Conversion of Reference C Code to Dataflow Model:

H.264 Encoder Case Study

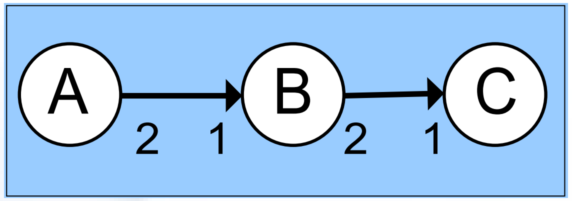
Hyeyoung Hwang, Taewook Oh, Hyunuk Jung, Soonhoi Ha
{hyhwang, twoh, jung, sha}@iris.snu.ac.kr

Contents

Introduction

- **Model-Based Design Methodology**
  - Advantages of Model-Based Design Approach
    - Ease of design ⇨ reduce design complexity
    - Ease of reuse ⇨ meet time-to-market
  - Bottleneck of Model-Based Design Approach
    - Algorithms are often represented in sequential reference code
    - Conversion to appropriate model requires in-depth knowledge of the algorithm
  - Contribution of our work
    - Systematic conversion of sequential code to dataflow spec.
    - Application to H.264 encoder algorithm (reference code)

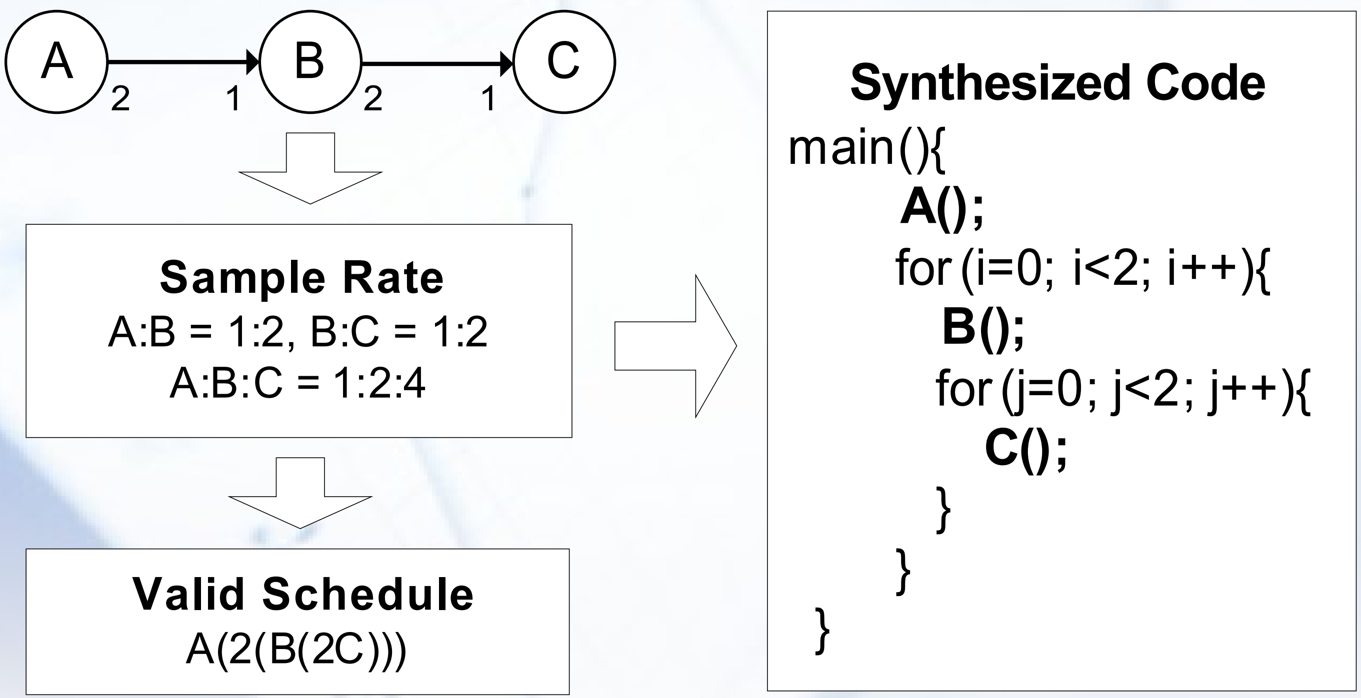
Background



- **Synchronous Data Flow (SDF) Model**
  - Elements of SDF Model
    - Node (function block)
      - Represents a coarse grain function block
      - Ports are used to send/receive data samples
      - A *node* executes only after receiving specific number of data samples at all input ports
    - Arc
      - Represents a stream of data samples
      - Connects one producer *node* to one consumer *node*
    - Sample Rate
      - Number of samples produced/consumed by a *node* per execution
      - The sample rate is always a static integer number

