Parameterized Dataflow Graphs

Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD, USA
ssb@umd.edu, http://www.ece.umd.edu/~ssb/

With contributions from Bishnupriya Bhattacharya,, Hojin Kee,
William Plishker, Sebastian Puthenpurayil, Sankalita
Saha ,Nimish Sane, and Hsiang-Huang Wu
Outline

• Introduction
• Reconfigurable Dataflow Graphs
• Parameterized Dataflow Meta Model
• Parameterized SDF
• Parameterized CSDF
• Dataflow clustering transformations
• Application Examples
• Conclusions
Transformations in DSP System Implementation

• High-level transformations have been studied in various contexts
  – High-level synthesis
  – Synthesis of DSP software
  – Fault detection in parallel processing systems
  – Transformation of high-level models of DSP applications such as dataflow graph transformations

• Primarily explored in the context of decidable dataflow graphs, …

• especially synchronous dataflow and its important subclasses
Modeling Design Space

Verification / synthesis power
Exploiting Analysis Potential with High Level Dataflow Transformations

- A well designed dataflow representation exposes opportunities for high level algorithm and architecture transformations.
- High level of abstraction $\rightarrow$ high implementation impact
- Dataflow representation is suitable both for behavior-level modeling, structural modeling, and mixed behavior-structure modeling
  - Transformations can be applied to all three types of representations to focus subsequent steps of the design flow on more favorable solutions
- Complementary to advances in
  - Platform-oriented language technologies (intra-actor functionality)
  - Object oriented methods (library management, application service management)
  - HDL synthesis (intra-actor functionality)
Representative Dataflow Analyses and Optimizations

• Bounded memory and deadlock detection: consistency
• Buffer minimization: minimize communication cost
• Multirate loop scheduling: optimize code/data trade-off
• Parallel scheduling and pipeline configuration
• Heterogeneous task mapping and co-synthesis
• Quasi-static scheduling: minimize run-time overhead
• Probabilistic design: adapt system resources and exploit slack
• Data partitioning: exploit parallel data memories
• Vectorization: improve context switching, pipelining
• Synchronization optimization: self-timed implementation
• Clustering of actors into atomic scheduling units
Beyond Decidable Models

• Limited expressive power: DSP applications increasingly employ high-level dynamics in their behavior
  – User interface functionality
  – Mode changes
  – Adaptive algorithms
  – Reconfiguration of processing resources/parameters

• However, key subsystems still exhibit large amounts of “quasi-static” structure --- structure that stays fixed across significant windows of time.

• Various dynamic dataflow models have been proposed that address the limitation above by abandoning most or all restrictions related to decidable dataflow models (DDMs)

• However, these methods are correspondingly limited in their ability to exploit the quasi-static structure described above → i.e., weaker classes of enabled transformations

• Models like SBF and EIDF provide some form of actor level structure in terms of DDMs, but higher level structure must be derived bottom up
Parameterized Dataflow

- A hierarchical meta-modeling technique based on a structured, hierarchical concept of reconfiguration
  - Application parameters
  - Parameter domains
  - Parameter configurations
  - Unspecified parameter values (configured at run-time)
  - Can be applied to architectural and behavioral attributes

- Exposes precise *locality* conditions and constraints for preserving useful subsystem properties throughout well-defined windows of time
Parameterized Dataflow: Structured Control of Dynamic Parameters

\[ \Phi = \text{subsystem}(H) \]

params(\Phi_b)

params(\Phi_s)

params(\Phi_i)

params(\Phi) \text{ set from above}
Parameterized Dataflow Terminology

- Actors, edges, and subsystems can be parameterized, each with any number of parameters.
- Each parameter has an associated parameter domain, which can be used to understand the extent to which different aspects of the system can vary.
- When all actor, edge, and subsystem parameters are assigned specific values from their respective domains, this is called a configuration of the overall dataflow graph.
  - This can be viewed as an instance of the underlying parameterized dataflow graph.
- Parameterized dataflow graphs are reconfigurable in that their parameters can change dynamically.
  - This gives rise to a sequence of dataflow graph instances (configurations) in the execution of a given parameterized dataflow graph.
Meta-modeling with Parameterized Dataflow

