
Prof. Shuvra S. Bhattacharyya, ECE/UMIACS/MC2, ssb@umd.edu
December 7, 2015

With contributions from John Kuykendall, Kyunghun Lee, Shuoxin Lin, Yanzhou Liu, Nicholas McCarthy, William Plishker, and George Zaki.

This research has been sponsored in part by the Laboratory for Telecommunications Sciences (LTS).
Outline

• Motivation: GNU Radio as a tool for secure wireless communication system design
• DSPCAD Group Overview
  – Computer-aided Design (CAD) for Digital Signal Processing Systems (DSP)
• Background on model-based design in terms of dataflow graphs
• LIDE: The Lightweight Dataflow Environment
• Summary
GNU Radio

- A software development framework that provides software defined radio (SDR) developers a rich library and a customized runtime engine to design and test radio applications.

http://gnuradio.org
Two Implementation Approaches

**Classical Approach**
- **Re-write** your kernels (using new parallel algorithms) and programming models (e.g., CUDA, OpenMP).
- Customized efficient implementation.
- Programming models are targeted towards a very specific platform.

**Model Based Approach**
- Represent your application in terms of a high level model independent from the platform and write a separate platform model.
- Use tools to analyze and schedule your implementation.
- **Re-use** your previous investments in optimized kernels and developed systems.
Pre-optimized Kernels: GNU Radio

http://gnuradio.org
DSPCAD Methodologies

**Applications and Tasks [Bhattacharyya 2013]**

- **Image**: medical, computer vision, feature detection, etc.
  - Imaging device → Data preprocessing → Image reconstruction
  - Post reconstruction → Advanced image analysis → Image visualization
  - Imaging device:
    - Post reconstruction: Image visualization
  - Advanced image analysis:
    - Post reconstruction: Image visualization
  - Image reconstruction:
    - Post reconstruction: Image visualization

- **Video**: coding, compression, etc.
  - Color processing → Prediction → Transformation & Quantization → Entropy Coding
  - Audio device:
    - Color processing: Image reconstruction
  - Prediction:
    - Color processing: Image reconstruction
  - Transformation & Quantization:
    - Prediction: Image reconstruction
  - Entropy Coding:
    - Transformation & Quantization: Image reconstruction

- **Audio**: sample rate conversion, speech, etc.
  - Audio device → Data preprocessing → Feature extraction → Data postprocessing
  - Audio device:
    - Data preprocessing: Image reconstruction
  - Feature extraction:
    - Data preprocessing: Image reconstruction
  - Data postprocessing:
    - Feature extraction: Image reconstruction

- **Wireless communication systems**
  - Source encoding → Channel encoding → Digital modulation → D/A conversion
  - Source encoding:
    - Wireless communication systems: RF Back-end
  - Channel encoding:
    - Wireless communication systems: RF Back-end
  - Digital modulation:
    - Wireless communication systems: RF Back-end
  - D/A conversion:
    - Wireless communication systems: RF Back-end

**Platforms**

- Programmable DSP
- GPU
- FPGA
- Microcontroller

**Platforms**

- Programmable DSP
- GPU
- FPGA
- Microcontroller

**Applications and Tasks [Bhattacharyya 2013]**

- **Image**: medical, computer vision, feature detection, etc.
  - Imaging device → Data preprocessing → Image reconstruction
  - Post reconstruction → Advanced image analysis → Image visualization
  - Imaging device:
    - Post reconstruction: Image visualization
  - Advanced image analysis:
    - Post reconstruction: Image visualization
  - Image reconstruction:
    - Post reconstruction: Image visualization

- **Video**: coding, compression, etc.
  - Color processing → Prediction → Transformation & Quantization → Entropy Coding
  - Audio device:
    - Color processing: Image reconstruction
  - Prediction:
    - Color processing: Image reconstruction
  - Transformation & Quantization:
    - Prediction: Image reconstruction
  - Entropy Coding:
    - Transformation & Quantization: Image reconstruction

- **Audio**: sample rate conversion, speech, etc.
  - Audio device → Data preprocessing → Feature extraction → Data postprocessing
  - Audio device:
    - Data preprocessing: Image reconstruction
  - Feature extraction:
    - Data preprocessing: Image reconstruction
  - Data postprocessing:
    - Feature extraction: Image reconstruction