❑ Synchronous Data Flow Model (2)

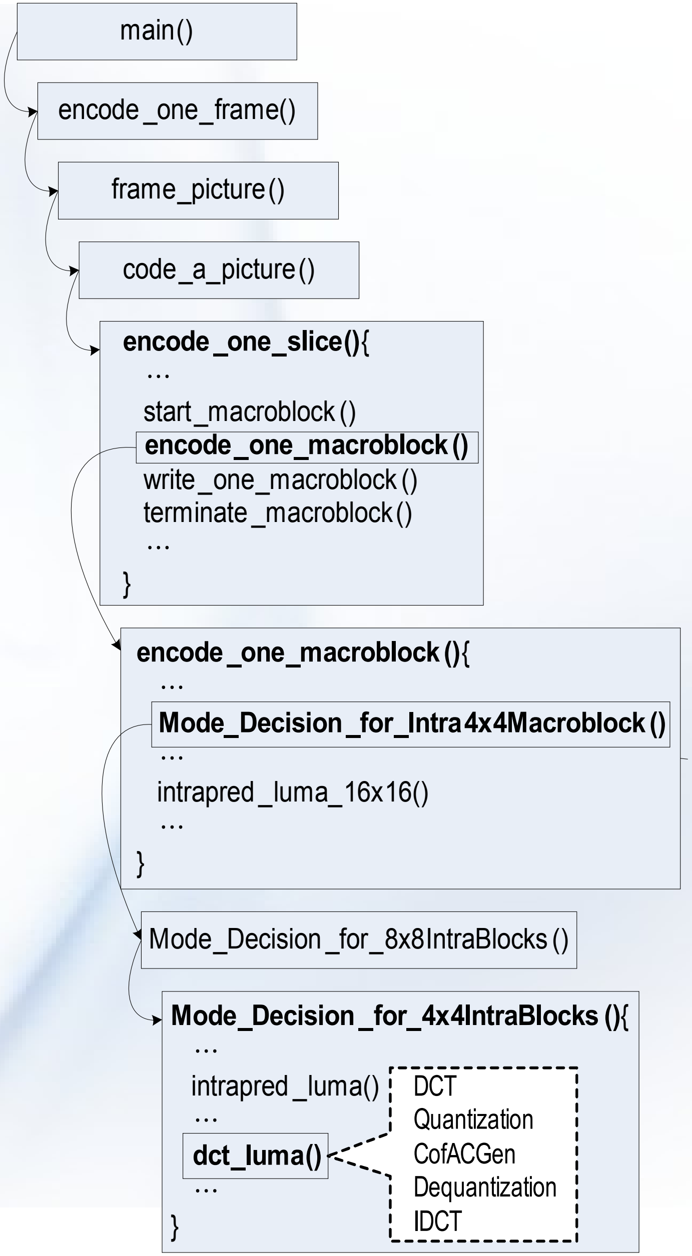
- ❑ Code structure of synthesized code from dataflow spec.
  - ❑ Execution schedule of functions blocks are statically determined
  - ❑ All function blocks appear in the main loop



Example of synthesized code from SDF

Background

- **Example: H.264 Reference Code**
  - Characteristics of reference code
    - Call depth is very deep
    - Key functions reside close to the bottom
    - Use of global variables are common
    - Data is very tightly coupled
    - Difficult to grasp the data flow dependency between functions



Background

- **Example: Transformed Code**
  - Characteristics of SDF code
    - Call depth is shallow
    - Key functions reside in top loop
    - Use of global variables is minimized
    - Easy to grasp data usage
    - Easy to grasp data flow dependency

```
main(){
  …
  ReadOneFrame()
  …
  for(99){
    …
    generate_mb()
    …
      intrapred_luma_16x16()
      for(16){
        intra4_prediction()
        Intra4PredNSel()
        Intra4Dct4x4()
        Intra4Quant()
        Intra4cofACGen()
        Intra4DeQ4x4()
        Intra4Idct4x4()
        Intra4PreReBlockGen()
        Intra4ReBlockGen()
      }
      inter()
      …
  }
}
```

dct_luma()

