COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications

Jieru Zhao	*Hong Kong Univ. of Science and Technology*
Liang Feng	*Hong Kong Univ. of Science and Technology*
Sharad Sinha	*Nanyang Technological Univ., Singapore*
Wei Zhang	*Hong Kong Univ. of Science and Technology*
Yun Liang	*Peking Univ., China*
Bingsheng He	*National Univ. of Singapore*

香港科技大學 THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

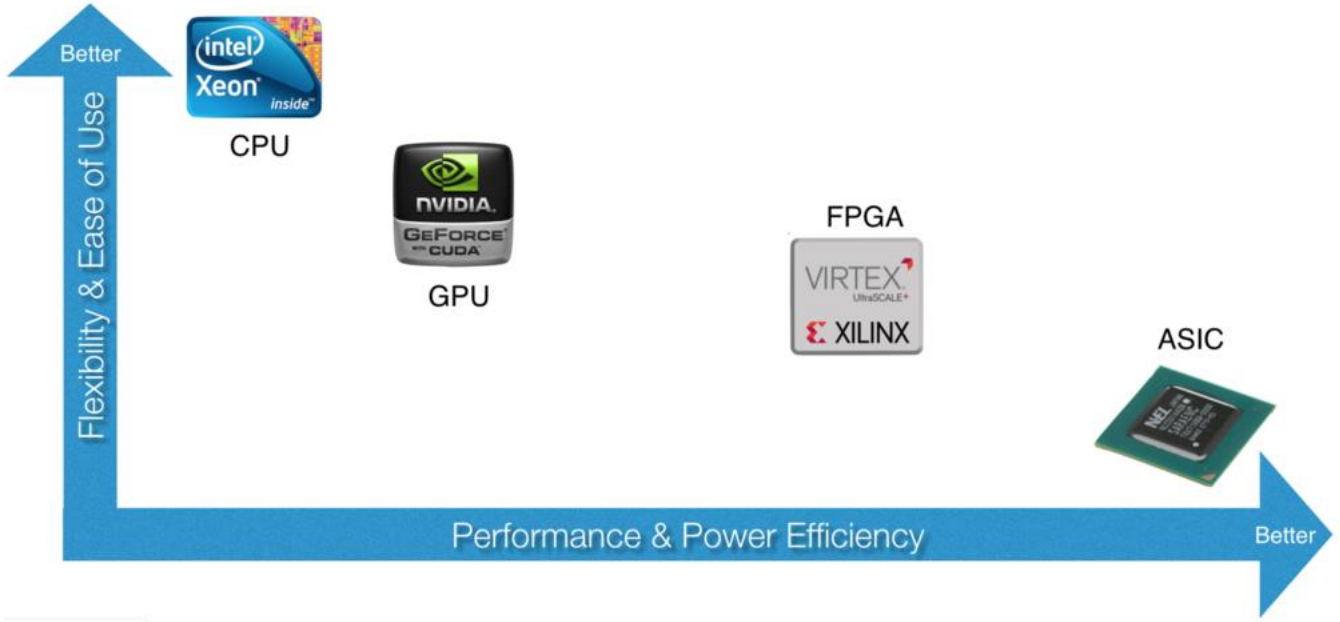
NANYANG TECHNOLOGICAL UNIVERSITY

北京大學 PEKING UNIVERSITY

NUS National University of Singapore

Background

Why FPGA?



- **High** performance
- **Low** power
- Post-fabrication **reconfigurability**
- **Short** time-to-market

