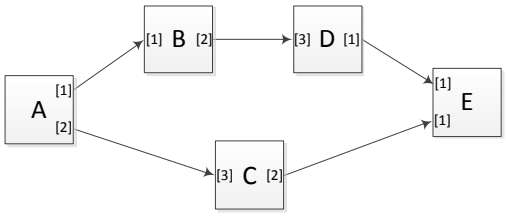


Figure 3. Cyclic Static Dataflow model of computation

The CSDF/SDF model of computation captures the semantics of streaming applications and allows several optimization and transformation techniques for buffer sizes [23], partitioning and mapping of stream graphs onto multicore architectures [8, 9, 29], and computational methods [10, 18].

In the CSDF model of computation, a stream program, given as a graph, is a set of actors communicating through FIFO channels. Each actor has a set of input and output ports. Each channel connects an output port of an actor to an input port of another actor. Each actor cyclically executes through a set of execution steps. At each step, it consumes a fixed number of tokens from each of its input channels and produces a fixed number of tokens to each of its output channels. This model of computation is interesting as it allows static scheduling with bounded buffer space.

Figure 3 shows an example of a CSDF stream graph. Actor A has two output ports. Each time actor A execute, it produces 2 tokens on its lower port and 1 token on its upper port Actor B consumes 1 and produces 2 tokens each time it executes. Actor E alternately consumes 0 and 1 token on its upper port and 1 and 0 tokens on its lower port each time it executes. The theory of CSDF and SDF provides algorithms to compute the number of times each actor has to execute within one *iteration* of the *whole* stream graph, so that the total number of tokens produced on each channel between two actors is equal to the total number of tokens consumed. In other words, the number of tokens on each channel between actors remains unchanged after one iteration of the whole stream graph. For example, in one iteration of the stream graph in Figure 3, actors A, B, C, D, E have to execute 3, 3, 2, 2, 5 times respectively. A possible schedule for one iteration for the whole stream graph is $3(A), 3(B), 2(C), 2(D), 5(E)$. This basic schedule can be iteratively repeated. As we can see with this basic schedule, the number of tokens on each channel remains the same after one iteration of the whole stream graph. For instance, in the channel between B and D, in one iteration, B produces $3 \cdot 2$ tokens while D consumes $2 \cdot 3$ tokens.

2.2 Teleport Messaging

TMG enables executing an actor B asynchronously with an actor A. Ordering constraints implied by a TM tagged with a *latency*, which specifies message processing delay, must be enforced. The latency parameter determines the execution of B at which the TM handler in B is invoked. To enable this, actor B declares a message handler function. Then, the container[1] of actor A and B then declares *portal*, a special type of variable, of the type `portal<B>`, as in Figure 4. Each portal variable is associated with a specific type, e.g. `portal<B>`, which is again associated with specific actor type, e.g. B. A portal variable of type `portal<B>` could invoke messages handlers declared within B. This portal

[1] Actors within a containter are similar to objects contained within another object.

variable is then passed to the entry function of actor A to enable actor A to invoke the message handler functions of actor B with some latency parameters. For example, for actor A to send a TM to B with a latency of k means that on the k^{th} iteration of firing actor A, the TM is sent alongside with the data token to actor B. Once the k^{th} data token reaches actor B, actor B fires the message handler before consuming the data token. Other actors, such as C could also use the portal to send B a message.

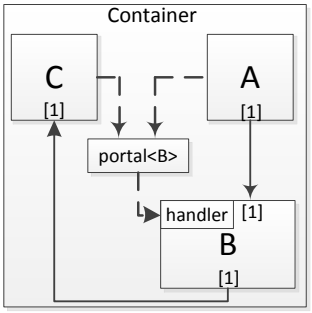


Figure 4. Example of teleport portal.

2.3 TMs Timing with SDEP

As TMs are sent between actors, it is mandatory that we need to have a way to specify when TMs should be processed by receiving actors as current (time when TMs are sent) status of receiving actors could be not appropriate to process TMs. In other words, we would need to find exact executions of receiving actors that TMs should be processed. The following SDEP function provides a way to find processing *time* of TMs for receiving actors.

Thies et al. [30] formally present an approach to compute the invocation of actors based on their dependencies. We borrow their formulation, and use it in our dependency analysis. We briefly describe the semantics from [30] for the reader.

We first present Definition 1, which is a stream dependency function SDEP that represents the data dependency of executions of one actor on the execution of other actors in the stream graph. $\texttt{SDEP}_{A \leftarrow B}(n)$ represents the minimum number of times actor A must execute to allow actor B to execute n times. This is based on the intuition that an execution of a downstream actor requires data from some execution of an upstream actor; thus, the execution of the downstream actor depends on some execution of the upstream actor. In other words, given an execution n^{th} of a downstream actor B, SDEP function could return the latest execution of a upstream actor A that the data it produces, going though and being processed by intermediate actors, will affect the input data consumed by execution n^{th} of actor B.