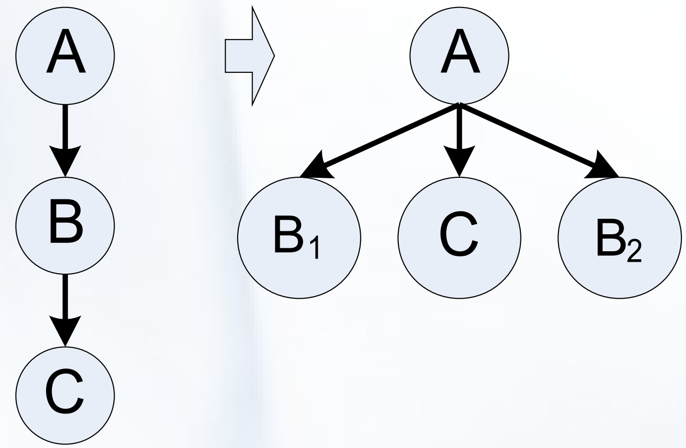
Code Transformation Technique

- **3 Techniques of code transformation**
  - Function restructuring
    - Basic function blocks are identified and moved to top level
  - Variable classification
    - Scope of variables are classified and analyzed
    - Used for removing global variables (when possible)
    - Critical for isolating function blocks that communicate with other function blocks via port variables only
  - Data sample rate decision
    - Determine sample rate of function blocks
    - Used for determining execution frequency of function blocks

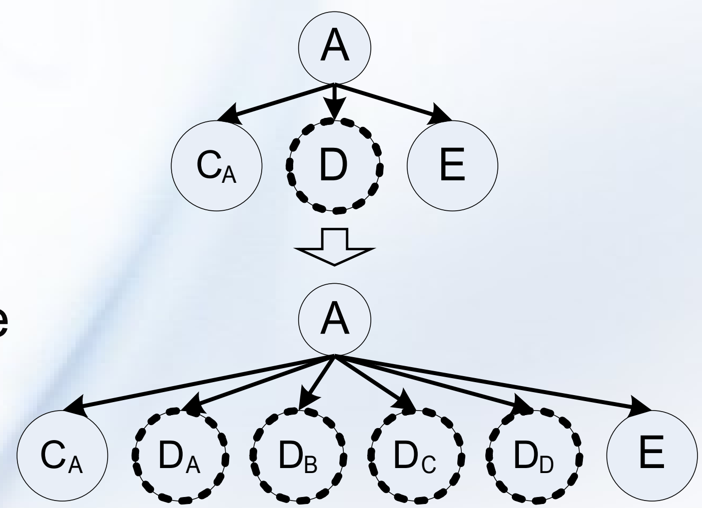
Code Transformation Technique

Function Restructuring (1/2)

- Flattening
  - Call a function from a higher level
  - Makes a function block visible at top



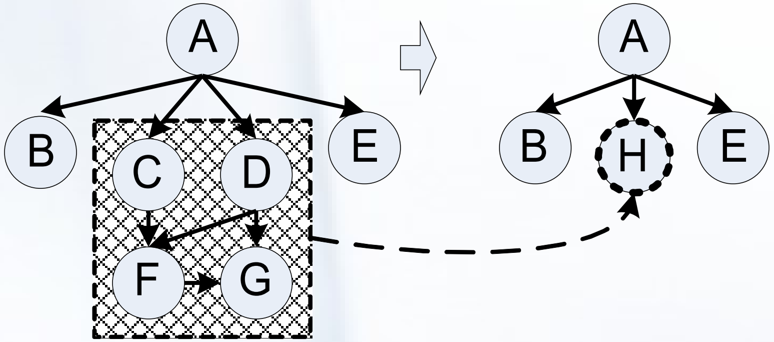
- Splitting
  - Partition a function into many small functions
  - Reduces complexity of a function
  - Allows a function to perform a single functionality



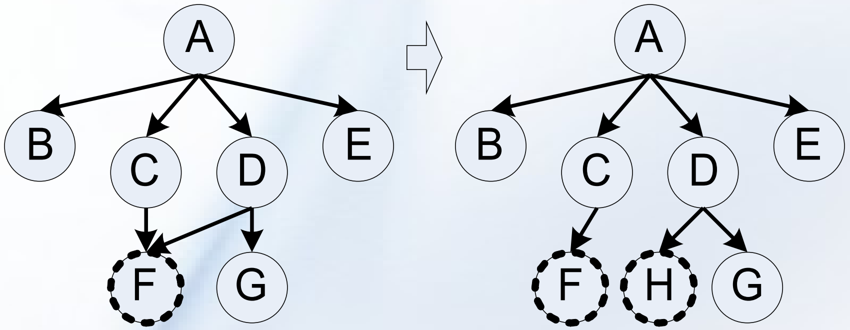
Code Transformation Technique

Function Restructuring (2/2)