- **Wireless communication systems**
  - Source encoding → Channel encoding → Digital modulation → D/A conversion
  - Source encoding:
    - Wireless communication systems: RF Back-end
  - Channel encoding:
    - Wireless communication systems: RF Back-end
  - Digital modulation:
    - Wireless communication systems: RF Back-end
  - D/A conversion:
    - Wireless communication systems: RF Back-end
Abstraction and Mapping

• For modern, complex systems we would like to
  – Create an application description independent of the target
  – Interface with a diverse set of tools and teams
  – Achieve high performance
  – Arrive at an initial prototype quickly

• But algorithms are far removed from their final implementations
  – Low level programming environments
  – Diverse and changing platforms
  – Non-uniform functional verification
  – Entrenched design processes
  – Tool selection
Model-based Design for Embedded Systems

- High level application subsystems are specified in terms of components that interact through formal models of computation
  - C or other “platform-oriented” languages can be used to specify intra-component behavior
  - Model-specific languages can be used to specify inter-component behavior
  - Object-oriented techniques can be used to maintain libraries of components

- Popular models for embedded systems
  - Dataflow and KPNs (Kahn process networks)
  - Continuous time, discrete event
  - FSM and related control formalisms
Dataflow-based Design

Applications and Tasks [Bhattacharyya 2010]

Image: medical, computer vision, feature detection, etc.

- Imaging device
- Data preprocessing
- Image reconstruction
- Post reconstruction
- Advanced image analysis
- Image visualization

Video: coding, compression, etc.

- Color processing
- Prediction
- Transformation & Quantization
- Entropy Coding

Audio: sample rate conversion, speech, etc.

- Audio device
- Data preprocessing
- Feature extraction
- Data postprocessing

Wireless communication systems

- Source encoding
- Channel encoding
- Digital modulation
- D/A conversion

Platforms

Programmable DSP

GPU

FPGA

Microcontroller
Key Projects in the DSPCAD Group

- **Dataflow interchange format (DIF)**
  - Design language for representing abstract properties of dataflow graphs and graph components
  - DIFML: XML variant

- **The lightweight dataflow environment (LIDE)**
  - Abstract (retargetable) application programming interfaces (APIs) for implementing applications in terms of dataflow models
  - Language and platform agnostic

- **DSPCAD integrative command line environment (DICE)**
  - Package of utilities that facilitates efficient management of software and HDL projects
  - Emphasis on support for projects that integrate heterogeneous programming languages
Dataflow-based Design for DSP

- A variety of development environments is based on **dataflow models of computation**.
  - Applications are designed in terms of signal processing block diagrams.

- By using these design tools, an application designer can
  - Develop complete functional specifications of model-based components.
  - Verify functional correctness through model-based simulation and verification.
  - Implement the designs on embedded platforms through supported platform-specific flows.
DSP-oriented Dataflow Models of Computation

• Application is modeled as a **directed graph**
  – Nodes (actors) represent functions
  – Edges represent communication channels between functions
  – Nodes produce and consume data from edges
  – Edges buffer data (**logically**) in a FIFO (first-in, first-out) fashion

• **Data-driven** execution model
  – An actor can execute whenever it has sufficient data on its input edges.
  – The **order in which actors execute is not part of the specification**.
  – The order is typically determined by the compiler, the hardware, or both.

• Iterative execution
  – Body of loop to be iterated a large or infinite number of times
Dataflow Graphs

- Vertices (actors) represent computational modules
- Edges represent FIFO buffers
- Edges may have delays, implemented as initial tokens
- Tokens are produced and consumed on edges
- Different models have different rules for production (SDF → fixed, CSDF → periodic, BDF → dynamic)
Dataflow Production and Consumption Rates

- $p_{x,y}$ denotes the number of tokens produced onto edge $e_x$ by the $y$th firing of its source actor (for $y = 1, 2, \ldots$).

