SDF to Synchronous Cross Domain Analysis in ForSyDe Stream Processing Framework

Jun Zhu, Axel Jantsch, and Ingo Sander
ECS/ICT, Royal Institute of Technology, Stockholm, Sweden
{junz, axel, ingo}@kth.se

Abstract
Stream processing has been a very active field in parallel programming for its suitability to express the concurrent architecture in embedded systems. Caused by its concurrent reasoning features, stream programming frameworks are built on some abstract models of computation (MoCs) to handle the complexity and unpredictability. To allow us focus on the essential issues of time, communication and synchronisation of the parallel tasks, the support from a sound heterogeneous MoCs framework to stream application system is still in need. ForSyDe is our high level executable design framework to express multi-computational-models, based on stream processing concept. It is a heterogeneous diagram to describe intricate application behaviors, and offers cross domain analysis features to support multi-domains integration and optimization. A case study in ForSyDe framework shows that the communication structure of a stream application in SDF domain could be migrated to the synchronous domain without any extra work on its computation functions. To integrate it with our work on a communication based NoC simulator, we believe some more interesting design exploration work could be done on the analysis of communication and computation efforts, besides power issues.

1. Introduction

With the prosperous emergence of multi-core processors and multi-processor systems-on-chip (SoCs), the 'free performance lunch era' [1] for conventional sequential programming in embedded system design has come to the end. To be able to design the incoming heterogeneous parallel systems with high reliability, predictability, low cost, high performance, besides extreme power efficiency, the theory and foundations of traditional design need to be advanced significantly. This underscores the pressing demand for improvements in programming models which could not only efficiently map to the complex parallel structure, but also be able to implement the applications with disciplined methods to achieve guaranteed real-time properties.

Stream programming, with exposed communication structure, are an extremely promising paradigm to allow parallel processing. To define how the series of computation kernel functions interact and how time is represented, stream languages are always based on particular MoCs.

2. Related work

StreamIt [2-3] is a high level streaming computation language, with a good matching with the communication dominated NoC-based RAW architecture. It is based on synchronous dataflow (SDF) model, in which data rates of the applications are compile-time static. Although SDF model is suitable to model signal processing system, and efficient in buffer analysis, it ignores the computation or communication time, and could not be used handle the interaction with asynchronous events. To express time dependent system behaviors, N-synchronous programming

language [4] builds on synchronous model, which helps to gain increased control on buffer management in reactive and safe-critical real-time systems.

Instead of using a single given MoC basis in programming model to address the heterogeneity at the implementation level (full custom HW, FPGA, DSP and embedded SW), a sound MoCs framework need to model the heterogeneity at the system level. Ptolemy [5] has been pioneered with the development of the integration of different MoCs to simulate the heterogeneous streaming processing system. However, to model, analyze, and design complex, heterogeneous embedded systems and SoCs, we propose ForSyDe [6-7] framework for heterogeneous MoCs (timed models, synchronous models and untimed models).

Another very active field in parallel programming is skeletal parallel programming [8], which is a framework with functional style primitive skeletons, and aims to facilitate the traditional language programmers to build parallel skeleton programs in a sequential way. In commercial uses, MapReduce [9] model in Google has been very successful, which uses the similar functional style framework to parallelize programs, with the general concepts “map” and “reduce” borrowed from functional languages. It has achieved great performance improvement on clusters of commodity machines, with the support of the scalable Google File System.

3. Heterogeneous ForSyDe framework

ForSyDe is a denotational framework, which classifies different MoCs based on the abstract denotation of timing in the model. It is developed with the objective to move system design to a higher level of abstraction and to bridge the abstraction gap by transformational design refinement. Furthermore, it suites very well to model stream processing applications in different MoCs domains without any extension itself. In this paper, we consider ForSyDe streaming programming in SDF and synchronous domains.