- Merging
  - Merge small function into one large function
  - Hides dependency between merged functions

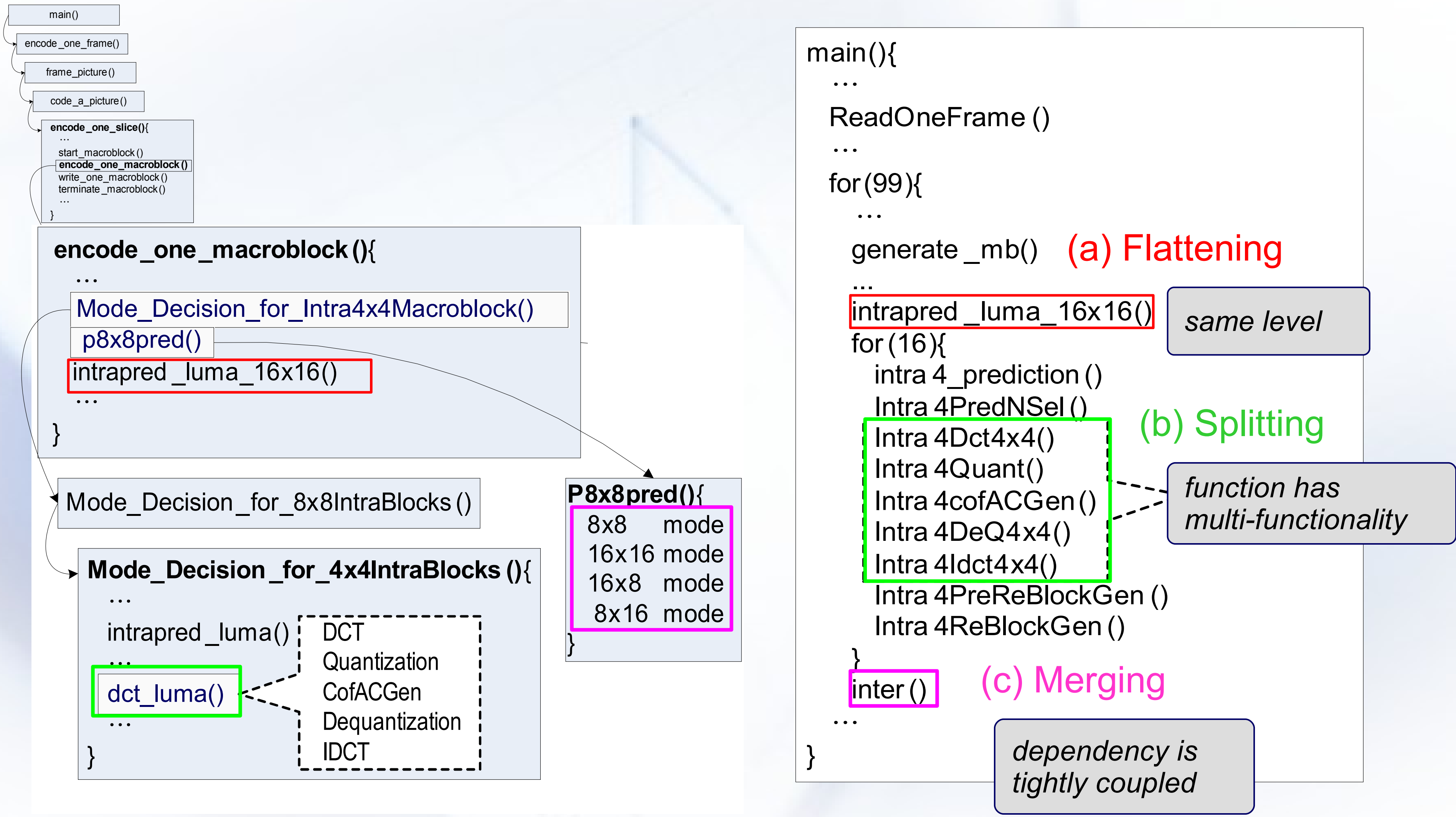


- Duplication
  - Create a new function that has the identical functionality as an existing function
  - Removes dependency between functions



