A Model-based Synthesis Framework for the Execution of Dynamic Dataflow Actors

Omair Rafique and Klaus Schneider
Department of Computer Science
University of Kaiserslautern
Kaiserslautern, Germany
{rafique,schneider}@cs.uni-kl.de

Abstract—Dataflow process networks (DPNs) offer a convenient model of computation (MoC) for modeling parallel behaviors. However, finally synthesizing them to efficient implementations on heterogeneous hardware platforms is still a difficult task. The open computing language (OpenCL) emerged as a common hardware abstraction for programming heterogeneous hardware devices supported by many hardware vendors. In this paper, we present a model-based synthesis framework which logically incorporates OpenCL as an operating system (OS) for synthesizing DPNs to software for parallel heterogeneous implementations. In general, the state of the art frameworks for DPN synthesis incorporate static analysis and scheduling, but are limited to static DPNs, and they only allow static mappings from models to platforms. In contrast, our framework employs a more generalized data dependent and actor-based dataflow model, and allows dynamic mappings from actors to platforms. We demonstrate by simple but concrete dynamic dataflow actors that the proposed framework is capable of handling efficiently dynamic token rates and dynamic data paths at runtime.

Index Terms—model-based design, heterogeneous synthesis, parallel computing languages

I. INTRODUCTION

A heterogeneous computing platform can accommodate different devices like single/multi-core processors, graphical processing units (GPUs), digital signal processors (DSPs), hardware accelerators like FPGAs, and application-specific microprocessors (ASIPs), all interconnected together on the same platform. At the level of its software architecture, it may consist of many components that execute and communicate with each other on these devices based on particular models of computation (MoCs) [1]–[4]. To develop such complex systems, model-based design frameworks have been introduced: A model-based design is generally characterized with a hardware-agnostic abstract model based on a particular MoC and is equipped with a tool chain typically providing simulators, tools for verification, code generators, and tools for system and communication synthesis.

Apart from model-based design, a considerable effort has been invested in introducing programming languages [5]–[7] for the abstraction and better resource utilization of heterogeneous architectures. Among them, the open computing language (OpenCL) has found a lot of interest in heterogeneous computing and is supported by the leading hardware vendors including Intel, Apple, AMD, and many others. In contrast to proprietary specification languages with limited hardware choices, OpenCL allows task-parallel and data parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs [7]. Fortunately, OpenCL is an abstract specification that does not impose any specific MoC, and can therefore be used for implementing low-level details of a particular MoC, but also of different MoCs.

The dataflow MoC [8], [9] defines a system as a network of autonomous process nodes that communicate with each other via unidirectional FIFO buffers. The autonomy provided by this MoC allows distributed computations, and hence innately qualifies for modeling parallel behaviors. The existing model-based design frameworks that target heterogeneous architectures are based on different subsets of the dataflow MoC [10], [11]. We appreciate the convenient use of the existing frameworks, but we also address one of the major limitations of their application: Almost all the existing frameworks employ static analysis and scheduling, and hence are limited to applications modeled with static (synchronous) dataflow (SDF) MoC or even fixed data rate actors. To further extend the flexibility and improve expressiveness, and to support dynamic dataflow behaviors, a framework for more generalized dataflow MoCs with dynamic data rates and dynamic data paths is needed.

To this end, we present a model-based synthesis framework capable of synthesizing a more generalized dataflow MoC with concrete dynamic actors exhibiting dynamic data rates and data paths to corresponding heterogeneous implementations. The proposed framework conceptually employs OpenCL as an operating system (OS) for implementing low-level details i.e., task scheduling, resource allocation, data communication, etc., based on the underlying MoC. Employing OpenCL does not only allow us to target vendor-neutral heterogeneous architectures, but also provides a standard abstraction for better resource utilization in parallel architectures.

In this paper, we mainly present the complete design flow that realizes a generalized dataflow MoC, capable of modeling dynamic behaviors, and automatically synthesizes them to parallel implementations. We demonstrate the ability of the framework to capture dynamic behaviors by first introducing a set of concrete dynamic actors that are developed especially for reflecting dynamic data rates and data paths, followed by an experiment constructed using those actors.
II. THE PROPOSED FRAMEWORK

The proposed framework is systematically arranged in two stages of modeling and synthesis, as shown in Fig. 2: First, the modeling stage allows the application designer to model a system with a generalized dataflow model, capable of capturing dynamic dataflow behaviors. Finally, the synthesis stage produces and executes the corresponding implementation of the modeled behavior based on the underlying semantics, on the targeted OpenCL-abstracted heterogeneous hardware.