• Parameterized dataflow can be applied to any dataflow model of computation (“base model”) to augment that model with dynamic reconfiguration capabilities in a structured way
  – Provides for efficient quasi-static scheduling
  – Enables execution to be viewed in terms of a sequence of dataflow graphs in the base model

• Parameterized dataflow + XYZ → “Parameterized XYZ”

• Examples of parameterized dataflow models: PSDF, PCSDF, …
Reconfiguration and Interface Behavior

Modification of child subsystem parameters can affect the interface behavior of a hierarchical module.
Parameterized Dataflow Constraints

• Each subsystem must have a consistent view of actor interface dataflow behavior throughout any given iteration of that subsystem.

• This applies to hierarchical and primitive actors in a subsystem.

• The “consistent view” is in terms of the base model of computation that is being integrated with parameterized dataflow.
  - PSDF: consistent view → (locally) SDF behavior
  - PCSDF: consistent view → (locally) CSDF behavior
  - etc.

• Depending on where a subsystem parameter is configured (init or subinit), the parameter value can change across well-defined windows of time.

• The notion of iteration here is dependent on the base model, and can be tailored further by the designer.
A PSDF specification (subsystem) $\Phi$ is composed of three PSDF graphs — the \textit{init} graph $\Phi_i$, \textit{subinit} graph $\Phi_s$, and \textit{body} graph $\Phi_b$. 

\begin{itemize}
  \item \textbf{Subsystem}:
    \begin{itemize}
      \item sets $s_2$
      \item \text{params} = \{i_1\}
    \end{itemize}
  \item \textbf{Init}:
    \begin{itemize}
      \item \text{params} = \{s_1,s_2,s_3\}
      \item sets $s_3$
      \item sets $b_1$
    \end{itemize}
  \item \textbf{Body}:
    \begin{itemize}
      \item \text{params} = \{b_1,b_2\}
      \item \text{params} = \{i_1,s_1\}
      \item sets $b_2$
    \end{itemize}
\end{itemize}
PSDF specifications 2

• Semantic hierarchy
• Subsystem parameters
  – Configured in init/subinit
  – Used in body
• Dynamically reconfigurable
• Subinit reconfiguration more restricted than init
Execution of PSDF Specifications

• Operational semantics
  – \texttt{init} invoked at the beginning of each invocation of parent graph
  – \texttt{subinit} invoked at the beginning of each invocation of the associated subsystem
  – body invoked after each invocation of \texttt{subinit}

• Consistency (correctness) depends on local SDF scheduling
  – Ensures bounded memory, and deadlock-free operation
  – Can be used to address real-time requirements

• \textit{Local synchrony} constraints for PSDF graphs and subsystems

• Framework for quasi-static scheduling and local synchrony verification
Local Synchrony

• “Local synchrony” conditions can be formulated and checked in a quasi-static fashion to ensure that bounded token production and consumption along with bounded delays lead to bounded memory requirements overall.
  – This is not true of unstructured dynamic dataflow models, such as general dynamic dataflow, Boolean dataflow, and bounded dynamic dataflow
• Techniques for construction of streamlined looped schedules for synchronous dataflow graphs have natural and efficient extensions to the construction of parameterized looped schedules for PSDF graphs.
Local synchrony of PSDF graphs

- Objective: local SDF scheduling
- For each run-time configuration \( C \) of graph \( G \), the instantiated SDF graph has the following properties
  - Sample-rate consistent
  - Deadlock free
  - Satisfies upper and lower bounds on token transfer and delay values
  - Every child subsystem is locally synchronous
- Depends on input sequences applied to the system
  - Inherently locally synchronous
  - Inherently locally non-synchronous
  - Partially locally synchronous
Local Synchrony of PSDF Specifications