DEFINITION 1. *(SDEP)*

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

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

Using the above SDEP function, we could formally specify when TMs are processed as follows:

DEFINITION 2. *Suppose that actor A sends a TM to actor B with latency range $[k_1 : k_2]$ during the n^{th} execution of A. Then, we consider two cases:*

- *If B is downstream of A, then the message handler must be invoked in B immediately* before *its m^{th} execution, where m is constrained as follows:*

$$n + k_1 \leq \texttt{SDEP}_{A \leftarrow B}(m) \leq n + k_2 \quad (1)$$

- *If B is upstream of A, then the message handler must be invoked in B immediately* after *its m^{th} execution, where m is constrained as follows:*

$$\texttt{SDEP}_{B \leftarrow A}(n + k_1) \leq m \leq \texttt{SDEP}_{B \leftarrow A}(n + k_2) \quad (2)$$

To illustrate the usage SDEP to find appropriate executions of receiving actors, we take the FIR example in Figure 1(c) and try to find the execution m of a Multiply that will need to process a TM. Suppose that at the 5^{th} execution ($n = 5$), the Source actor sends a TM to a Multiply actor with latency $k_1 = k_2 = 2$. Then we have:

$$\begin{aligned} n + k_1 \leq & \texttt{SDEP}_{\texttt{Source} \leftarrow \texttt{Multiply}}(m) \leq & n + k_2 \\ 5 + 2 \leq & \texttt{SDEP}_{\texttt{Source} \leftarrow \texttt{Multiply}}(m) \leq & 5 + 2 \\ 7 \leq & \texttt{SDEP}_{\texttt{Source} \leftarrow \texttt{Multiply}}(m) \leq & 7 \\ & \texttt{SDEP}_{\texttt{Source} \leftarrow \texttt{Multiply}}(m) = & 7 \end{aligned}$$

Each time the Source actor fires, it produces one token and each time one Multiply actor fires, it produces one token and consumes one token, therefore, in order for a Multiply actor fires m times, the Source actor has to fires m times, in other words, $\texttt{SDEP}_{\texttt{Source} \leftarrow \texttt{Multiply}}(m) = m$. Hence, $m = 7$.

2.4 SDEP Calculation

StreamIt computes the SDEP function using the simple pull schedule [30]. For brevity, we refer the readers to [30] for further details on the algorithm and details. However, to intuitively illustrate the SDEP calculation, we take a simple example of actors B and D in Figure 3. The $\texttt{SDEP}_{B \leftarrow D}(m)$ is as in Table 1. In the example, when D does not execute, it does not require and number of executions of B. In order for D to execute the first time, it requires three tokens on its input channel. Based on this requirement, the *pull* schedule algorithm will try to pull three tokens from the supplier, actor B. To supply the three tokens, B has to execute at least two times, therefore we have $\texttt{SDEP}_{B \leftarrow D}(1) = 2$. Similar, when D wants to execute one more time, it needs two more tokens so it will try to pull the two tokens from B. Again, B has to execute one more time to supply the two tokens and we have $\texttt{SDEP}_{B \leftarrow D}(2) = 3$.

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

Table 1. Dependency Function Example

Periodicity of SDEP: As CSDF is periodic, therefore SDEP is also periodic. This means that one does not need to compute SDEP for all executions, instead, one could compute SDEP for some executions then based on the periodic property of SDEP to query future dependency information. The following equation was adapted from [30]:

$$\texttt{SDEP}_{A \leftarrow B}(n) = i * |\mathcal{S} \wedge A| + \texttt{SDEP}_{A \leftarrow B}(n - i * |\mathcal{S} \wedge B|) \quad (3)$$

where $\mathcal{S}$ is the execution of actors within one iteration of a stream graph and $|\mathcal{S} \wedge A|$ is the number of executions of actor A within one iteration of its stream graph. i is some iteration such that $0 \leq i \leq p(n)$ where $p(n) = n \div |\mathcal{S} \wedge B|$ is the number of iterations that B has completed by its execution n.[2]

[2] We define $a \div b = \lfloor \frac{a}{b} \rfloor$

3. Execution Dependencies

The StreamIt compiler relies on the SDEP dependency function to determine when a TM handler could be invoked based on message latencies. TMs actually impose constraints on execution orders of actors on the path between senders and receivers as actors cannot execute arbitrarily whenever data are available at their inputs, rather, TM receiving actors have to wait for possible TMs so that they cannot execute too far ahead. Constraints on executions of receiving actors will again constrain executions of intermediate actors (actors between TMG senders and receivers). When intermediate actors on the path between a sender and a receiver are not involved in TMG communication, in other words, there are no TMs overlapped as on the right side of Figure 2, only a single constraint is imposed on execution orders and a valid message latency could be easily checked using the following condition: an upstream message cannot have a negative latency.

When some intermediate actors on the path between a sender and a receiver are also involved in some other TMG communication as on the left side of Figure 2, this scenario is called overlapping of TMs or overlapping constraints by Thies et al. [30]. Because additional constraints impose on execution orders of intermediate actors, added constraints might make it impossible to schedule executions of actors as in the example in Section 1.3 because of circular dependencies.

Checking for this general scheduling problem with overlapping of TMs is not straightforward as the SDEP stream dependence functions are not *linear*, e.g. as in Table 1, due to mis-matching input/output rates of actors.

To solve the above circular dependencies checking problem in the presence of overlapping of TMs, we could exploit the graph unfolding technique [25] to construct a directed graph that helps reason about dependencies between executions of actors in a stream graph.

However, to construct such a directed execution dependency graph, we first need to characterize and classify all kinds of execution dependencies.