Code Transformation Technique

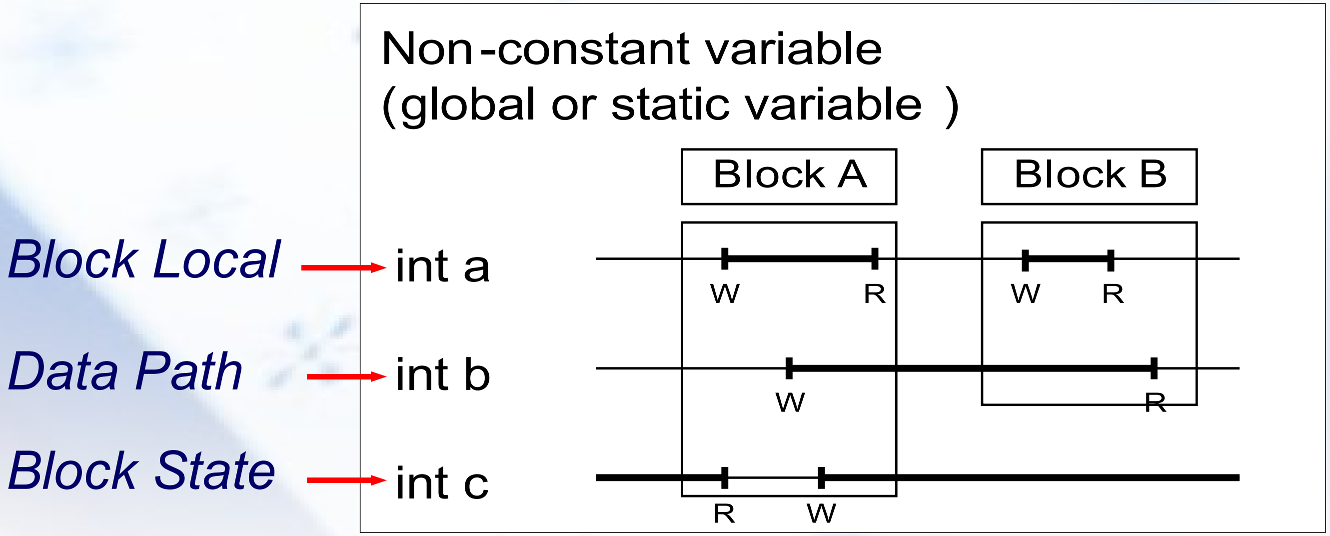
❑ Function Restructuring (2/2) - Example



Code Transformation Technique

❑ Variable Classification (1/2)

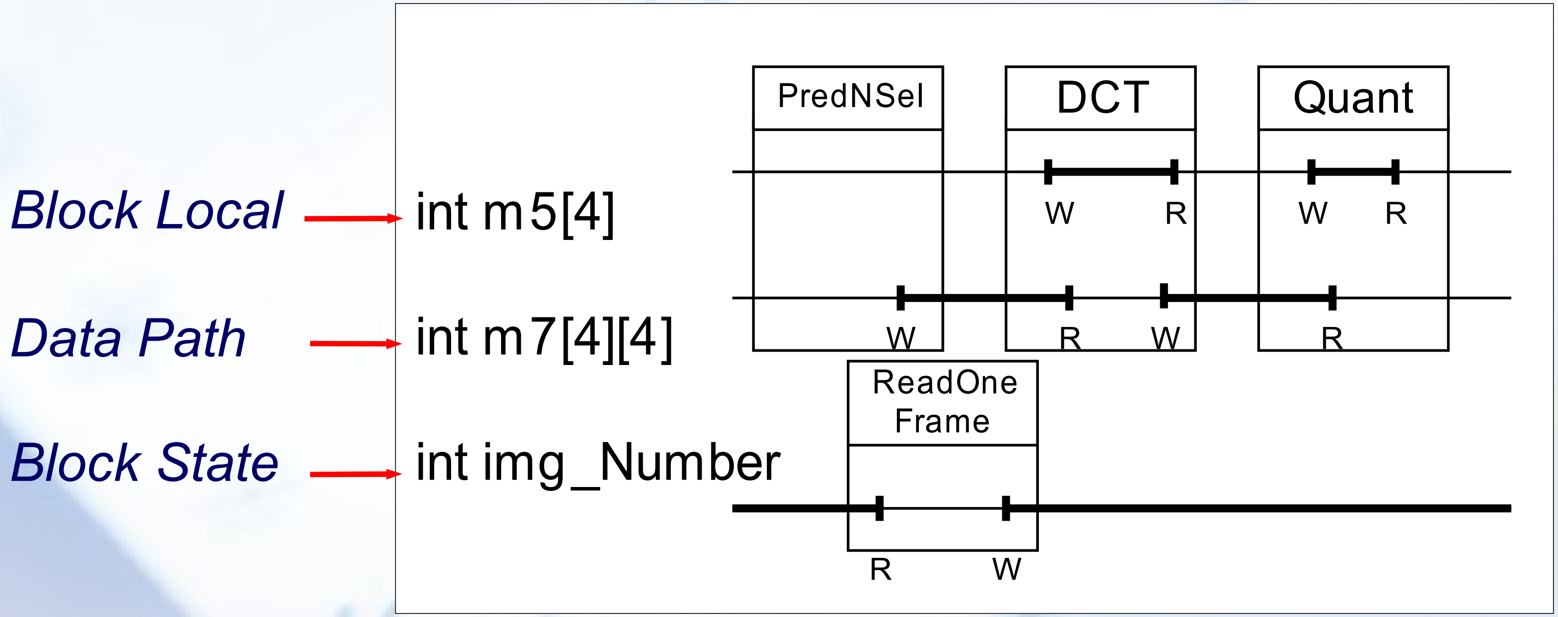
- ❑ Non-constant variables
  - ❑ Data path: A variable used to pass data between function blocks
  - ❑ Block local: A variable read/modified by multiple function block
  - ❑ Block state: A variable that affects the outcome of next iteration
- ❑ Constant variables
  - ❑ Block parameter: Constant variable used inside a function block