- Similarly, $c_{x,y}$ denotes the number of tokens consumed from edge $e_x$ by the $y$th firing of its sink actor.
# Representative Dataflow Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Abbr.</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous Dataflow</td>
<td>SDF</td>
<td>Fixed firing behavior for all actors</td>
<td>Static</td>
</tr>
<tr>
<td>Cyclo-static Dataflow</td>
<td>CSDF</td>
<td>Periodic firing behavior</td>
<td>Static</td>
</tr>
<tr>
<td>Boolean Dataflow</td>
<td>BDF</td>
<td>Firing behavior may be contingent on the value of a Boolean token</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Parameterized Synchronous Dataflow</td>
<td>PSDF</td>
<td>Firing behavior may be changed parametrically between graph iterations</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Enable Invoke Dataflow</td>
<td>EIDF</td>
<td>Modes have fixed behavior, but actors may dynamically switch between modes</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Core Functional Dataflow</td>
<td>CFDF</td>
<td>A deterministic subclass of EIDF in which the next mode of an actor is always unique</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>
Representative Techniques for Analyzing and Transforming DSP-Oriented Dataflow Graphs

- **Scheduling**
  - Fully static, self-timed, quasi-static, dynamic, ...
  - Latency, throughput, and memory-centric

- **Loop rolling**
  - Translating implicit iteration constructs into efficient control structures (explicit iteration) for implementation
  - Joint code and data minimization

- **Buffer management**
  - Buffer sharing versus buffer merging
  - Bounded memory verification

- **Clustering techniques**
  - E.g., for static, single-rate, or single-processor regions
Outline

• Motivation: GNU Radio as a tool for secure wireless communication system design
• DSPCAD Group Overview
  – Computer-aided Design (CAD) for Digital Signal Processing Systems (DSP)
• Background on model-based design in terms of dataflow graphs
• LIDE: The LIghtweight Dataflow Environment
• Summary
Lightweight Dataflow Programming

• LWDF = “Lightweight Dataflow”
• A dataflow programming approach for model-based design and implementation of DSP systems.
  – By “lightweight”, we mean minimally intrusive on existing design processes, and requiring minimal dependence on specialized tools or libraries.

• Features
  – Improve the **productivity** of the design process and the quality of derived implementations.
  – **Retargetability** across different platforms.
  – Allow designers to integrate and experiment with **dataflow modeling approaches** relatively quickly and flexibly in the context of existing design methodologies and processes.
A “Lightweight” Approach to Writing Dataflow Actors (in C)

- A C-based actor \texttt{xyz} can be implemented as an \textit{abstract data type (ADT)} to enable efficient and convenient reuse of the actor across arbitrary applications

- Such ADTs allow us to provide object-oriented implementations in C
  - Encapsulation, and separation of interface and implementation
  - Inheritance (e.g., through memory layout conventions of C structures)
  - Polymorphism (e.g., through function pointers)

- ADT components
  - Header file \texttt{xyz.h}
    - Definitions that are exported to application developers
  - Implementation file \texttt{xyz.c}
    - Implementation
    - Private (implementation-specific) definitions
Core Functional Dataflow (Model of Computation in LWDF)

- Divide each actor into a set of *modes*
  - Each mode has a fixed consumption and production behavior
- Specify the enabling conditions for each mode
- Specify the computation associated with each mode
  - Including next mode to *enable* and then *invoke*
- For example, consider a standard *switch* actor:

<table>
<thead>
<tr>
<th>Control Input</th>
<th>Data Input</th>
<th>True Output</th>
<th>False Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control:</td>
<td>2, 7, 1, 5,</td>
<td>2, 7, 5,</td>
<td></td>
</tr>
<tr>
<td>Data:</td>
<td>-2, 7, 9,</td>
<td>1, -2, 7, 9,</td>
<td>…</td>
</tr>
</tbody>
</table>

**Example Operation:**

Data:  2, 7, 1, 5, -2, 7, 9, …
True:  2, 7, 5, …
False:  1, -2, 7, 9, …
Switch Actor: Color-coded Illustration of Execution (First Four Executions)

- Control/Data are **synchronous dataflow (SDF)** ports
- True/False are **dynamic dataflow** ports

**Example Operation:**
Control:  
T, T, F, T, F, F, F, ...

Data:  
2, 7, 1, 5, -2, 7, 9, ...

True:  
2, 7, 5, ...