3.1 Actor Execution Dependencies

DEFINITION 3. *Execution dependency: An execution e_1 of an actor is said to be dependent on another execution e_2 of some actor (could be the same actor) when e_1 has to wait until e_2 has finished before it can commit its result.*

From our insight, we have three kinds of execution dependency as follows:

- Causality dependency: The $(n+1)^{th}$ execution of A has to be executed after the n^{th} execution of A, or $A_n \prec A_{n+1}$
- Data dependency: For two directly connected actors A and B and actor B is a downstream actor of actor A, then an n^{th} execution of B, let's call B_n, will be data-dependent on a m^{th} execution of A, called A_m, if B_n consumes one or more tokens produced by A_m.
- Control dependency due to TMs sent between actors. As an actor S at iteration n^{th} sends a TM to an actor B with latency $[k_1, k_2]$, then for all the m^{th} executions of R satisfying the Definition 2, R_m is said to be control dependent on S_n as it might consumes some control information from S_n.

3.2 Directed Execution Dependency Graph

To check for circular dependencies we would need to construct a dependency graph and check for circles in that graph. If there is no circle in a dependency graph, then the graph is just a directed acyclic graph, and an evaluation order could be found using topological sort. Our dependency classification in the previous section enables us to construct such a directed execution dependency graph of actor executions.

The construction of such a directed dependency graph is done in two steps. First, we simply replicate executions of actors in its original CSDF model of computation. Second, we add causality dependency, data dependency and control dependency edges to the graph.

The first step could be done by expanding iterations of a CSDF stream a graph. For example, we have a stream graph as in Figure 3. Within one iteration of the whole stream graph, according to CSDF model of computation, actors A, B, C and D execute 6, 3, 2 and 4 times respectively. Each execution of an actor is replicated as one vertex in execution dependency graph as in Figure 5. Note that we use A_2^i to denote the 2^{nd} *relative* execution of actor A within the i^{th} iteration of the stream graph, this is equivalent to the $(i*6+2)^{th}$ *absolute* (from the beginning when the program starts) execution of actor A as A executes 6 times in one iteration for the stream graph in Figure 3. [3]

In the second step, although dependency edge calculation is based on our dependency classification, however, detailed methods for calculating the dependency edges have not been presented. In the following section, we will show how to calculate execution dependencies for the second step.

3.2.1 Calculating Execution Dependencies

Although we enumerated three kinds of execution dependencies in Section 3.1, so far we have not shown how to compute those dependencies. Computing causality dependency is straightforward.

For data dependency edges, we utilize the SDEP function implemented using the pull scheduling algorithm in [30]. For any two connected actors A, upstream, and B downstream, we create a dependency edge between the execution $\texttt{SDEP}_{A \leftarrow B}(n)^{th}$ of A to the execution n^{th} of B, we denote $A_{\texttt{SDEP}_{A \leftarrow B}(n)} \prec B_n$. Note that $A_m \prec A_{m+1}$ based on causality dependency condition, therefore, $B_n \succ A_m, \forall m = 1 \rightarrow \texttt{SDEP}_{A \leftarrow B}(n)$. Thus, we do not need to add any dependency edges between B_n and $A_m, \forall m < \texttt{SDEP}_{A \leftarrow B}(n)$, as those dependencies are *implicit* and could be inferred from $A_m \prec A_{m+1}$ and $A_{\texttt{SDEP}_{A \leftarrow B}(n)} \prec B_n$.

Finally, for control dependency edges due to TMs, those dependencies can already be computed using SDEP as proposed in [30]. If an actor S sends a TMs to an actor R with latencies $[k_1, k_2]$. Applying the Definition 2, we have two cases:

- If R is downstream of S, then create an edge $R_m \rightarrow S_n$ s.t. $n + k_1 \leq \texttt{SDEP}_{S \leftarrow R}(m) \leq n + k_2$. However, because of the causality dependency and SDEP is monotonic, e.g. $R_{m|\texttt{SDEP}_{S \leftarrow R}(m)=n+k_1} \preceq R_{m|\texttt{SDEP}_{S \leftarrow R}(m)=n+k_2}$, and we would like to have as few dependency edges as possible, therefore we only need to add one dependency edge $R_{m|\texttt{SDEP}_{S \leftarrow R}(m)=n+k_1} \rightarrow S_n$
- If R is upstream of S, then create an edge $R_m \rightarrow S_n$ such that $\texttt{SDEP}_{R \leftarrow S}(n + k_1) \leq m \leq \texttt{SDEP}_{R \leftarrow S}(n + k_2)$. For the same reasons as in the previous case, we only add one edge $R_{\texttt{SDEP}_{R \leftarrow S}(n+k_1)+1} \rightarrow S_n$. We need to add one to $\texttt{SDEP}_{R \leftarrow S}(n + k_1)$ because the message handler at R is invoked *after* the execution $\texttt{SDEP}_{R \leftarrow S}(n + k_1)$ of R.

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

3.2.2 Illustrating Example

Let us come back to the example in Figure 3, however, now we suppose that E sends TMs to A with latency 0 and B sends TMs to D with latency -1, then the execution dependency graph is as in Figure 5, where the causality dependency edges are exhibited using arrows with dashed lines, data dependency edges are in dash-dot arrows, and control dependency edges are in solid arrows.