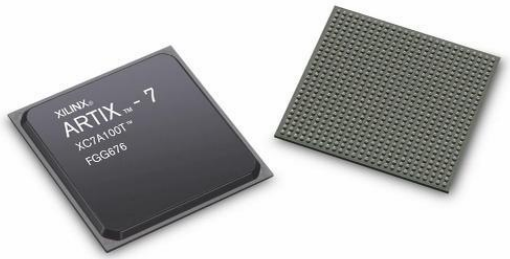
FPGA Applications

Data Center

Medical

Wireless Communications

Broadcast



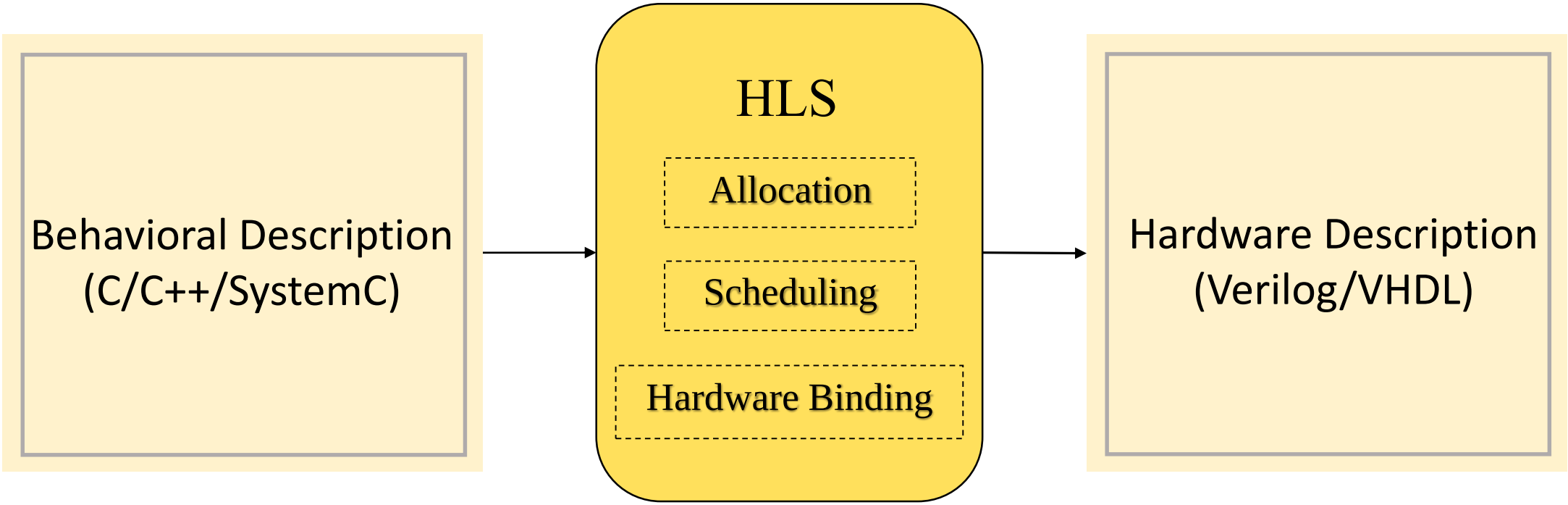
Aerospace and Defense

Automotive

Wired Communications

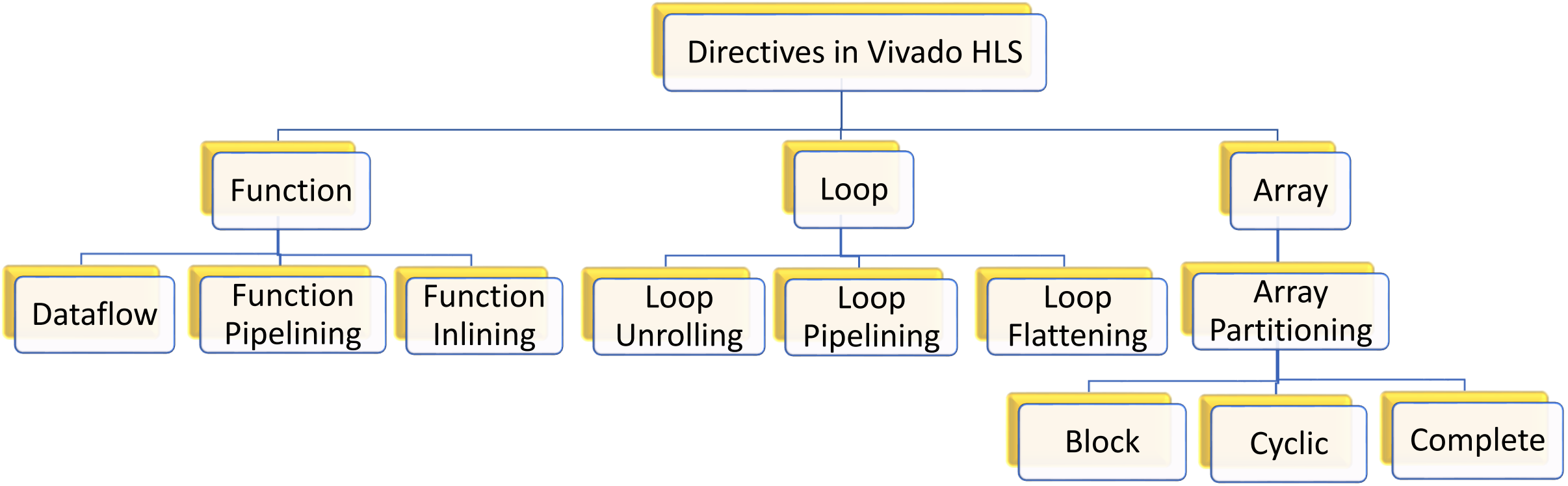
High Level Synthesis

Automatic Transformation



Programmability of FPGA is further improved!

Pragmas in HLS tool

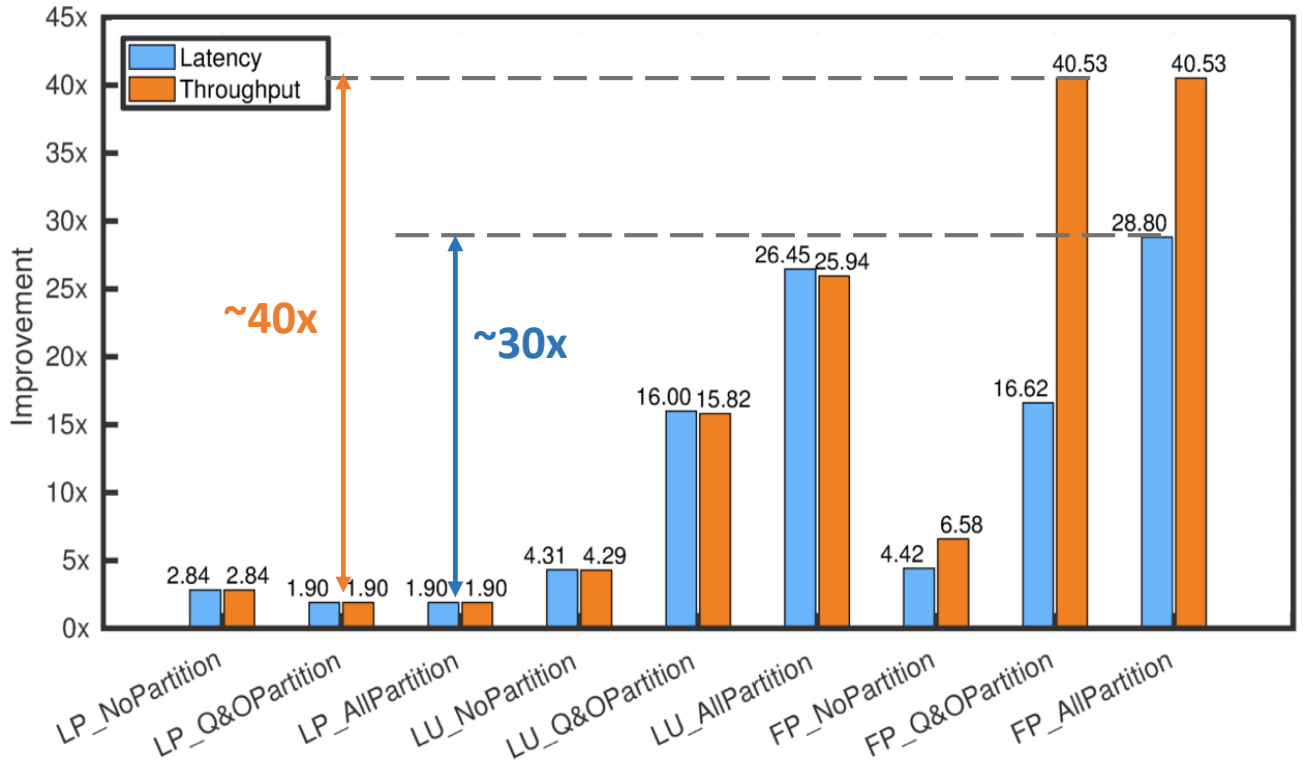


Selection of **directives/pragmas** → **Determines** → The **quality** of RTL designs

How to choose the most **suitable** pragmas quickly?

