Introduction

The growing complexity of modern signal processing applications along with their dynamic nature requires a generalized framework for efficiently modeling and scheduling such applications. We have developed the core functional dataflow (CFDF) model of computation that facilitates natural description of actors and provides a general framework for specifying static and dynamic dataflow applications. A variety of commonly used, existing dataflow models can be naturally represented in CFDF model thus allowing rapid prototyping for heterogeneous applications.

We have extended our MocGraph package, a Java-based package supporting graph-theoretic analysis for models of computation (mosc) that has evolved from the graph package in Ptolemy II, to support trees and generalized schedule trees. A GST is a data structure for representing the execution structures of dataflow graphs employing a wide variety of dataflow models of computation. The CFDF model, along with the features of GST representations in MocGraph package, has enabled us to support simulations in the DIF package that can be used for verification and analysis of static and dynamic dataflow models.

We have demonstrated the effectiveness of our recently developed scheduling techniques for heterogeneous dataflow applications modeled as CFDF graphs that exploit stastically known mode transition patterns in dynamic dataflow actors.

Core Functional Dataflow

Core functional dataflow is a deterministic form of enable-invocate dataflow (EIDF) [1]. EIDF is a generalized dataflow model which allows natural description of actors for static and dynamic dataflow models.

Every actor is specified using a set of nodes that capture the dataflow and functional behavior of the actor along with two methods enable and invoke. The enable method checks if sufficient number of tokens are available on all of the actor inputs so that it can be fired in a given mode.

The invoke method models the functionality of the actor. It is called only if the actor is enabled in that mode. It consumes specified number of tokens available in the corresponding inputs, produces required tokens required on the outputs, and returns the next mode in which the actor must be fired.

The dataflow behavior of an actor for a given mode is static in that number of tokens consumed (produced) by an actor from (on) its inputs (outputs) is fixed for any given mode of the actor.

Generalized Schedule Trees (GSTs)

- Ordered trees with leaf nodes representing actors and internal nodes representing loop counts for the schedule tree rooted at that node.
- Execution involves traversing the GST in depth-first manner and repeating a firing sequence for the number of times denoted by the loop count.
- Unguarded execution involves firing an actor every time the GST traversal algorithm reaches that actor and hence, the scheduler must ensure that the actor is fireable.
- Guarded execution of an actor implies calling invoke method of the actor only if its enable method returns true for the given mode.
- GST representation is independent of the underlying dataflow model and hence can be used to represent schedules for a variety of known dataflow models. In particular, the CFDF model has allowed us to support functional simulations inside the DIF package (Functional DIF, based on the dataflow interchange format (DIF) language).

Generalized Scheduling Strategy

- We exploit the fact that for a given mode, each CFDF actor behaves like an SDF actor. We exploit the underlying static behavior of the CFDF actor to decompose the graph into multiple SDF graphs.
- These graphs are then scheduled using the known SDF scheduling strategies. These SDF schedules are then merged into a single schedule (as shown in Fig. 2(b)).
- The simulator switches between these schedules depending upon the current mode of the CFDF actor.
- Though the number of SDF graphs can be in general grow exponentially with the number of modes for each of the CFDF actors, in most of the practical applications, it is possible to eliminate most of the mode-combinations that can never occur and hence limit the number of SDF graphs to a much smaller number.
- It is also possible to explore the mode transition behavior. In most of the graphs it is possible to group certain modes, for which transitions have compile-time predictability e.g. transition from True_mode or False_mode to Control_mode for the Switch actor.
- This fact is then incorporated into the scheduling strategy before decomposing the graph into SDF graphs. This is more efficient in terms of buffer requirements and number of SDF components in the schedule, as shown in Table (2).