We use the function SDEP between actors B and D in Table 1 to illustrate our method. For actors B and D, the dependency SDEP function is given in Table 1, then for any iteration n of the stream graph, we add data dependency edges $D_1^n \rightarrow B_2^n$ and $D_2^n \rightarrow B_3^n$ as in Figure 5 because within one iteration, the first execution of D is data dependent on the second execution of B and the second execution of D is data dependent on the third execution of B.

For control dependency edges, as TMs from actor B to D have delays of $k_1 - 1$, and $\texttt{SDEP}_{B \leftarrow D}(n * \texttt{e}(D) + 1) = (n * \texttt{e}(B) + 3) - k_1, \forall n \in \mathbb{N}$, where $\texttt{e}(X)$ is the number of times actor X executes within one iteration of a stream graph, therefore, we add an edge $D_1^n \rightarrow B_3^n$. Similarly , $\texttt{SDEP}_{B \leftarrow D}(n * \texttt{e}(D) + 2) = ((n+1) * \texttt{e}(B) + 1) + k_1, \forall n \in \mathbb{N}$, we add an edge $D_1^n \rightarrow B_1^{n+1}$.

Although, Figure 5 only shows dependencies for a portion of a whole infinite graph, the basic dependency structures of the whole execution dependency graph are similar for all iterations $n \in \mathbb{N}$, except for some initial iterations, as CSDF model of computation is periodic.

We can see that, the above naive graph construction process will result in an *infinite* directed graph as the execution sequence of a streaming application based on CSDF model of computation is presumed to be infinite. Because an infinite directed graph could not be used directly to check for cyclic dependency of actor executions, we need a way to translate them into a similar *finite* directed graph that captures all dependency structures in the original infinite graph.

4. Checking for Circular Dependencies

4.1 Dynamic/Periodic Graph

The technique for translating such infinite execution dependency graphs into finite static graph is already available in the *dynamic/periodic graph* theories [13, 24] if we notice that the infinite execution dependency graphs are *periodic* similarly to the dynamic/periodic graph definition below.

DEFINITION 4. *A directed dynamic periodic graph $G^\infty = (V^\infty, E^\infty)$ is induced by a static graph $G = (V, E, T)$, where V is the set of vertices, E is the set of edges, $T : E \rightarrow \mathbb{Z}^k$ is a weight function on the edges of G, via the following expansion:*

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

If we interpret $t_{uv} \in T$ as transit time representing the number of k-dimensional *periods* it takes to travel from u to v along the edge, then the vertex v^p of G^∞ could be interpreted as vertex v of G in a k-dimensional *period* p and edge $(u^p, v^{p+t_{uv}})$ represents *travelling* from u in period p and arriving at v t_{uv} periods later.

Intuitively, a k-dimensional periodic graph is obtained by replicating a basic graph (cell) in a k-dimensional orthogonal grid. Each vertex within the basic graph is connected with a finite number of other vertices in other replicated graphs and the inter-basic-graph connections are the same for each basic graph.

Our observation is that an execution dependency graph of a CSDF stream graph is an infinite 1-dimensional dynamic graph with its basic graph composed of vertices that are actor executions of the CSDF stream graph within one iteration. The basic cell is repeatedly put in a 1-dimensional time grid. Data, causality and control dependencies form directed edges between vertices. Some of edges created by causality and control dependencies could be inter-cell (inter-iteration) connections. As the CSDF model of computation is periodic by nature, the pattern of the inter-cell (inter-iteration) connections is the same for each cell (iteration).

Based on the above observation and the theory of infinite periodic graph, to check for cyclic dependency of an infinite dependency graph with TM constraints, we could construct an equivalent *static* finite graph. We then can prove that there is a cyclic dependency in a execution dependency graph of a CSDF program with added TM constraints iff the equivalent static graph has a cycle with weight equal to 0. In the next section, we will show how to translate 1-dimensional periodic graph in Figure 5 into a static finite graph.

4.2 Translating to Static Finite Equivalent Graph

Figure 6(a) shows the equivalent static finite dependency graph of the infinite execution dependency graph in Figure 5. In the graph, all edges have do not have specified weights are of weight 0.

Intuitively, all the vertices within one arbitrary iteration, say iteration n, are kept to form vertices in the equivalent static graph, however, iteration indices are removed, e.g. A_1^n becomes A_1. Directed edges between vertices within the one iteration are also kept and their weights are set to 0. For directed edges cross iterations, only *outgoing* edges (edges from this iteration to some other iterations) are used to translate to equivalent edges in the new static graph. The translation is done as follows, suppose that an outgoing edge is $S_x^n \rightarrow R_y^m$, then we add a directed edge $S_x \rightarrow R_y$ with weight $n-m$. $n-m$ is called *relative iteration*, which is the gap between iterations of two actor executions. For example, the directed edge $D_2^n \rightarrow B_1^{n+1}$ in Figure 5 becomes the edge $D_2 \rightarrow B_1$ with weight -1 in Figure 6(a). Note that an edge $S_x \rightarrow R_y$ is equivalent to *any edge* $S_x^i \rightarrow R_y^j$ in the execution dependency graph as long as $i - j = n - m$ because of the repetitive nature of the CSDF model of computation.

4.3 Graph Equivalence

We cite the following Lemma 1 from Lemma 1 in [24].