Motivational Example

```
void decode_block (int Quant[64], int Out[64], int Huff[64]){
    IZigzagMatrix(Huff, Quant); //one loop with bound 64
    IQuantize(Quant);           //one loop with bound 64
    ChenIDct(Quant,Out);        //three loops with bounds 8,8,64
    PostshiftIDctMatrix(Out);   //one loop with bound 64
    BoundIDctMatrix(Out);       //one loop with bound 64
}
```



- More than **millions** of combinations.
- The performance difference can be very **large**!
- **Important:**

  To choose the **best** configuration.
- **Difficult:**

  To **quickly** find the best one.
- For this application, COMBA spends **10** min to find the high-performance configuration, in a design space with 7.61×10^{12} points.

Contributions

State-of-the-art[1]

- **Three** pragmas
  1) Loop unrolling
  2) Loop pipelining
  3) Array partitioning

COMBA

- **Seven** pragmas
  1) Loop unrolling
  2) Loop pipelining
  3) Array partitioning
  4) Function pipelining
  5) Dataflow
  6) Loop flattening
  7) Function inlining

[1]Lin-analyzer.: a high-level performance analysis tool for fpga-based accelerators. DAC Conference. 2016.

Contributions

State-of-the-art[1]

- **Performance** models for **simple** code structures
- **No resource** models
- Analysis on **execution trace**
- **Small** design space: ~10^2 points
- Design space exploration: **Brute force** method

COMBA

- **Performance** models for more **complex** code structures
- **Resource** models: DSP and BRAM
- Analysis on **source code**
- **Enlarged** design space: ~10^8 points
- Design space exploration: **Efficient** Metric-Guided DSE algorithm

[1]Lin-analyzer.: a high-level performance analysis tool for fpga-based accelerators. DAC Conference. 2016.

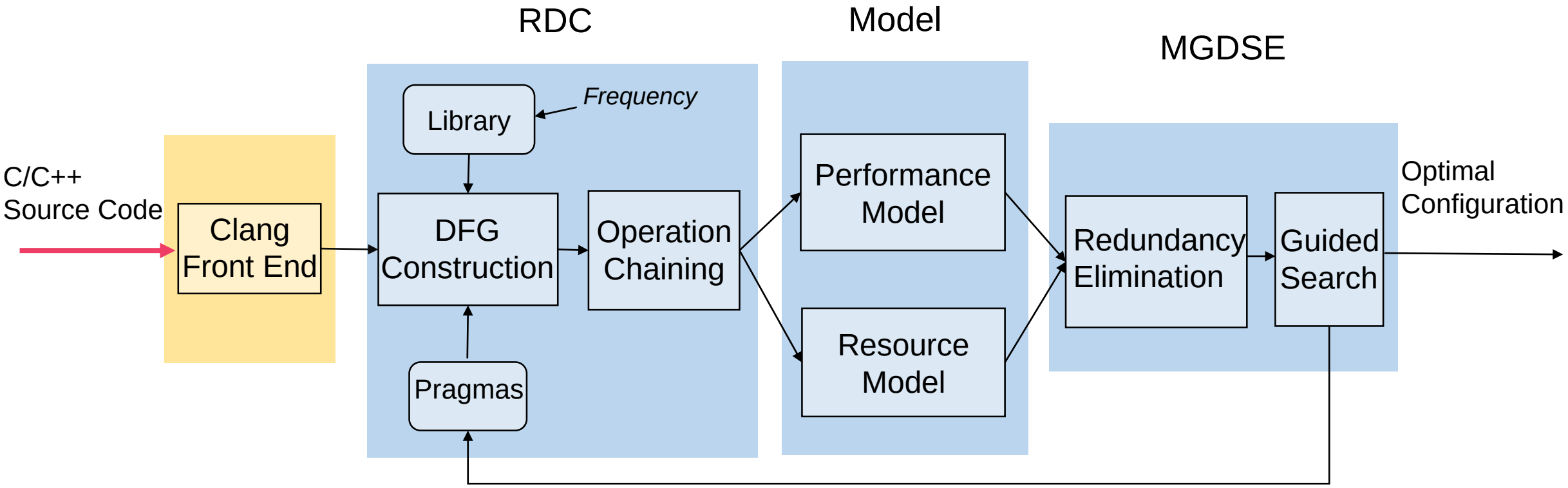
Background

Framework

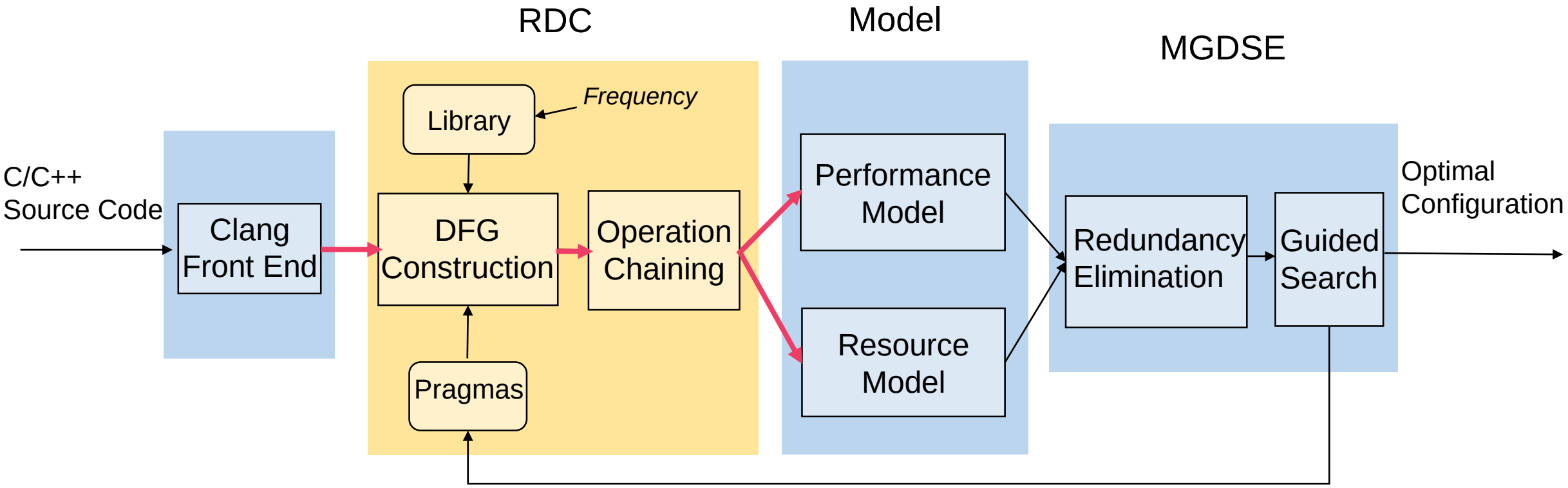
1. RDC
2. Models
3. MGDSE

BA **iterates** until it finds the **high-performance** configuration.