A. Modeling

Model-based design flows are based on models of computation (MoCs) that precisely determine why, when and which atomic action of a system is executed. Based on that, the dataflow MoC organizes the behavior of a system as a network of process nodes (actors) that communicate via unidirectional FIFO buffers, where each actor autonomously monitors its input buffer and may fire if there is sufficient data available. However, various subsets of dataflow MoC exist [10], [11] that mainly differ based on the order in which actors are executed, i.e., either statically scheduled or data dependent, and data rates with which actors consume/produce data i.e., either fixed or data dependent. The existing frameworks based on dataflow MoCs generally incorporate static scheduling, and/or essentially rely on fixed data rates, thus do not support dynamic behaviors. In contrast, our proposed framework employs a data dependent dataflow MoC that allows us to model more flexible and expressive actors to capture dynamic behaviors.

1) Formal Description of the Employed MoC: A network is a directed graph, i.e., an ordered 4-tuple $N := (A, F, s, t)$ where:

- $A$ is a finite set of actors.
- $F$ is a finite set of bounded FIFO buffers.
- $s : F \rightarrow A$, assigning to each FIFO buffer its source actor.
- $t : F \rightarrow A$, assigning to each FIFO buffer its target actor.

An actor $a \in A$ is a 3-tuple $(F_{in}, F_{out}, G_{A})$ as shown in Fig. 1, where:

- $(F_{in} \subseteq F) \land (F_{out} \subseteq F)$.
- $G_{A}$ (guarded-actions) is a finite set of 2-tuple $g_{a} = (\gamma, \tau)$.

$\tau$ is an action, i.e., a 3-tuple $(F_{i}, F_{o}, R)$ where:

- $F_{i} \subseteq (F_{in} \times C_{in})$.
- $F_{o} \subseteq (F_{out} \times C_{out})$.
- $R$ is a function of the form $f : \mathbb{R} \rightarrow \mathbb{R}$.
- $C_{in}$ is a finite set of input token count per action of an actor.
- $C_{out}$ is a finite set of output token count per action of an actor.

$\gamma$ is a guard, i.e., a 2-tuple $(F_{g}, R_{g})$ where:

- $F_{g} \subseteq (F_{i})$.
- $R_{g}$ is a Boolean function of the form $f : F_{g} \rightarrow B$, where $B = \{0, 1\}$.

2) Execution Semantics: Each actor $(\in A)$ in $N$ may execute whenever there is input data available for any action $\tau$ and if space is available for the outputs of that action. The data rate and the data path of an actor can change per execution depending on which $g_{a}$ (guarded-action) is executed. For each $g_{a} (\in G_{A})$ of an actor, $F_{i}$ is peeked, $F_{o}$ is checked for space, and $\gamma$ (guard) is evaluated using the function $R_{g}$. In case if the guard holds, the corresponding $\tau$ is executed which consumes the peeked $F_{i}$, performs a computation $R$, and finally produces output $F_{o}$ of that $\tau$. For any non-guarded $\tau$, $\gamma$ is automatically evaluated as true.

Each actor is permitted to check $F_{in}$ by peeking before it can be finally consumed, thus allows a non-blocking read. This makes the proposed MoC more flexible, however may lead to non-deterministic behaviors e.g., a non-determinate merge [9]. In contrast to the existing frameworks, the proposed framework does not consider special types of actors, instead, an abstract notion of an actor is used with input buffers, guarded actions and output buffers as shown in Fig. 1 to realize a generalized data dependent MoC.

B. Synthesis

The synthesis stage integrates a number of different components as shown in Fig 2. These components contribute together to finally implement abstract models as supplied by the modeling stage in accordance with the semantics of the employed MoC. As discussed, the framework logically incorporates the OpenCL specification as an operating system (OS) mainly because of two reasons: first, it provides an abstraction for heterogeneous hardware, and second, the framework uses this abstraction in the composition of the synthesis where different components implement different low-level details. To this end, OpenCL distinguishes between a host and kernels where the host is a centralized entity that is connected to one or more compute devices (CPU, GPU etc.) and is responsible for the execution of kernels. Kernels are C-like functions that actually implement the abstract behavior of the system or part of the system. Therefore, the proposed framework adopts this idea of hosts and kernels for the implementation of the synthesis stage.