3.1 Streams in MoCs

Stream in ForSyDe is a list of events. We distinguish the MoC by using different kinds of events in streams. Given a set of value V, which represents the data communicated over the streams. SDF events are just values without further information, $\dot{E} = V$. Synchronous events include an extra pseudo value $\perp$ to model absent value, in addition to the normal values, to represent physical time slot; hence $\bar{E} = V \cup \{\perp\}$. In this way, ForSyDe use the similar representation of streams in different domains to maintain the global time in an abstract way. The execution rule is strictly data driven and based on the availability of data.

3.2 Generic process and composition operators

In heterogeneous MoCs, to handle the streams of normal values together with possible timing information, we extend the traditional kernel function in stream processing to generic process (Fig 1). The generic process is a 3-layer structure (only the blocks in shadow are compulsory), which integrates the communication and computation functions. Based on the timing information in the input stream and the corresponding settings in the constructor wrapper, this model could be used to instantiate specified types of processes with different functionalities in multi-domain of MoCs. Now, ForSyDe framework is written in function language Haskell, and with the support of Haskell

foreign function interface (FFI), it could invoke the conventional sequential language, such as C [10].

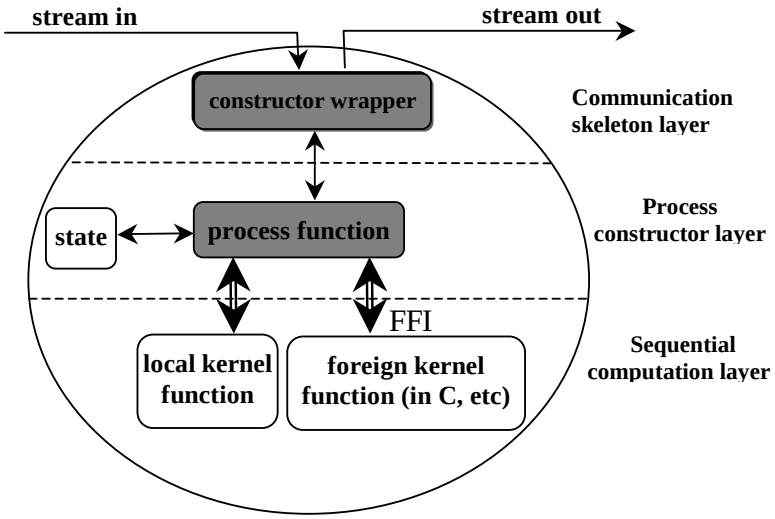


Fig. 1. The 3-layer structure of the generic process

We define only a few basic types of process constructors that could be used to compose more complex processes and process networks. Process instantiated with *map* constructor operates on the input stream in sequence to generate the output stream, and has no internal state. Constructor *mealy* create process resemble Mealy state machines in that they have a next state function and an output encoding function that depends on both the input and the current state. Furthermore, to handle arbitrary input and output processes, we introduce *zip* and *unzip* processes that merge two input streams into one and split one compound input stream into two output streams.

With the hierarchical composition abilities, only three basic composition operators are considered, namely sequential composition ($\circ$), parallel composition ($\|$), and feedback (μ).

3.3 Generic MoC

One of our objectives is to capture the different computational models in a uniform way. From the structure of the generic process, we could see that the processes in ForSyDe are more independent of the MoC, that allows cross domain analysis and optimization become easier.

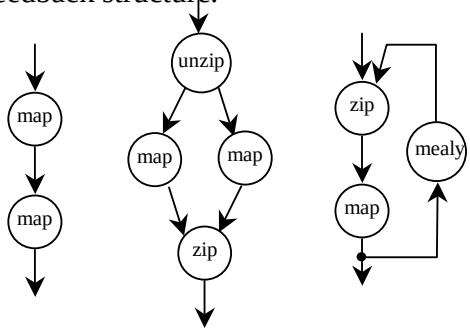
Thus, we define the generic MoC.