Overview


RDC
Model
MGDSE
C/C++
Source Code
Clang
Front End
Library
Frequency
DFG
Construction
Operation
Chaining
Pragmas
Performance
Model
Resource
Model
Redundancy
Elimination
Guided
Search
Optimal
Configuration

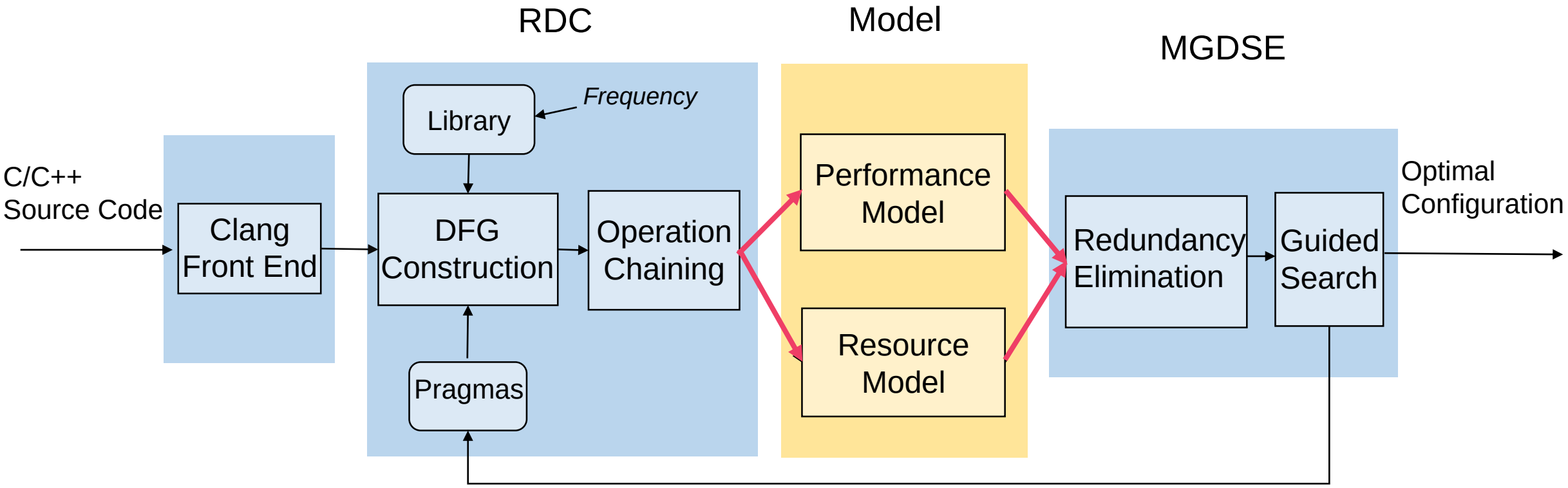
Overview



RDC: Recursive data collector.

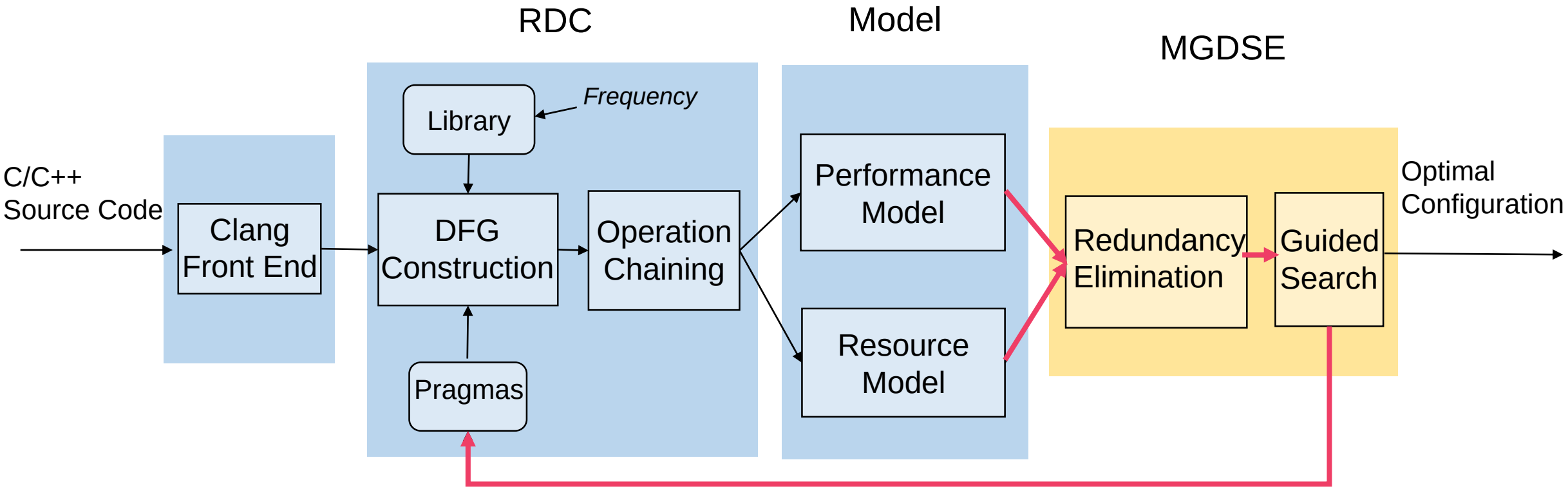
Computes the parameters required by our model.