1) Code Generator: The code generator is a core component of the framework that generates code strictly based on the semantics of the employed MoC and the OpenCL
specification. To this end, each actor is modeled based on the description presented in Section II-A1 using a subset of the CAL actor language [12]. Based on the OpenCL paradigm, the code generator supplied with abstract models generates: an OpenCL kernel for each actor of the network, and a queue of actor objects denoted as \( \text{Actors-Queue} \). Each kernel is generated in the form of a 3-tuple \((F_{\text{in}}, F_{\text{out}}, G_A)\) as shown in Fig. 1, based on the described semantics. The \( \text{Actors-Queue} \) is a component of the centralized host and contains a special object for each actor. Each object provides the desired information about each actor to the host such as, the associated FIFO buffers, the actor’s status (idle, running or blocked), the associated kernel, etc. The queue, once generated, is then maintained and updated by the host.

2) Centralized Host: The centralized host of the framework is further composed of essential components that work together for implementing low-level details such as the scheduling policy, the communication mechanism, resource allocation, etc. To this end, apart from the \( \text{Actors-Queue} \) that is provided by the code generator, the host creates a \( \text{Device-Queue} \) using the OpenCL specification that lists all the available devices of the target hardware. Each element of this queue provides a command queue of a device, where the actors can be mapped for execution as shown in Fig. 3. Each command queue can represent a complete device (e.g., a CPU) or even a compute unit of that device (e.g., a CPU-core).

The overall implementation of the host revolves around the Runtime-Manager, as shown in Fig. 2. The Runtime-Manager is a part of the host that exploits different components and provides: the scheduler for scheduling actors, the communication mechanism between the host and kernels, a dispatcher for mapping actors to devices and the status update mechanism using callbacks.

To this end, the scheduler based on the used MoC iterates through the list as provided by the Actors-Queue in a round robin fashion, and fetches an actor from the list if there is data available for the input and space available for the output for at least one action \( \tau \). Next, the scheduler examines the Device-Queue and finds the device with the least weight (i.e., a device currently assigned with least number of actors), and dispatches the fetched actor on that device. Altogether, the scheduler employs a simple weighted-round-robin strategy that is based on the current load of devices, in terms of assigned actors. The corresponding kernel of the dispatched actor is then executed based on the semantics as described in Section II-A1. Depending on real-time values of the available data, different guarded actions \((\in G_A)\) can be invoked on each execution, resulting in dynamic data rates and data paths. Thus, following a data dependent dataflow MoC, each execution of
an actor may consume/produce a different number of tokens, and therefore allows the implementation of dynamic behaviors.

OpenCL allows data-level parallelism, capable of executing multiple actor executions in parallel. Depending on the availability of data, multiple executions of the actor can be scheduled in parallel. When all the instances of the kernel are executed, i.e., the dispatched actor executions are completed, the Runtime-Manager should be notified to update the components. For that purpose, a status update mechanism is developed using callbacks as shown in Fig. 3. The Runtime-Manager generates a callback each, for every existing device in the Device-Queue during the initialization of the queue. The scheduler sets up a callback event for each fetched actor and links it with the callback handler of the device where it is dispatched. Hence, the completion of the kernel of the dispatched actor, automatically invokes the callback handler of the used device. The callback handler performs a set of tasks including: retrieving data from the kernel (OpenCL buffers), updating all the FIFO buffers of the actor, updating the status of the actor, updating the Actors-Queue, updating the device’s load, and finally updating the OpenCL buffers. To maintain consistency, the data communication between the host and kernels is realized using OpenCL buffers with the explicit Read/Write API, even if the host and kernels reside on the same device.

III. EXPERIMENTAL RESULTS
This section first introduces a set of concrete dynamic actors, followed by an experiment to validate the ability of the framework to handle dynamic behaviors.

A. Dynamic Dataflow Actors
The main focus of the presented framework is to realize a model-based synthesis based on a data dependent dataflow MoC that allows modeling dynamic behaviors and synthesizes them to heterogeneous implementations. To validate the functionality of the framework to implement dynamic behaviors, a set of concrete dynamic dataflow actors as shown in Fig. 4 are especially modeled and synthesized with the proposed framework.