Definition A generic MoC is a 2-tuple MoC=(C, O), where C is a set of process constructors instantiated by generic process, each of which, when given constructor specific MoC domain and parameters, instantiates a process. O is a set of process composition operators, each of which, when given processes as arguments, instantiates a new process.

$$\begin{aligned} C &= \{map, mealy, zip, unzip\} \\ O &= \{\|, \circ, \mu\}. \end{aligned} \tag{1}$$

Using this definition and the composition operators, we construct some basic structures with denotable directed graph, where nodes represent communication and computation compound of processes and edges represent data streams, shown in Figure 2.

Especially, the mealy process in the loopback branch is modeled to provide the initial firing token needed by the feedback structure.



a) sequential b) parallel c) feedback loop

Fig. 2. Basic structures in ForSyDe stream processing framework

3.4 Cross domain analysis

Given the interface process p_I between two different domains of MoCs, the definition of moving *map* based process p_{mapSDF} from SDF domain to a function equivalent process p' in synchronous domain is given as:

$$p_I \circ p_{mapSDF} = p' \circ p_I \quad \textit{where} \quad p' = p'_{serialize} \circ p'_{mapSync} \circ p'_{pack} \tag{2}$$

in which, p_{mapSDF} and $p'_{mapSync}$ are the same type of map processes. They could share the same kernel function, but are instantiated from the generic process model in a different synchronous domain; As p_{mapSDF} consumes n $(n \geq 1)$ events during each execution cycle, while $p'_{mapSync}$ consumes only one, p'_{pack} and $p'_{serialize}$, as the names specified, are the processes for packing proper input stream data format for p'_{map} and serializing the output stream from p'_{map} separately. p'_{pack} and $p'_{serialize}$ could both be modelled as *mealy* processes, and are only needed when the corresponding consuming or outputting events are more than one; otherwise, they could be omitted as dummy processes.

In this way, the process cross domain analysis from SDF to synchronous domain could be formalized, but we do not need to modify the behavior of the kernel function, which nicely preserve the separation of parallel communication skeletons from computation functions.

3.5 Process merge

Merging and splitting of processes, without changing the system behavior, could form a well-structured design to map onto the implementation architecture. In addition, it could achieve a significant influence on non-functional properties such as performance, cost, and power consumption. Processes that form a perfect or rational match can be easily merged [6].

$$
\begin{aligned}
map_1 \circ map_2 &= map' \\
map_1 \circ mealy_1 &= mealy' \\
mealy_1 \circ map_1 &= mealy' \\
mealy_1 \circ mealy_2 &= mealy'
\end{aligned}
\qquad (3)
$$

Thus, the synchronous domain p' in (2) could be seen as a merged mealy process:

$$
\begin{aligned}
p' &= p'_{mealySync} \\
where \quad p'_{mealySyn1} &= p'_{mapSync} \circ p'_{pack} \qquad (4) \\
p'_{mealySync} &= p'_{serialize} \circ p'_{mealySync1}
\end{aligned}
$$

4. Application study

We use the FM software radio application as in [2]. The parallel communication structure is sketched out using ForSyDe SDF library, with its denotable graph shown in Fig. 3. The numbers besides each edge stand for the amount of consume/output events of the end side node. The kernel functions are written in ANSI C, which exchange data with the process skeleton through Haskell FFI.

Using the ForSyDe cross domain migration semantics, we could get one alternative of the FM radio designs in synchronous domain, shown in Fig. 4. When the *map* process in SDF domain consumes and outputs only 1 event per evaluation cycle, we will get the same type *map* process in synchronous domain; otherwise, we get a functional equivalent *mealy* process, in which the state function and output function are based on a semantic preserved transformation from the kernel function of the *map* process in SDF domain and the pack/serialize function in ForSyDe library.