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

The above lemma is useful for proving the following theorem:

THEOREM 1. *A dependency circle in an execution dependency graph is equivalent to a cycle with length of zero in the equivalent static graph.*

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

4.4 Detecting for Zero Cycles

We have shown that a circle of execution dependencies is equivalent to a cycle of zero weight in an equivalent graph. However, we have not shown how a cycle of zero weight is detected. In [13], Iwano and Steiglitz propose an algorithm for detecting zero cycles in an 1-dimensional[4] static graph with complexity $\mathsf{O}(n^3)$ (Theorem

[4] Distances between vertices have only one dimension.

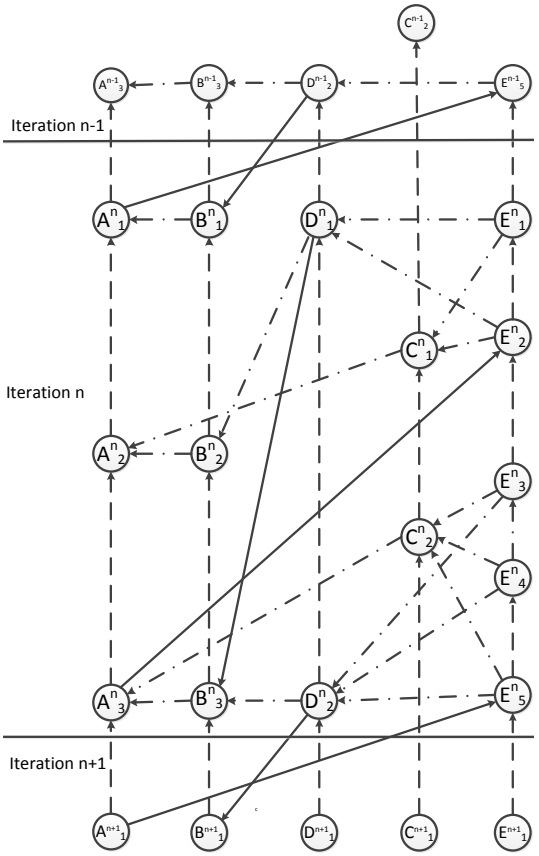
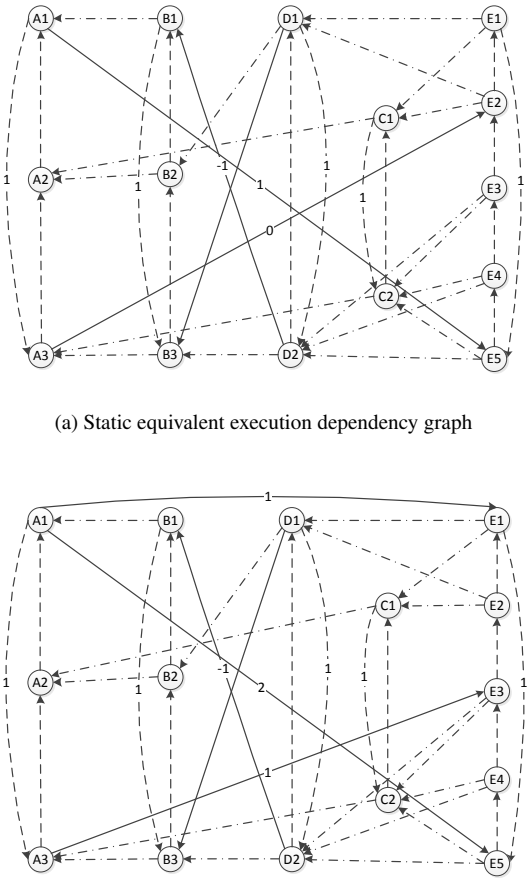


Figure 5. The execution dependency trace of a CSDF stream graph with teleport messaging is an infinite periodic directed execution dependency graph



(a) Static equivalent execution dependency graph

(b) Static execution dependency graph without zero cycles

Figure 6. Static graphs

4 in [13]) where n is the number of vertices in the 1-dimensional static graph.

4.5 Illustrating Example

To illustrate the circle checking method better, we take the translated static equivalent graph in Figure 6(a). We run the cycle detection algorithm in [13] and detect a zero cycle, A_1, E_5, D_2, B_1, this zero cycle in the static graph is equivalent to the cycle of dependencies in the execution dependency graph in Figure 5, $A_1^n \succ E_5^{n-1} \succ D_2^{n-1} \succ B_1^n \succ A_1^n$.

Now, when the latency of TMs from E to A is -4, the equivalent static graph is shown on Figure 6(b). The graph has no zero cycles, thus, there is no circle of dependencies in the execution dependency graph, therefore, the set of constraints imposed by teleport communication is feasible.

5. Direct Construction of Static Graphs

In the previous section, we assumed that we already have constructed an infinite dynamic execution dependency graph already and showed how to convert it to a finite static equivalent graph. However, it is not possible and useful to construct an infinite graph from a stream program, instead, we could use a similar mechanism to construct the static equivalent graph directly from a stream program as we know the repetitive dependency structures across iterations of a CSDF stream graph. Basically, we would like to construct weight function T of a $G = (V, E, T)$ from a CDSF stream graph. The Algorithm 1 shows how to the direct translation process works.