Code Transformation Technique

❑ Variable Classification (2/2) - Example

- ❑ Example: H.264 Encoder



Code Transformation Technique

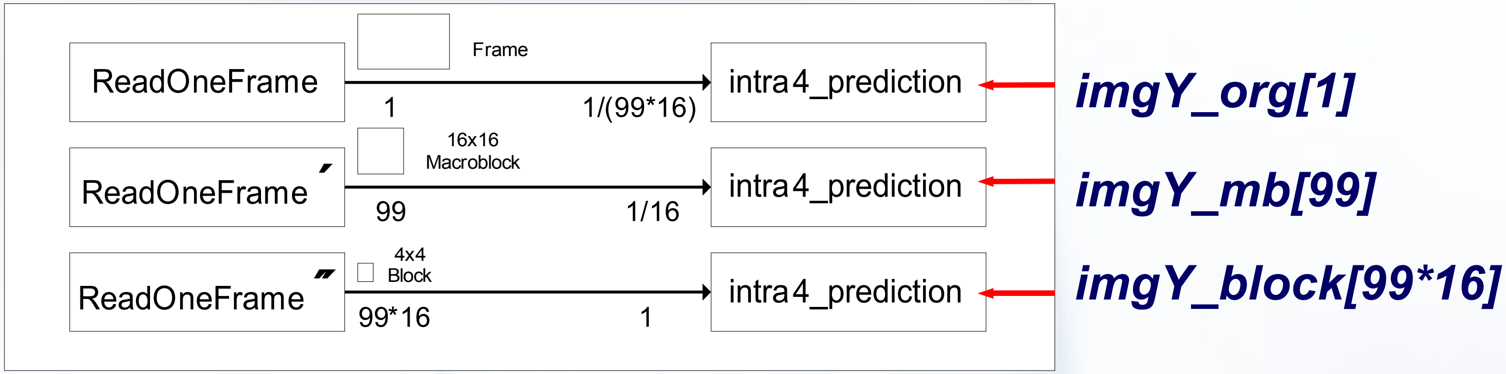
- **Data Sample Rate Decision (1/2)**
  - Determine data sample rate & execution frequency
    - Examine how many data samples are produced / consumed at each port of function blocks
    - Rate represent relative execution frequency of function blocks
  - Example: H.264 encoder
    - Analyzing the rate of data production and consumption for function blocks *ReadOneFrame* and *intra4_prediction*
    - 1 invocation of the function block ReadOneFrame triggers the function block *intr4_prediction* 99x16 times.

```
main(){
   …
   ReadOneFrame (p_in, img_number, 176,144, imgY_org, imgUV_org);
   …
   for(99x16){
       …
       intra 4_prediction ( imgY_org, pred_data, output, org_block, Mb_nr,
                            imgY _mb_phase, pred_data_Y);
    }
   …
}
```

3 possible assignments of sample rates between two blocks

Code Transformation Technique

❑ Data Sample Rate Decision (2/2)