False:  
1, -2, 7, 9, ...
Switch Actor in Terms of CFDF

Mode Transition Table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Input: Control</th>
<th>Input: Data</th>
<th>Output: True</th>
<th>Output: False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>True</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>False</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Mode transition diagram between CFDF modes
Design Flow using Lightweight Dataflow

- Actor library
  - Dataflow graph application
    - Graph transformation and analysis
    - Scheduling and buffer mapping
      - Graph-level functional/implementation validation
- Communication library
  - Unit testing
    - Programmable DSP
    - GPU
    - FPGA
LWDF Design Principles

- Each actor has an operational context (OC), which encapsulates:
  - parameters
  - mode variables
  - local variables
  - references to the FIFOs
    - corresponding to the input and output ports of the actor as a component of the enclosing dataflow graph.
  - reference to the execution functions ("work functions") of the actor
Actor Context Type: Common Definitions (Structure Members)

/* Common elements across context type of actors. */
int mode;
lide_c_actor_enable_function_type_type enable;
lide_c_actor_invoke_function_type_type invoke;

From:
File: "lide_c_actor_context_type_type_common.h",
Release: LIDE Version 0.2

Function pointers that provide **polymorphism**, and are made available through **inheritance**
Inner Product Actor in Terms of CFDF

Parameterized set of modes (PSM) \(\rightarrow\)
(Parameterized) Mode Transition Table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>X</th>
<th>Y</th>
<th>M</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Length</td>
<td>0</td>
<td>0</td>
<td>1 [Length]</td>
<td>0</td>
</tr>
<tr>
<td>Process</td>
<td>Length</td>
<td>Length</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Inputs: X, Y, M
Output: Out

Modes:
Store Length: sets the Length parameter (internal actor parameter)
Process: computes the inner product

Process \(\rightarrow\) PSM in terms of Length

Mode transition diagram

Invalid Length ("self loop")
/* Actor modes */
#define LIDE_C_INNER_PROD_MODE_ERROR 0
#define LIDE_C_INNER_PROD_MODE_STORE_LENGTH 1
#define LIDE_C_INNER_PROD_MODE_PROCESS 2

typedef struct {
#include "lide_c_actor_context_type_common.h"

/* Persistent "local variables" for temporary work */
int length;

/* input ports */
lide_c_fifo_pointer m;
lide_c_fifo_pointer x;
lide_c_fifo_pointer y;

/* output port */
lide_c_fifo_pointer out;
} lide_c_inner_prod_context_type;

inner product

In1 (length)
In2 (vector)
In3 (vector)
out
Operational Context: Example in LWDF-C

In `lide_c_actor_context_type_common.h` header file:

/* Common elements across context type of actors. */
int mode;
lide_c_actor_enable_function_type enable;
lide_c_actor_invoke_function_type invoke;

In `lide_c_actor.h` header file:

/***************************************************************************/
A pointer to a "lide_c_actor_invoke_function", which is a function that invokes an actor with a given context.
***************************************************************************/
typedef void (*lide_c_actor_invoke_function_type)
    (struct lide_c_actor_context_struct *context);

/***************************************************************************/
A pointer to a "lide_c_actor_enable_function", which is a function that enables an actor with a given context.
***************************************************************************/
typedef boolean (*lide_c_actor_enable_function_type)
    (struct lide_c_actor_context_struct *context);
LWDF Design Principles

• Methods that are involved in the implementation of an actor
  – *Enable & Invoke*: implement the CFDF semantics associated with an actor firing.

• Static (constructor) function that is involved in the implementation of an actor
  – *New*: connects an actor to its input and output edges (FIFO channels), and performs any other pre-execution initialization associated with the actor.

• Method for terminating an actor
  – *Terminate*. Performs any operations that are required for “closing out” the actor after the enclosing graph has finished executing.
Code Example: Inner Product Actor Implementation

Key functions in the implementation (.c) file:

lide_c_inner_prod_new
lide_c_inner_prod_enable
lide_c_inner_prod_invoke
lide_c_inner_prod_terminate
Inner Product Actor: Constructor Prototype

```c
lide_c_inner_prod_context_type *lide_c_inner_prod_new(
    lide_c_fifo_pointer m,
    lide_c_fifo_pointer x,
    lide_c_fifo_pointer y,
    lide_c_fifo_pointer out);
```
Inner Product Actor: Constructor Body

```c
lide_c_inner_prod_context_type *context = NULL;

context = lide_c_util_malloc(
    sizeof(lide_c_inner_prod_context_type));
context->mode =
    LIDE_C_INNER_PROD_MODE_STORE_LENGTH;
context->enable = (lide_c_actor_enable_function_type)
    lide_c_inner_prod_enable;
context->invoke = (lide_c_actor_invoke_function_type)
    lide_c_inner_prod_invoke;
context->length = 0;
context->m = m;
context->x = x;
context->y = y;
context->out = out;
return context;
```
boolean lide_c_inner_prod_enable(
    lide_c_inner_prod_context_type *context) {
    boolean result = FALSE;