Overview



Model: **Estimates** the performance and resource usage

Overview

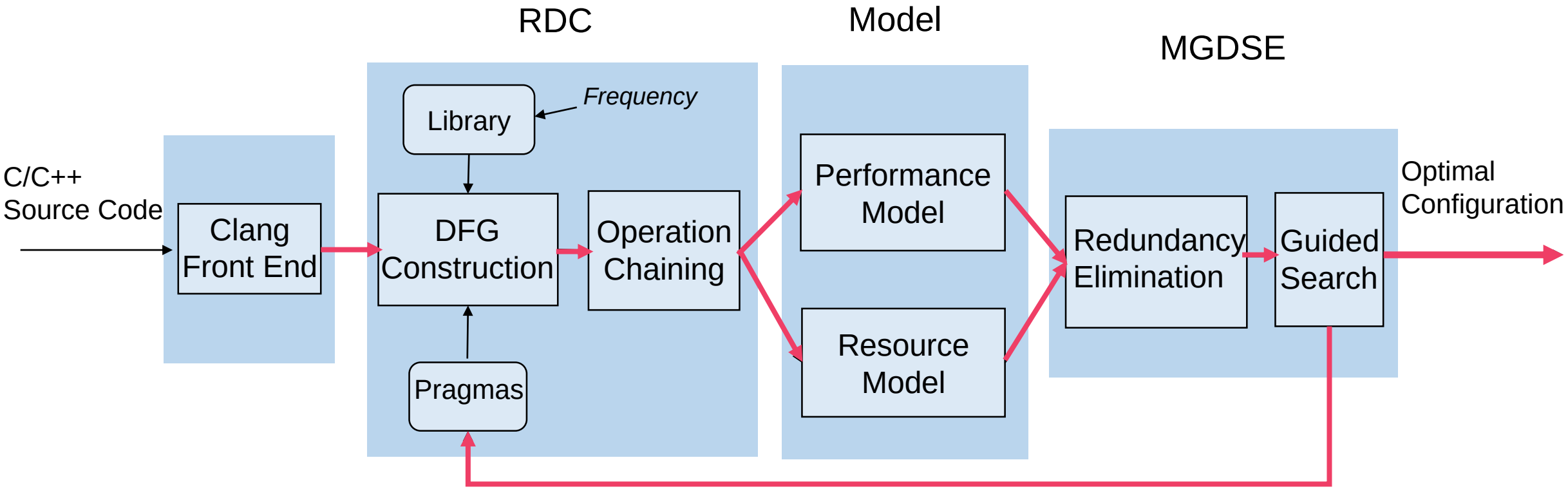


MGDSE: an algorithm for design space exploration

Evaluates the estimated results

Sets the next configuration for RDC.

Overview



COMBA **iterates** until it finds the **high-performance** configuration.

Framework

1. **RDC**
2. **Models**
3. **MGDSE**

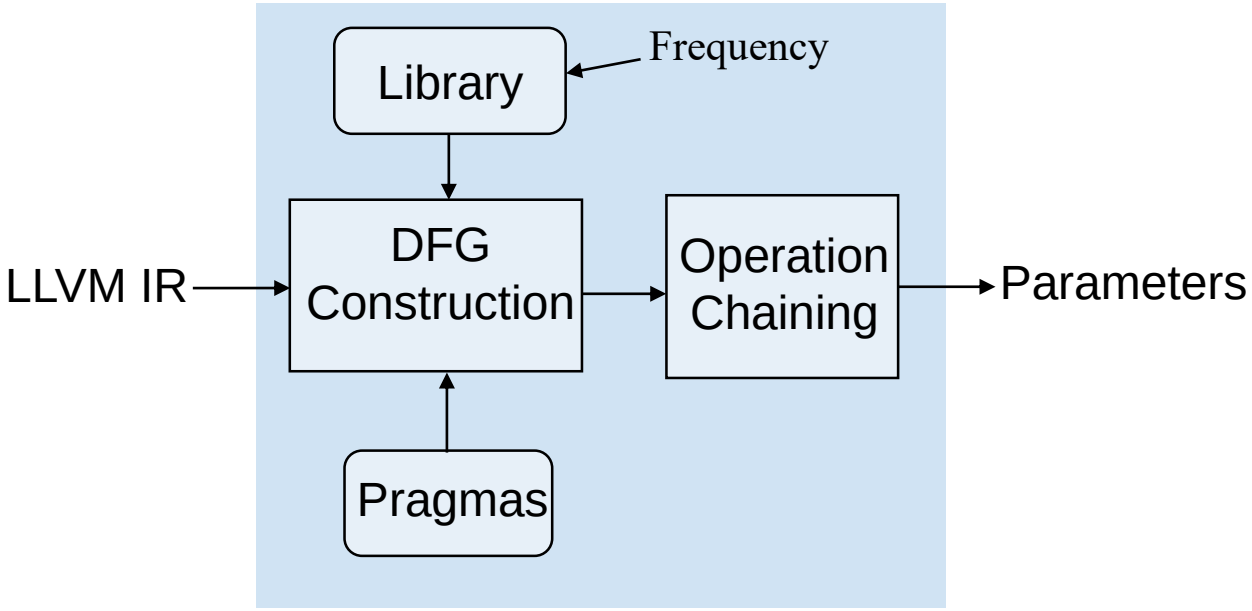
e Data Collection

graph (DFG) construction
on chaining

More than one operation can be scheduled in one cycle if possible.

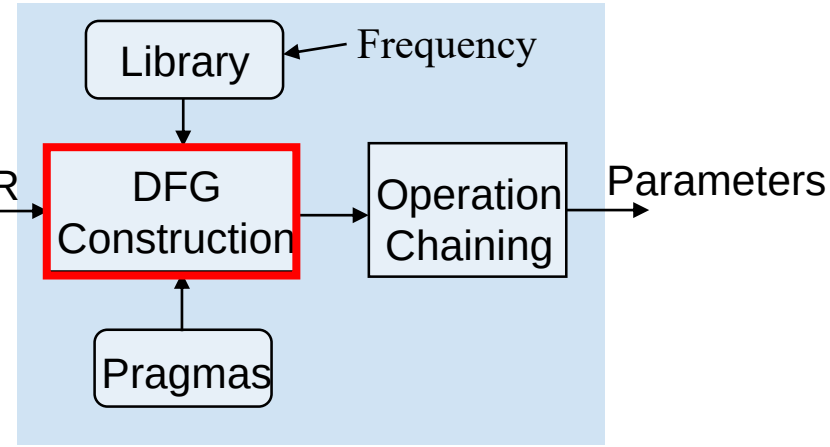
A **dynamic programming** approach is employed to trace each path and compute the latency in the critical path.