Similar to infinite directed dependency graphs, equivalent static graphs are constructed in two steps. First, executions of one actors are replicated the same number of times that the actor fires *within one iteration* of a CSDF model, each execution becomes one vertex in the equivalent static graph. The next step is to add dependency edges between vertices based on three kinds of dependency enumerated in Section 3.1 and methods presented in Section 3.2.1.

In the Algorithm 1, where `get_num_reps` function returns the number of repetitions of an actor within one iteration of the whole stream graph. The function `compute_rel_iter_exe` computes *relative* iteration i_r and *relative* execution e_r, we elaborate more on the meanings of those return values.

Algorithm 1 Algorithm for Constructing Static Equivalent Graphs

$(V, E, T) \leftarrow (\emptyset, \emptyset, \emptyset)$
$sched \leftarrow$ `compute_CSDF_schedule`$(streamGraph)$
{Create the set of vertices}
for all $actor$ **do**
  for $exe = 1 \rightarrow sched.$`get_num_reps`$(actor)$ **do**
    {Each vertex is one actor execution in one iteration}
    $V \leftarrow V +$ `new_vertex`$(actor, exe)$
  end for
end for
{Add data dependency edges}
for all $v \in V$ **do**
  for all $actor \in$ `upstream_actors`$(v.actor)$ **do**
    $absolute\_exe \leftarrow$ `SDEP`$_{actor \leftarrow v.actor}(v.exe)$
    {Translate from absolute execution to relative one}
    $iteration \leftarrow (absolute\_exe - 1) \div sched.$`get_num_reps`$(actor)$
    $exe \leftarrow 1 + (absolute\_exe - 1) \mod sched.$`get_num_reps`$(actor)$
    $u \leftarrow$`get_vertex`$(actor, exe)$
    $e \leftarrow$`new_edge`(u, v)
    $E \leftarrow E + e$
    $T \leftarrow T +$ (`weigth`$(e) \leftarrow (-iteration)$)
  end for
end for
{Add causality dependency edges}
for all $v \in V$ **do**
  if $v.exe > 1$ **then**
    {Edges within one CSDF iteration have weight 0}
    $u \leftarrow$ `get_vertex`$(v.actor, v.exe + 1)$
    $e \leftarrow$ `new_edge`(v, u)
    $E \leftarrow E + e$
    $T \leftarrow T +$ (`weigth`$(e) \leftarrow 0$)
  else
    {Edges to previous CSDF iterations have weight 1}
    $u \leftarrow$ `get_vertex`$(v.actor, sched.$`get_num_executions`$(actor))$
    $e \leftarrow$ `new_edge`(v, u)
    $E \leftarrow E + e$
    $T \leftarrow T +$ (`weigth`$(e) \leftarrow 1$)
  end if
end for
{Add control dependency edges}
for all $actor$ **do**
  if `send_teleport_msg`$(actor)$ **then**
    for $exe = 1 \rightarrow sched.$`get_num_reps`$(actor)$ **do**
      for all $recv \in$ `get_teleport_receivers`$(actor)$ **do**
        {Get minimal teleport message latency}
        $k_1 \leftarrow$ `get_min_latency`$(actor, recv)$
        {Determine relative iterations and}
        {relative executions of teleport receivers}
        $(i_r, e_r) \leftarrow$ `compute_rel_iter_exe`$(actor, recv, exe, k_1)$
        $s \leftarrow$ `get_vertex`$(actor, exe)$
        $r \leftarrow$ `get_vertex`$(recv, e_r)$
        $e \leftarrow$ `new_edge`(s, r)
        $E \leftarrow E + e$
        {Relative iteration of each receiver}
        {is the edge weight}
        $T \leftarrow T +$ (`weigth`$(e) \leftarrow i_r$)
      end for
    end for
  end if
end for

Suppose that a teleport sender S at its absolute n^{th} execution is sending a message to a receiver R with minimum latency k_1. Then the absolute execution of the receiver R, m^{th} will be computed as in Section 3.2.1. We then use the conversion from absolute executions of actors to relative executions and iterations as in Section 3.2. Suppose that S and R execute $s = |\mathcal{S} \wedge S|$ and $r = |\mathcal{S} \wedge R|$ times within one iteration of a stream graph respectively, then the relative executions and iterations of S and R for their exections n^{th} and m^{th} is computed as follows:

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

then $i_r = i^S - i^R$ and $e_r = r^R$.

5.1 Uniqueness of Contructed Static Graph

LEMMA 2. *i_r is the same for all i^S.*

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

- If R is downstream of S. Note that CSDF is periodic, therefore, if `SDEP`$_{S \leftarrow R}(m) = n + k_1$ then `SDEP`$_{S \leftarrow R}(m + r * j) = n + s * j + k_1$ based on Equation 3. Thus $i_r = ((n + s * j) \div s) - ((m + r * j) \div r) = (i^S + j) - (i^R + j) = i^S - i^R$.
- If R is upstream of S then $m =$ `SDEP`$_{R \leftarrow S}(n + k_1)$. As CSDF is periodic, therefore, $m + r * j =$ `SDEP`$_{R \leftarrow S}(n + s * j + k_1)$ based on Equation 3. Thus, $i_r = ((n + s * j) \div s) - ((m + r * j) \div r) = (i^S + j) - (i^R + j) = i^S - i^R$.