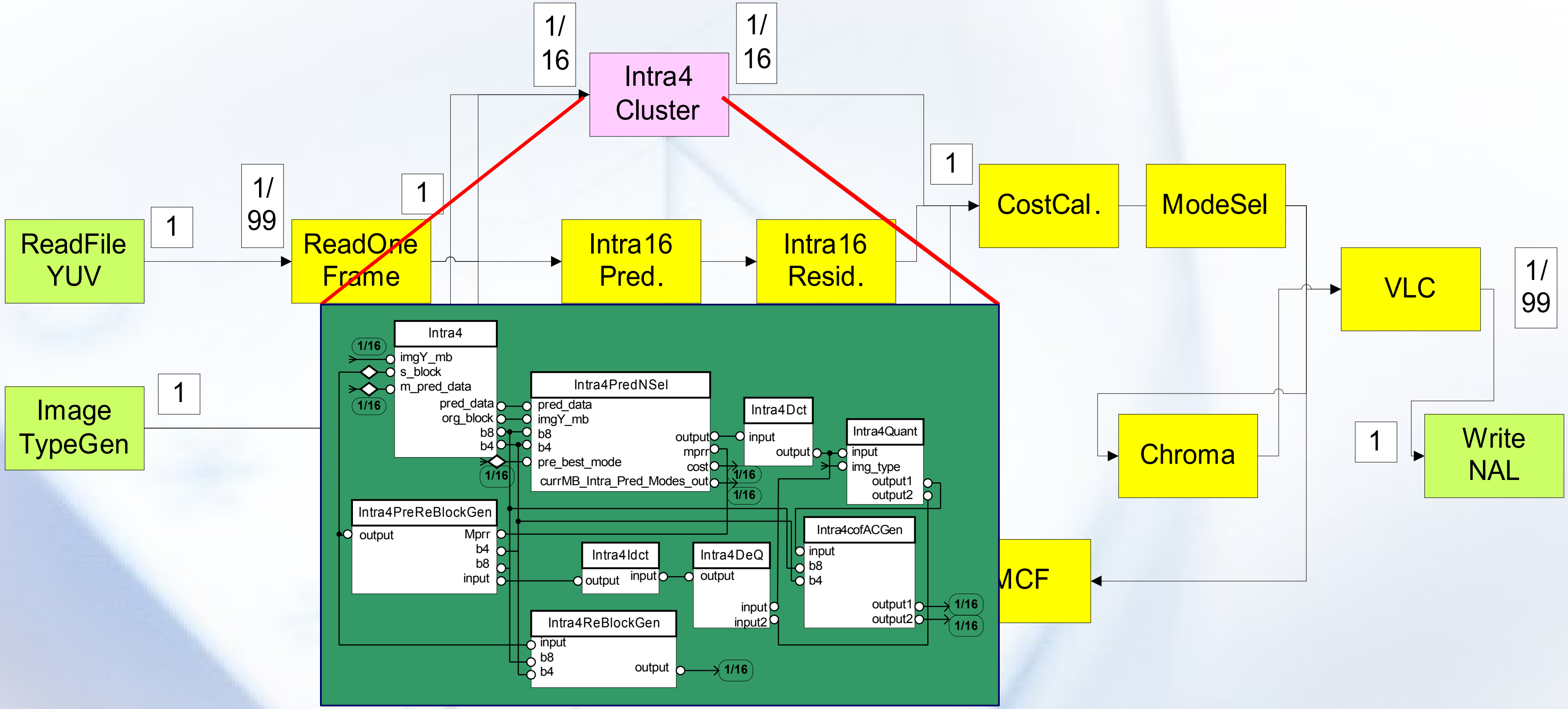

```
main(){
  …
  imgY_org[1];
  …
  ReadOneFrame(...imgY_org...);
  …
  for(99){
    …
    for(16){
      …
      intra4_prediction(...imgY_org...);
    }
}
```

```
main(){
  …
  imgY_mb[99];
  …
  for(99){
    …
    ReadOneFrame'(...imgY_mb...);
    for(16){
      …
      intra4_prediction(...imgY_mb...);
    }
}
```

```
main(){
  …
  imgY_block[99x16];
  …
  for(99){
    …
    for(16){
      ReadOneFrame˝(...imgY_block...);
      …
      intra4_prediction(...imgY_block...);
    }
}
```