The Switch and the Select actors are the basic classical examples of dynamic actors. The Switch actor as depicted in Fig. 4 has four guarded actions ($G_A$). Based on the execution semantics as described in Section II-A1, for each $g_a \in G_A$, input FIFO buffers ($I, A \in F_i$) are peeked, the output FIFO buffer ($(O1|O2|O3|O4) \in F_o$) is checked for space, and the guard $\gamma$ on $I(\in F_g \subseteq F_i)$ is evaluated. Depending on the invocation of a particular $\gamma$, the corresponding action $\tau$ is executed which consumes the peeked $F_i$ and finally maps a token from buffer $A$ to $F_o$. This allows a dynamic data path behavior, where depending on a particular successful invocation of a $\gamma$ on $I$, a token from $A$ is mapped to one of the four output buffers.

The Select actor is modeled with a set of three guarded actions, and in contrast to Switch, based on the invoked $\gamma$ on $I(\in F_g \subseteq F_i)$, a token from the input FIFO buffer ($(A|B|C) \in F_i$) is mapped to the output FIFO buffer ($O \in F_o$). This actor also practices a dynamic data path behavior.

The next in the list is a simple dynamic worker (d_worker) actor as shown in Fig. 4 which is especially modeled and synthesized with three guarded actions to validate the behavior with dynamic data rates. For each guarded action $g_a$, the only input FIFO buffer $A(\in F_g \subseteq F_i)$ is peeked, the only output FIFO buffer ($O \in F_o$) is checked for space, and the $\gamma$ on $A$ is evaluated. Depending on the successful invocation of a particular $\gamma$, the corresponding $\tau$ is executed which consumes the peeked $F_i$ and finally maps data based on the token count per invoked action ($\in C_{out}$) from buffer $A$ to $F_o$. This allows producing/consuming data with dynamic token rates per execution of the d_worker actor.

The d_split and d_merge actors are special cases of the Switch and the Select actor, respectively. While the latter only practice dynamic data paths, the former extend them to indulge dynamic data rates. Similar to the Switch actor, the d_split actor has four guarded actions ($G_A$). However, to support changing data rates per execution, upon the successful invocation of a particular $\gamma$, $\tau$ is executed which consumes the peeked $F_i$ and finally maps data based on the token count per invoked action ($\in C_{out}$) from buffer $A$ to a set of output FIFO buffers ($(O1)(O1|O2)(O1&O2|O3)(O1&O2&O3&O4)) \in F_o$. This allows us to capture both the dynamic data path and the dynamic data rate per execution.

The d_merge actor is modeled with a set of three guarded actions, and in contrast to d_split, based on the invoked $\gamma$ on $I(\in F_g \subseteq F_i)$, a token each from a set of input FIFO buffers ($(A)(A&B)(A&B&C)) \in F_i$ is mapped to the output FIFO buffer ($O \in F_o$). This actor also practices both the dynamic data path and the dynamic data rate per execution.

Fig. 4. A set of dynamic actors
B. Experiment using Dynamic Actors

To demonstrate the ability of the proposed framework to synthesize dynamic behaviors, a sorting network based on the Mergesort algorithm is modeled and synthesized with a set of already presented concrete dynamic actors, as shown in Fig. 5. The Mergesort is a divide-and-conquer algorithm commonly used for sorting data in a particular order with a sequential time complexity of $O(n \log(n))$. It works on the principle of breaking down a set of unsorted data into several subsets until each subset consists of a single data element and finally merging those subsets in a way that converges into a set of sorted data.

The modeled network is composed of 5 actors, and the data is supplied externally by the host (source-actor) from outside the network space. The network accompanies two dynamic split actors (i.e., splitNsort actors) where each split actor splits the supplied unsorted data into subsets of single data element, and writes each data element on the output in the sorted ascending order. The dynamic merge actor (i.e., the mergeNsort actor) uses the Mergesort algorithm and finally merges two individual subsets of sorted data, as provided by the split actors, in a way that results into a complete set of sorted data. Since, the network employs two splitNsort actors, the predecessor actor i.e., the divide_data divides the source data as provided by the source actor into two halves, and feeds each split actor with a half. This allows sorting subsets of unsorted data in parallel. Moreover, the dynamicity offered by dynamic split and dynamic merge actors allows sorting any set of data ranging from two to eight data elements. Therefore, the source actor provides different sets of data with random sizes ranging from two to eight elements.