    switch (context->mode) {
    case LIDE_C_INNER_PROD_MODE_STORE_LENGTH:
        result = lide_c_fifo_population(context->m) >= 1;
        break;
    case LIDE_C_INNER_PROD_MODE_PROCESS:
        result = (lide_c_fifo_population(context->x) >=
                  context->length) &&
                  (lide_c_fifo_population(context->y) >=
                  context->length) &&
                  ((lide_c_fifo_population(context->out) <
                    lide_c_fifo_capacity(context->out)));
        break;
    default:
        result = FALSE;
        break;
    }
    return result;
}
switch (context->mode) {
    case LIDE_C_INNER_PROD_MODE_STORE_LENGTH:
        lide_c_fifo_read(context->m, &(context->length));
        /* Disregard this token if it results in an invalid length. */
        if (context->length <= 0) {
            context->mode = LIDE_C_INNER_PROD_MODE_STORE_LENGTH;
            return;
        }  
        context->mode = LIDE_C_INNER_PROD_MODE_PROCESS;
        break;

    case LIDE_C_INNER_PROD_MODE_PROCESS:
        for (i = 0; i < context->length; i++) {
            lide_c_fifo_read(context->x, &(x_value));
            lide_c_fifo_read(context->y, &(y_value));
            sum += (x_value * y_value);
        }
        lide_c_fifo_write(context->out, &sum);
        context->mode = LIDE_C_INNER_PROD_MODE_STORE_LENGTH;
        break;

    default:
        context->mode = LIDE_C_INNER_PROD_MODE_STORE_LENGTH;
        break;
}
void lide_c_inner_prod_terminate(
    lide_c_inner_prod_context_type *context) {
    free(context);
}
Another Example: Block Addition Actor Implementation

/* Actor modes */
#define LIDE_C_BLOCK_ADD_MODE_ERROR          0
#define LIDE_C_BLOCK_ADD_MODE_READ_BLOCK1    1
#define LIDE_C_BLOCK_ADD_MODE_READ_BLOCK2    2
#define LIDE_C_BLOCK_ADD_MODE_SUM            3

/* Structure and pointer types associated with block add objects. */

struct _lide_c_block_add_context_struct;
typedef struct _lide_c_block_add_context_struct
    lide_c_block_add_context_type;
Parameter-related Interface Functions

/* Actor parameter: block_length */

void lide_c_block_add_set_block_length(lide_c_block_add_context_type
 *context, int length);

int lide_c_block_add_get_block_length(lide_c_block_add_context_type
 *context);

LWDF-C design convention: use of coupled get/set methods for each actor parameter.

<actor name>_get_<parameter name>
<actor name>_set_<parameter name>
Standard Interface Functions

/* Constructor */

lide_c_block_add_context_type
*lide_c_block_add_new(lide_c_fifo_pointer in1,
    lide_c_fifo_pointer in2, lide_c_fifo_pointer out, int
    block_length);

/* CFDF interface */

boolean lide_c_block_add_enable(lide_c_block_add_context_type
    *context);

void lide_c_block_add_invoke(lide_c_block_add_context_type
    *context);

/* Destructor */

void lide_c_block_add_terminate(lide_c_block_add_context_type
    *context);
Lightweight Dataflow: Summary

• LIDE provides a “lightweight” approach to designing and implementing dataflow actors.
• The code for an actor is decomposed into a sequence of modes based on core functional dataflow semantics.
• Actors are programmed in terms of construct, enable, invoke, and terminate (deconstruct) methods.
• Parameters are implemented through a standardized interface convention involving coupled `get/set` methods.
• The approach is designed for agility, retargetability, and efficient integration into existing design processes.
• Use of CFDF modeling enables systematic integration with a wide variety of specialized dataflow methods for design, analysis and optimization.
LIDE

LIDE = The Lightweight Dataflow Environment

Available from:

http://www.ece.umd.edu/DSPCAD/projects/lide/lide.htm
Summary

• GNU Radio as a tool for secure wireless communication system design
• DSPCAD Group Overview
  – Computer-aided Design (CAD) for Digital Signal Processing Systems (DSP)
• Model-based design in terms of dataflow graphs
• LIDE: The Lightweight Dataflow Environment