Lemma 2 shows us a method to compute the relative iteration i_r. As i_r is the same for all i^S, therefore, we could take an arbitrary i^S that is large enough just that we could find the m^{th} execution of R from the n^{th} execution of S where $n = i^S * s + r^S$ as in Section 3.2.1. Based on founded m, we could can calculate i^R and r^R as shown above, subsequently, we could find i_r and e_r.

THEOREM 2. *The constructed static graph from a CSDF stream graph is unique.*

Proof: As all steps in Algorithm 1 is deterministic, therefore the result static graph is unique for one CSDF stream graph.

6. Finding Execution Schedule

We have shown our technique for checking for circular dependencies given a CSDF stream graph without showing how to find a schedule of actor executions under the constraints imposed by latencies of teleport communications.

6.1 Auto-discovered Schedule

However, we realize that even in the presence of TM overlappings, the current scheduling method in StreamIt will still work without being changed. Basically, the circular dependency checking shows the existence of at least one execution order that satisfies latency constraints of TMs implemented in StreamIt based on a pull scheduling algorithm.

6.2 Finding Schedule with Topological Sort of Actor Executions

Although a schedule could be discovered automatically when actors run, it is beneficial to find precomputed schedules as it might help optimize program performance.

One might thought that we already have execution dependency graphs, we can run traditional topological sort algorithms on the

graphs to obtain a schedule. However, the execution dependency graphs are infinite, therefore, a naive sorting approach will not work as it takes forever to finish. Instead, we need a systematic numbering method that assigns values to vertices based on their indices and structures of static graph.

We employ the topological sort method for acyclic dynamic periodic graphs described in Section 3.8 in [17]. For an 1-dimensional acyclic periodic graph G^∞, we could calculate a value $A(v^p)$ for each vertex v^p in the periodic graph such that if there is a vertex $v^p \prec u^l$ in the periodic graph, then $A(u^l) < A(v^p)$. The calculation is done first by constructing a static graph $G = (V, E, T)$ from G^∞ and then solving the following linear program:

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

where T_{uv} is the weight of edge $(u, v) \in E$.

The above linear program has a unique optimal solution for an acylic periodic graph [17], let's call it $(\sigma^*, \pi^*, \gamma^*)$. Then the value assignment procedure for each vertex v^p is follows:

$$A(v^p) = \pi_v^* - \gamma^* p \quad \forall v \in V \tag{4}$$

7. Experiment

We have implemented the algorithm as a StreamIt backend that is capable of checking for circular dependencies and topologically sorting actor executions based on the algorithms we presented in this paper.

Our circular dependency checking algorithm could correctly detect invalid sets of TMs as in the examples we presented with the same static dependency graphs generated.

For the example without any circular dependencies when latency for TMs from E to A is -4 and for TMs from B to D is -1 as in Figure 6(b), the topological sorting algorithm using linear programming find $\gamma^* = 8$ and $\pi^*_{E_5} = 0, \pi^*_{E_4} = 1, \pi^*_{E_3} = 2, \pi^*_{D_2} = 3, \pi^*_{E_2} = 5, \pi^*_{E_1} = 6, \pi^*_{D_1} = 7, \pi^*_{B_3} = 8, \pi^*_{C_2} = 8, \pi^*_{C_1} = 11, \pi^*_{A_3} = 9, \pi^*_{B_2} = 9, \pi^*_{A_2} = 12, \pi^*_{B_1} = 12, \pi^*_{A_1} = 13$. Based on those obtained values, we could find an execution order of actors for the graph that does not violate dependency constraints as follows: $(A_1^n, E_2^{n-1}) \prec (A_2^n, B_1^n) \prec (D_2^{n-1}, C_1^n) \prec (E_3^{n-1}) \prec (E_4^{n-1}, A_3^n, B_2^n) \prec (E_5^{n-1}, C_2^n, B_3^n) \prec (D_1^n) \prec (E_1^n)$.

8. Related Work

Thies et al. [30] introduce TM as a mechanism to relax the rigidness of CSDF. They present an analysis that computes processing time of TMs. However, this analysis is applicable to only non-overlapping TMs. Consequently, CSDF graphs with overlapping TMs cannot utilize this analysis. We address this limitation with our dependency analysis method applicable to any CSDF graph with TMs. Furthermore, we show that it is possible to compute schedules for the CSDF graphs with arbitrary TM structures, which is not well-defined in the work by Thies et al. [30].

Our method is closely related to the method by Rao and Kailath [26] for scheduling digital signal processing algorithms on arrays of processors, where they use reduced dependence graphs similar to the periodic graphs, as a proxy for scheduling and mapping computation operators on arrays of processors. However, the work mainly focuses on a class of applications equivalent to homogeneous SDF applications [5]. Our work is for a more general class of signal processing applications with sporadic control messages that require synchronizations between processes.

Zhou and Lee [31] tackle circular dependency analysis using causality interfaces. The primary focus of their work is on dataflow models. For SDF case, they do not account for control messages in their SDF models. Moreover, we also support dependency analysis for CSDF with overlapping TMs.

Horwitz et al. [12] propose a method for interprocedure program slicing by constructing dependency graphs between program statements with data and control dependency edges. The graph construction method is similar to our method in that they construct dependency graphs of statements and use the graphs to find dependency between interprocedure statements. However, our work exploits the CSDF domain specific program abstraction semantics to analyse a specific criterion, meanwhile, interprocedure program slicing is more a general approach which can be used in other test, debugging or verifying processes.