As discussed, this paper mainly emphasizes on presenting a synthesis methodology based on a data dependent dataflow MoC that allows modeling and synthesizing dynamic behaviors. To this end, the scheduling of actors as well as the consumption/production of data is based on real-time values of data and is decided at runtime. Therefore, to evaluate the framework overhead including the scheduling overhead, the communication, resource allocation and the status update, the experimental results of the sorting network are compared with a reference C++ implementation, as shown in Fig. 6.

As depicted in Fig. 6, the execution time of the complete network is recorded based on different number of samples, where each sample provides a data set. Based on that, the experimental results suggest a small execution overhead in comparison with a pure C++ implementation, even without exploiting any parallelism and with all the implementation overheads of the framework.

IV. RELATED WORK

The related work is observed from two main aspects as given in the following sections.

A. Model-based Design Frameworks without OpenCL

Model-based design methodologies in the related state-of-the-art mainly differ by their employed MoCs. In [13], the HW/SW co-design methodology based on the CAL actor programming language is built as an Eclipse plug-in on top of ORCC\(^1\) and OpenForge\(^2\). The synthesis provided by this framework is limited to multicore CPUs and FPGAs, where the user has to explicitly provide the specification of the target hardware.

The most popular and commercially recognized model-based design tool Matlab\(^3\) has introduced a variety of supporting toolkits over time. An interesting approach is presented in [14] where the Matlab code is transformed into KPN specification using the Compaan compiler. The HW backend Laura is used to map this KPN specification into hardware. The methodology is limited for the synthesis of application specific hardware.

Apart from these frameworks that support homogeneous modeling, Ptolemy [15] and Ptolemy II [16] support multiple MoCs including different subsets of dataflow MoC. Similarly,

---

\(^1\)http://orcc.sourceforge.net
\(^2\)https://sourceforge.net/projects/openforge
\(^3\)http://www.mathworks.com/matlabcentral/
FERAL [17] adopts and extends some of the concepts from the Ptolemy project. It provides a holistic model-based design approach to enable the coupling of specialized simulators in offline scenarios, i.e., without connecting them to real hardware. Both Ptolemy and FERAL are modeling and simulation frameworks, where the basic aim is to study different MoCs and even simulate systems modeled with the combination of different MoCs.

B. Model-based Design Frameworks with OpenCL

The framework presented in [18] introduces a design flow for executing applications specified as synchronous dataflow (SDF) graphs on heterogeneous systems using OpenCL. The main focus of this work is to develop and provide features and concepts to better utilize the parallelism and thereby improving end-to-end throughput in heterogeneous architectures.

The work presented in [19] provides an approach to translate behaviors modeled with a subset of dataflow MoC into programs running some of the computations on OpenCL platform. The methodology incorporates static analysis and transformations and thus confined to modeling static behaviors. Another approach presented in [20], [21] provides a dataflow programming framework not restricted to the SDF MoC only. The proposed MoC is described as a symmetric-rate dataflow, a restricted form of SDF, where the token production rate and the token consumption rate per FIFO channel is symmetrical. The individual actors considered are essentially described with fixed data rates. Therefore, the existing frameworks based on dataflow MoC and OpenCL, generally incorporate static scheduling, and/or essentially limited to actors with fixed data rates, thus do not support dynamic actors. In contrast, our proposed framework employs a data dependent dataflow MoC that allows us to model more flexible and expressive actors, capable of synthesizing dynamic behaviors.

V. CONCLUSION AND FUTURE WORK

The existing model-based design frameworks based on dataflow MoC and OpenCL, are generally limited to static analysis and scheduling, and/or essentially support actors with fixed data rates. In this paper, we present a framework that employs a generalized data dependent dataflow MoC which is capable of modeling and synthesizing actors with dynamic data rates and dynamic data paths. To this end, the semantics of the proposed MoC are formally described and the complete design flow of the framework is elaborated. The ability of the framework to capture dynamic behaviors is demonstrated by first introducing a set of concrete dynamic actors that are synthesized with dynamic data rates and dynamic data paths, followed by an experiment that actually used those actors for constructing a dynamic dataflow network. The experimental results suggest that the framework overhead is not significant even without exploiting any parallelism.

In the future, we plan to utilize the employed OpenCL abstraction for the substantial performance acceleration in heterogeneous architectures. Also, we intend to extend the framework to enable the modeling and synthesis of systems with heterogeneous combination of MoCs.

REFERENCES