1 Recursive Data Collection



- RDC: an optimization pass based on *llvm::Module* class.
- LLVM IR: LLVM intermediate representation

1 Recursive Data Collection

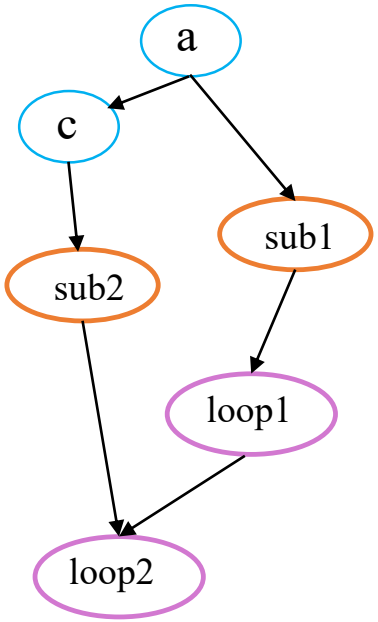


Data flow graph (DFG) construction

```
void function(…)
{
    int a = t * m;
    int c = a + b;

    sub_func1(a, *p);
    sub_func2(c, *q);

    loop1{
        p[i] …
    }
    loop2{
        q[i] …
    }
}
```

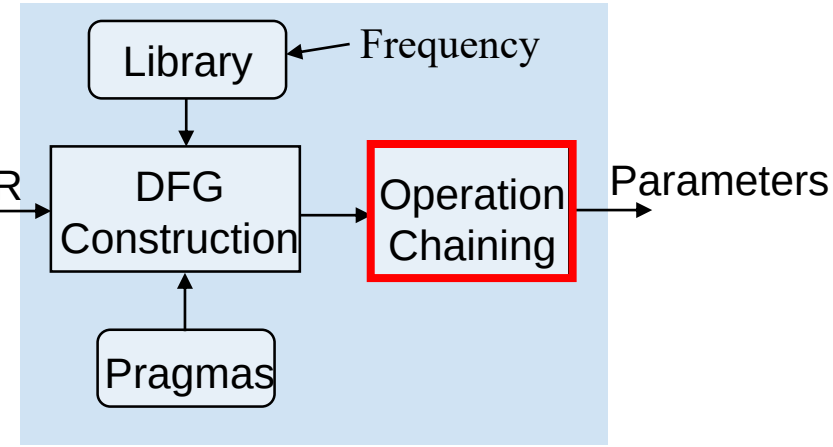


Logic operations

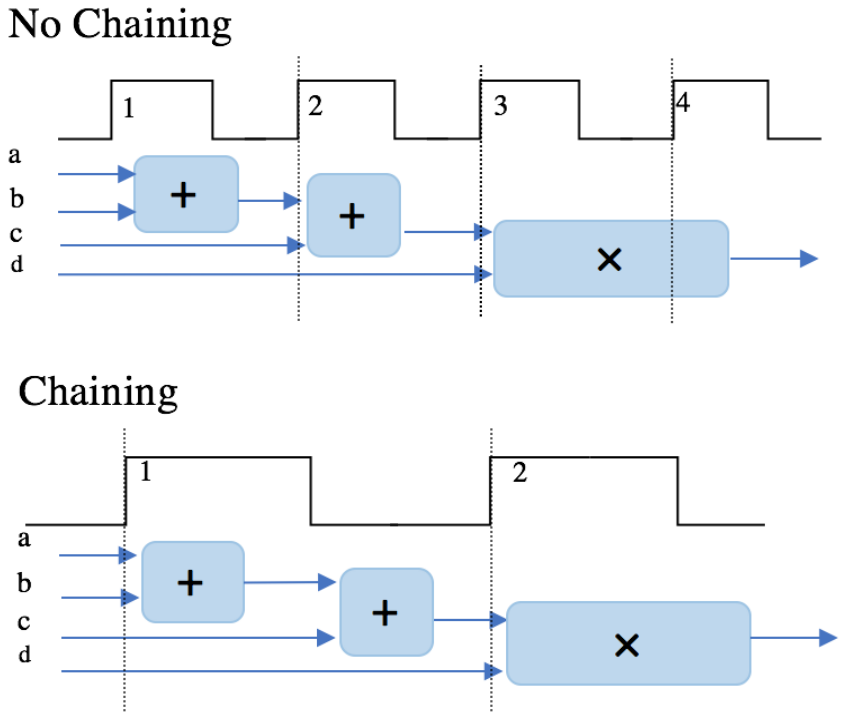
Invoked functions

Loops

1 Recursive Data Collection



Operation chaining



More than one operation can be scheduled in one cycle if possible.

A **dynamic programming** approach is employed to trace each path and compute the latency in the critical path.