The deadlock analysis method for communicating processes by Brook and Roscoe [4] characterizes the properties and structures of networks of communicating processes that can cause deadlocks, for example, which kind of communication could cause the dining philosophers problem. However, the method only works for some network of processes structures.

There are several works on deadlock detection in distributed systems [6, 27]. However, the algorithms proposed in those works mainly focus on detecting deadlocks when they happens rather than on deadlock avoidance and static deadlock analysis.

Wang et al. propose a method to avoid potential deadlocks in multithreaded programs when sharing resources. The method translates control flow graphs into Petri nets and uses Petri net theories to find potential deadlocks. Control logic code is then synthesized to avoid potential deadlocks. This bears some similarity to our construction of dependency graphs to detect for deadlocks. If there is no deadlock, topological sorting of actors is done to find a suitable order of executions.

Finally, as an extension of TMG in the StreamIt compiler, our work is based on SDF/CSDF [3, 19] semantics. We also adopt several results from dynamic/periodic infinite graphs [13, 16, 17, 24] and apply them to the TM circular checking and actor execution sorting problems.

9. Conclusion

In this paper, we have introduced a method of checking for invalid sets of specifications in the StreamIt language by exploiting its periodic nature. In fact, the method is applicable to other scheduling problems that have periodic dependencies.

We have implemented the method as a backend of the StreamIt compiler, however, we have not finished code generation phase for StreamIt applications by the time the paper is written. Our future work would be implementing code generation and evaluate the effectiveness of TMG as well as doing research on applications having overlapping of TMs. Furthermore, in the StreamIt compiler's backend, actors involved in TMG are not clustered because fused actors could not be identified and fusing actors could cause false dependencies, therefore, generated code could be not efficient. A modular code generation method such as the one proposed by Lublinerman et al. in [20] could avoid the problem of false dependencies caused by fusing actors.

As TMG's semantics are targeted the CSDF model of computation, it would be interesting to extend the work to the MDSDF model of computation for image processing applications as in [28] as MDSDF is a more natural way to express and compute image and video processing applications.

[5] Actors in homogeneous SDF applications only produce/consume one data token at each output/input ports whenever they execute.

References

[1] Shail Aditya, Arvind, Jan-Willem Maessen, Lennart Augustsson, and Rishiyur S. Nikhil. Semantics of ph: A parallel dialect of haskell. In *In proceedings from the Haskell workshop (at FPCA 95*, pages 35–49, 1995.

[2] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of dsp systems. In *Proceedings of the Acoustics, Speech, and Signal Processing, 2000. on IEEE International Conference - Volume 06*, pages 3362–3365, Washington, DC, USA, 2000. IEEE Computer Society.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397 –408, feb 1996.

[4] S. Brookes and A. W. Roscoe. *Deadlock analysis in networks of communicating processes*, pages 305–323. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

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

[6] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1:144–156, May 1983.

[7] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: a dsl approach to specifying streaming applications. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 1–17, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[8] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 151–162, New York, NY, USA, 2006. ACM.

[9] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 291–303, New York, NY, USA, 2002. ACM.

[10] Amir H. Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. MacroSS: macro-SIMDization of streaming applications. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 285–296, New York, NY, USA, 2010. ACM.

[11] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: portable stream programming on graphics engines. *SIGPLAN Not.*, 46:381–392, March 2011.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[13] K. Iwano and K. Steiglitz. Testing for cycles in infinite graphs with periodic structure. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 46–55, New York, NY, USA, 1987. ACM.

[14] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[15] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36:54–62, August 2003.

[16] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14:563–590, July 1967.

[17] Muralidharan S. Kodialam. *The O-D shortest path problem and connectivity problems on periodic graphs*. PhD thesis, Sloan School of Management, Massachusetts Institute of Technology, 1992.

[18] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear analysis and optimization of stream programs. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 12–25, New York, NY, USA, 2003. ACM.

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

[20] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 78–89, New York, NY, USA, 2009. ACM.

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

[22] P.K. Murthy and E.A. Lee. Multidimensional synchronous dataflow. *Signal Processing, IEEE Transactions on*, 50(8):2064 –2079, aug 2002.

[23] Praveen K. Murthy and Shuvra S. Bhattacharyya. Buffer merging - a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transaction of Design Automation Electronic System.*, 9:212–237, April 2004.

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

[25] Keshab K. Parhi and David G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. Comput.*, 40:178–195, February 1991.

[26] S.K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proceedings of the IEEE*, 76(3):259 –269, mar 1988.

[27] P. Rontogiannis, G. Pavlides, and A. Levy. Distributed algorithm for communication deadlock detection. *Information Software Technology*, 33:483–488, September 1991.

[28] Mainak Sen, Shuvra S. Bhattacharyya, Tiehan Lv, and Wayne Wolf. Modeling image processing systems with homogeneous parameterized dataflow graphs. In *in Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 05*, pages 133–136, 2005.

[29] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr 2002.

[30] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 224–235, New York, NY, USA, 2005. ACM.

[31] Ye Zhou and Edward A. Lee. A causality interface for deadlock analysis in dataflow. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, EMSOFT '06, pages 44–52, New York, NY, USA, 2006. ACM.