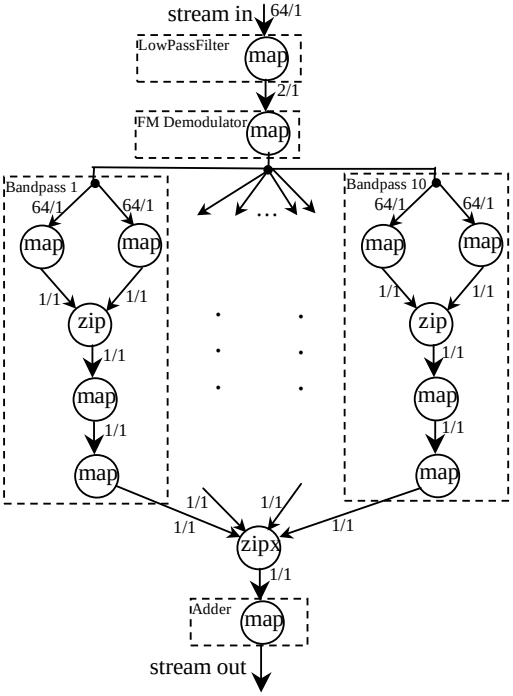


Fig. 3. SDF FM Radio design in ForSyDe

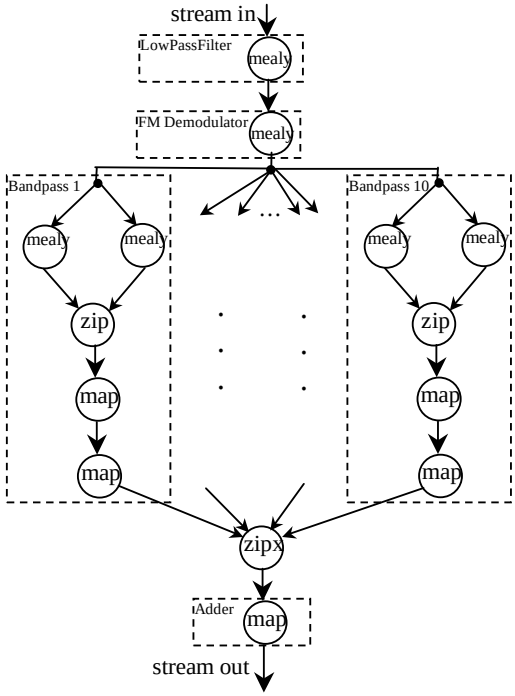


Fig. 4. Synchronous FM Radio design in ForSyDe

Assume each process node corresponds to one computation unit in the communication oriented NoC-architecture, the synchronous executable model of the design provides us a high level design exploration environment to consider non-functional effects of different alternatives. We could move the pack function of the two parallel mealy process in the "*Bandpass*" module up into the "*FM Demodulator*" process to avoid the duplicated data preparation in all "*Bandpass*" branches, merge the "*LowPassFilter*" and "*FM Demodulator*" into one *mealy* process, or merge all the processes in "*Bandpass*"into one, without change the whole system behaviors.

5. Conclusion and future work

In this paper, we have shown that ForSyDe is a very flexible framework to support different domains of stream processing applications. With the integration of heterogeneous MoCs in ForSyDe, it will facilitate the integration of multi-domain stream programming in one framework also.

To address communication cost and power budget redistribution issues in an early system level, we plan to develop a stream processing system with the communication framework based on our Nostrum [11] NoC simulator in the future.

Reference

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, March 2005.

[2] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[3] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[4] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. "N-sychronous Kahn networks," in *33th ACM Symposium on Principles of Programming Languages*, Charleston, SC, pp. 180-193, Jan. 2006.

[5] The Ptolemy Project, http://ptolemy.eecs.berkeley.edu.

[6] A. Jantsch, *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*, Morgan Kaufmann, 2003.

[7] I. Sander, "System Modeling and Design Refinement in ForSyDe," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2003.

[8] The skeletal parallelism homepage, http://homepages.inf.ed.ac.uk/mic/Skeletons.

[9] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, Dec. 2004.

[10] The Haskell 98 Foreign Function Interface 1.0, http://www.cse.unsw.edu.au/~chak/haskell/ffi.

[11] The Nostrum home page, http://www.imit.kth.se/info/FOFU/Nostrum.