Framework

1. RDC
2. **Models**
3. MGDSE

Models: Performance Model

ng

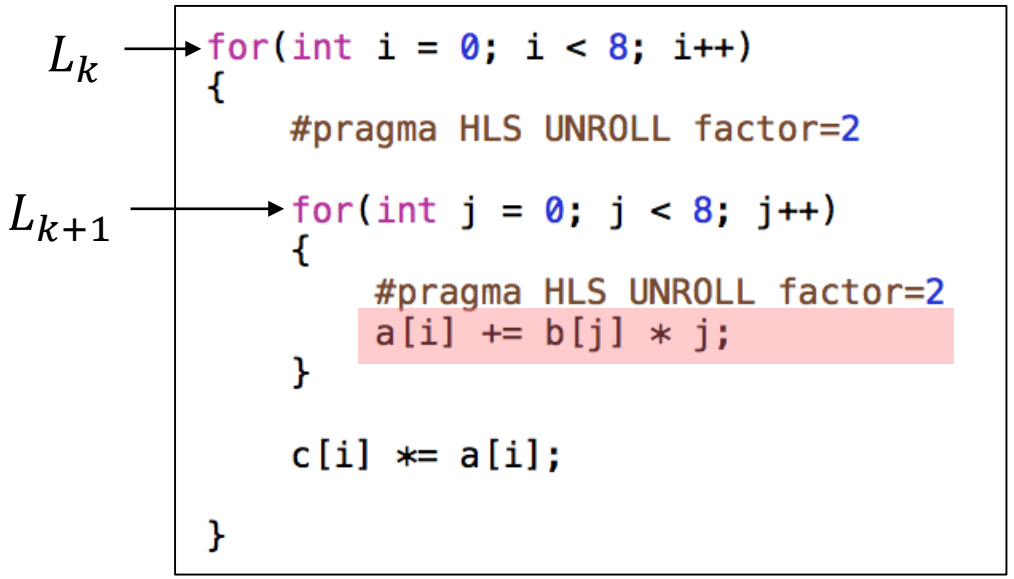
$C_{L_k}^{U_k}$

ion latency: $C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k}$

op latency: $Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}$

❷ Analytical Models: Performance Model

1. Loop Unrolling



$C_{L_{k+1}}^{U_{k+1}}$

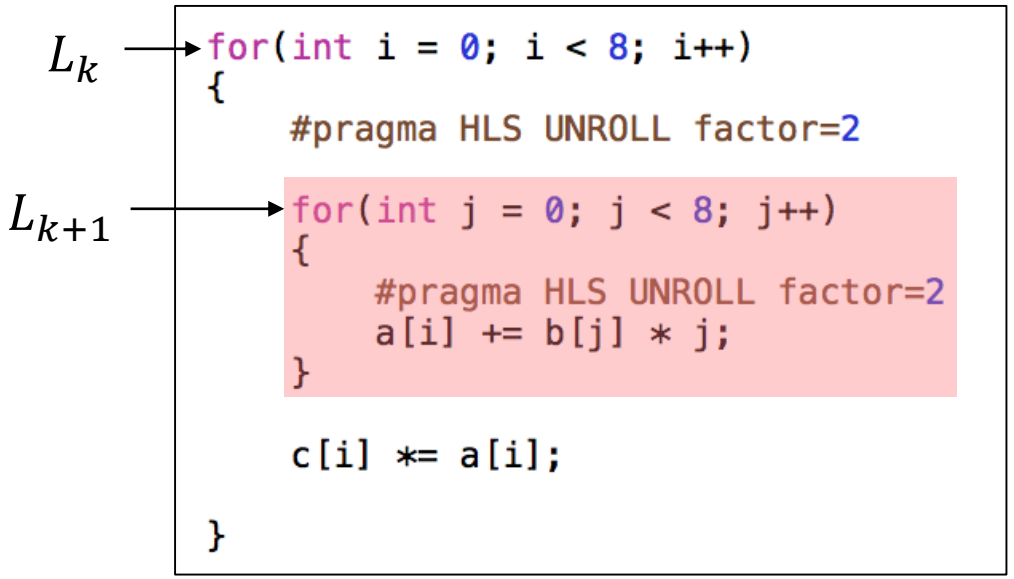
Iteration latency of L_{k+1}

Iteration latency: $$C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k}$$

Loop latency: $$Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}$$

❷ Analytical Models: Performance Model

1. Loop Unrolling



$$C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}}$$

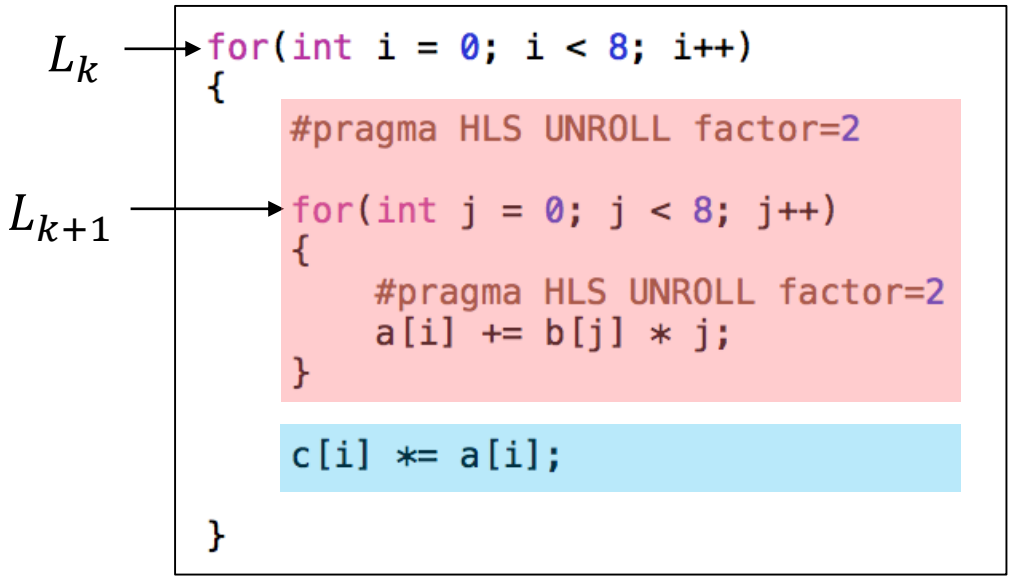
Loop latency of L_{k+1}

Iteration latency: $$C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k}$$

Loop latency: $$Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}$$

❷ Analytical Models: Performance Model

1. Loop Unrolling



$C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k$