Clustering and Scheduling

❑ Clustering and Scheduling

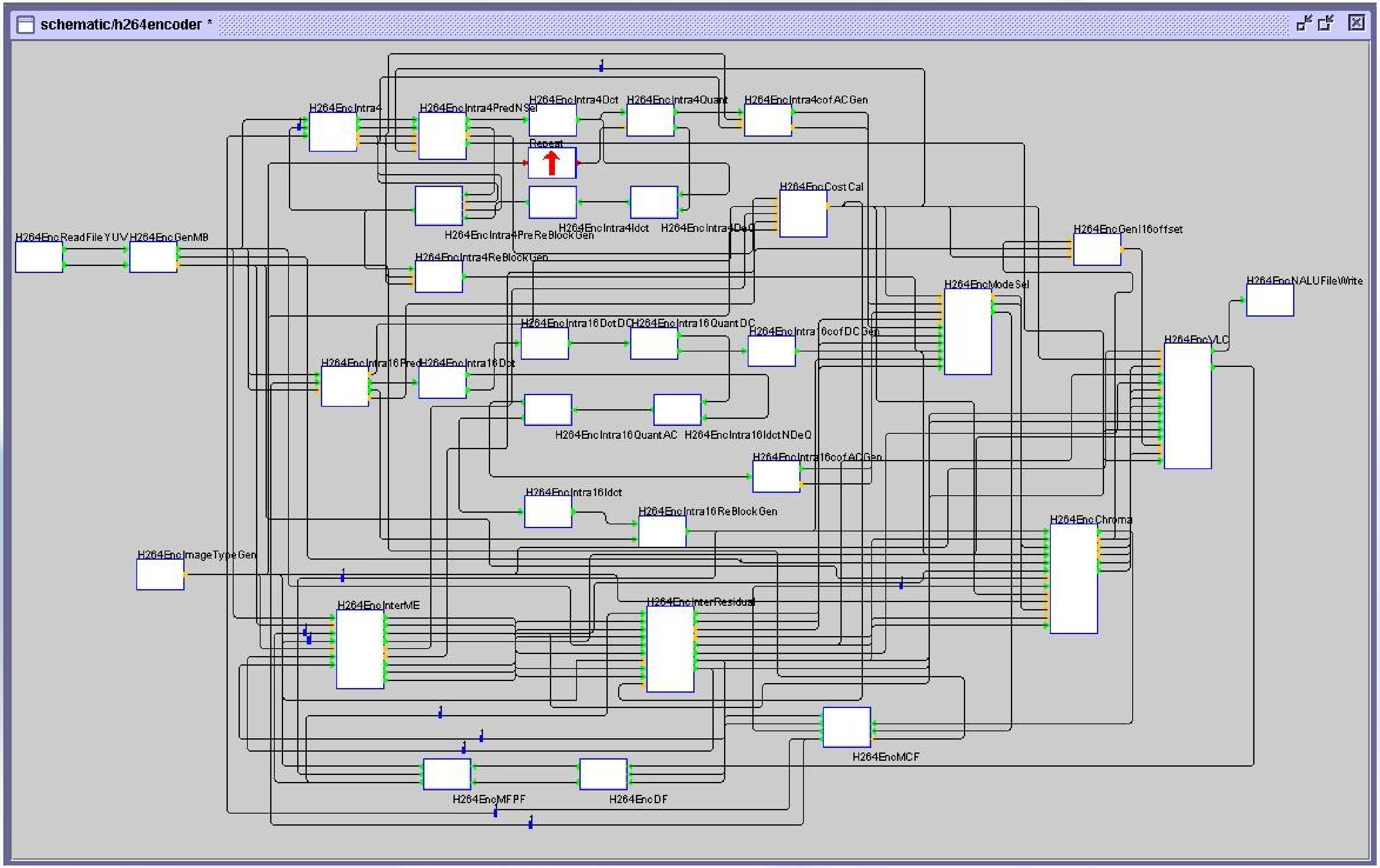


❑ Execution Rate ➔ **Green: Yellow: Pink = *1 : 99 : 16x99***

❑ Sample rate can be a fractional number (Fractional Rate DataFlow(FRDF[4])).

Experimental Results

❑ SDF modeled H.264 encoder in PeaCE [8]



Experimental Results

❑ SDF modeled H.264 encoder in PeaCE

Performance Criteria		Original JM	Modified JM	Synthesized JM
Encoding Time per Frame	I-Frame	0.04 sec	0.04 sec	0.05 sec
	P-Frame	0.70 sec	0.14 sec	0.15 sec
Code Size (byte)		369275	115863	148442
Data Size (byte)		7793536	2747900	2051908

- ❑ Development Time
  - ❑ 2 man-month for transform reference code to dataflow model
  - ❑ 2 man-month to draw a dataflow graph from the transformed code
  - ❑ Includes the learning time of H.264 encoder algorithm

Conclusion

- **Code transformation for model-based approach**
  - Key techniques for systematic procedure is presented
    - Function restructuring
    - Variable classification
    - Data rate sample decision
  - Successful demonstration using complex real-life appl.
    - Applied to H.264 reference encoder
    - Required 4 man-months
      - Experts in dataflow modeling
      - Little knowledge of H.264 algorithm
    - Performance comparison between modified vs. original code
      - Comparable execution time
      - Substantially improved memory utilization

Q & A

Thank You!