• For a PSDF specification $\Phi$
  – Interface dataflow behavior controlled entirely by init graph
• Each of $\Phi_i$, $\Phi_s$, $\Phi_b$ is locally synchronous
• $\Phi_i$ and $\Phi_s$ produce exactly one token on each interface output port on each invocation
• The token consumption at the interface input ports of $\Phi_s$ is independent of subinit graph parameters that are bound to dataflow inputs of $\Phi$
• The token transfer at the interface ports of $\Phi_b$ is independent of body graph parameters that are not configured in $\Phi_i$.
• Again, inherent vs. partial local synchrony applies.
PSDF scheduling

- **Parameterized loopeed schedule** for PSDF graph $G$
  
  $$S_G = (S_1, S_2, ..., S_m)(preamble_S)(body_S)$$

- **initChild** phase of $S$

- Body of $S$ contains compact *parameterized schedule loops* $(L \, T_1 \, T_2 \, ... \, T_n)$
  
  - Each $T_k$ is an actor or a nested schedule
  - $L$ is the iteration count
  - E.g., repeat $(p/g)$ times \{fire $X$, fire $Y$\}
  - Hierarchical actor $T_k = subsystem(\Phi_k)$ replaced by
    
    $$(VLS_s)(S_{\Phi_s}) (VLS_b)(S_{\Phi_b})$$

- Preamble of $S$: code that defines all compiler-generated PSDF variables, and performs local synchrony checks
PSDF Example: CD to DAT Conversion

```plaintext
repeat 5 times {
    fire setFac /* sets i_1, d_1, i_2, d_2, i_3, d_3, i_4, d_4 */
    int _g1 = gcd(i_1, d_2);
    int _g2=gcd((i_2 x i_1)/_g1, d_3);
    int _g3=gcd((i_3 x i_2 x i_1)/(_g2 x _g1), d_4);
    repeat (d_4/_g3) times {
        repeat (d_3/_g2) times {
            repeat (d_2/_g1) times {
                repeat (d_1) times {fire CD}
            }
            fire PF1
        }
        repeat (i_1/_g1) times {fire PF2}
    }
    repeat ((i_2 x i_1)/(_g2 x _g1)) times {fire PF3}
    repeat ((i_3 x i_2 x i_1)/(_g3 x _g2 x _g1)) times {
        fire PF4
    }
    repeat (i_4) times {fire DAT}
}
```
PSDF application example: speech compression
Speech compression: Parameterization details

- $L$ length of speech instance
- $N$ speech segment size
- $M$ model order (# of AR coefficients)
- $R$ zero padded speech length so it is divisible by $N$
- $N$, $M$, $R$ determined by Select (e.g., using Burg's algorithm for segment size selection)
- $R \geq L$
- Quasi static schedule uses fact that $R$ divides $N$
- Parameterized cyclo-static specification: get rid of zero padding
Quasi-static schedule

repeat (X) times {
  /* init graph schedule */
  execute setsp /* sets parameter L */
  /* subinit graph schedule */
  execute s1, execute select /* sets params. R, M, N */
  /* body graph schedule */
  execute s2
  repeat (R/N) times {
    execute A_n
    repeat (N) times {
      execute q1
      execute d1
    }
    repeat (M) times {
      execute q2
      execute d2
    }
  }
  execute S_n
}
execute PL
Application to cyclo-static dataflow: 
**PCSDF** version of speech compression
Meta-modeling with parameterized dataflow

- Parameterized dataflow can be applied to any dataflow model of computation (“base model”) to augment that model with dynamic reconfiguration capabilities in a structured way
  - Provides for efficient quasi-static scheduling
  - Enables execution to be viewed in terms of a sequence of dataflow graphs in the base model
- Parameterized dataflow + XYZ → “Parameterized XYZ”
- Examples of parameterized dataflow models: PSDF, PCSDF, …
For Further Reading

Available from:
http://www.ece.umd.edu/DSPCAD/papers/contents.html