Latency of U_k copies

$C_{L_k \setminus L_{k+1}}^{U_k}$

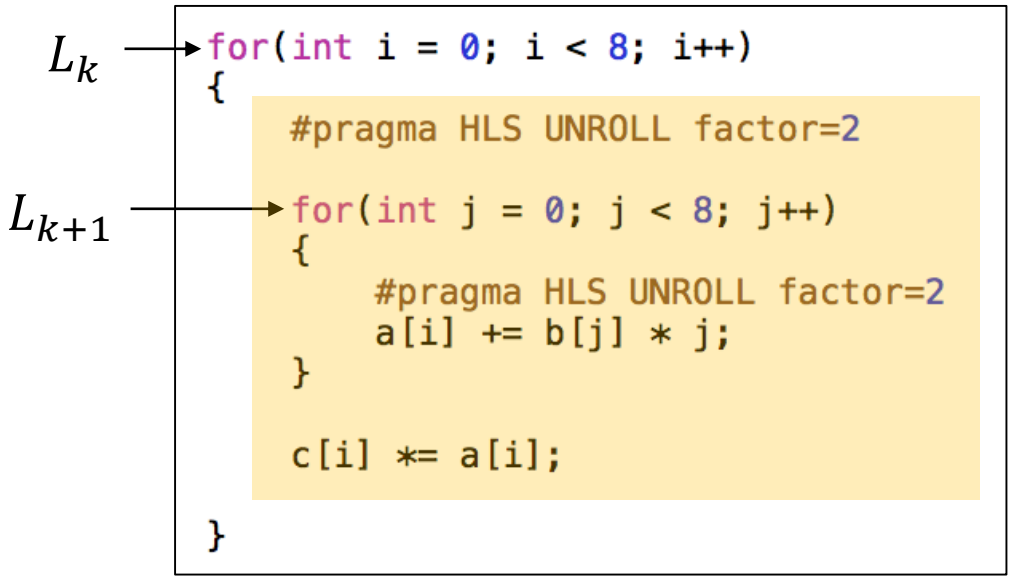
Latency of $L_k \setminus L_{k+1}$

Iteration latency: $$C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k}$$

Loop latency: $$Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}$$

❷ Analytical Models: Performance Model

1. Loop Unrolling



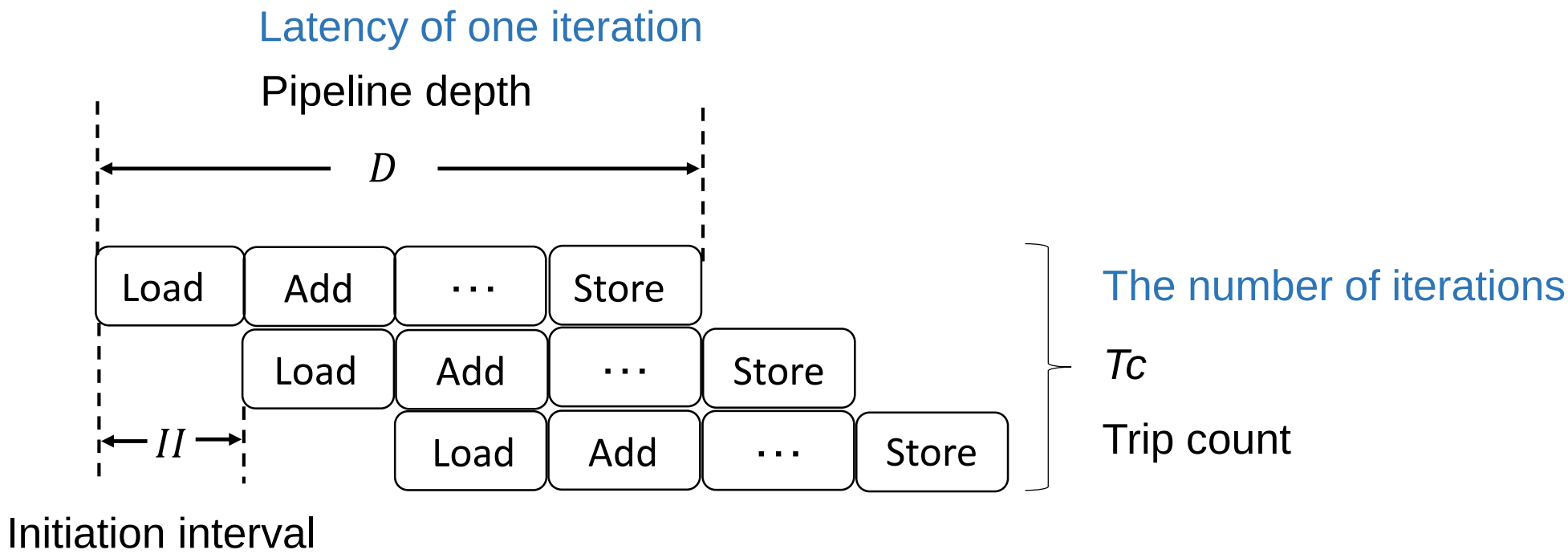
$C_{L_k}^{U_k}$

Iteration latency: $$C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k}$$

Loop latency: $$Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}$$

❷ Analytical Models: Performance Model

2. Loop Pipelining



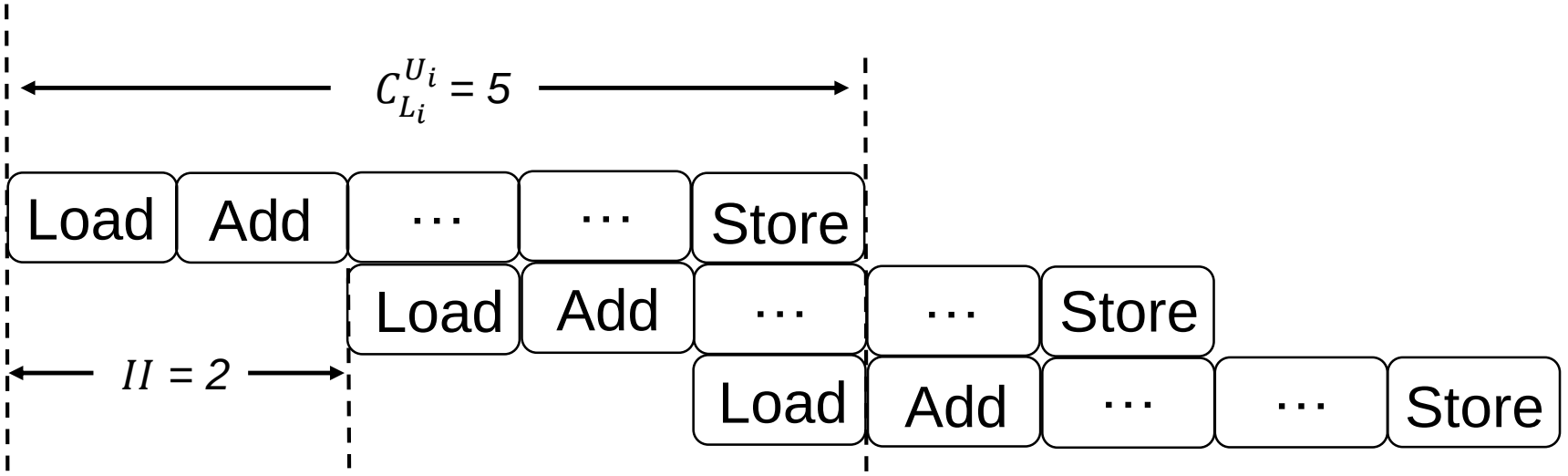
Latency of a pipelined loop: $Cycle_L = D + II \cdot (Tc - 1)$

❷ Analytical Models: Performance Model

2. Loop Pipelining: pipeline depth, $Cycle_{L_i} = \boldsymbol{D_i} + II_i \cdot (Tc - 1)$

Load & Store operations: access **different** arrays, $D_i = C_{L_i}^{U_i}$

access the **same** array, $D_i = \left\lceil \frac{C_{L_i}^{U_i}}{II_i} \right\rceil \cdot II_i$

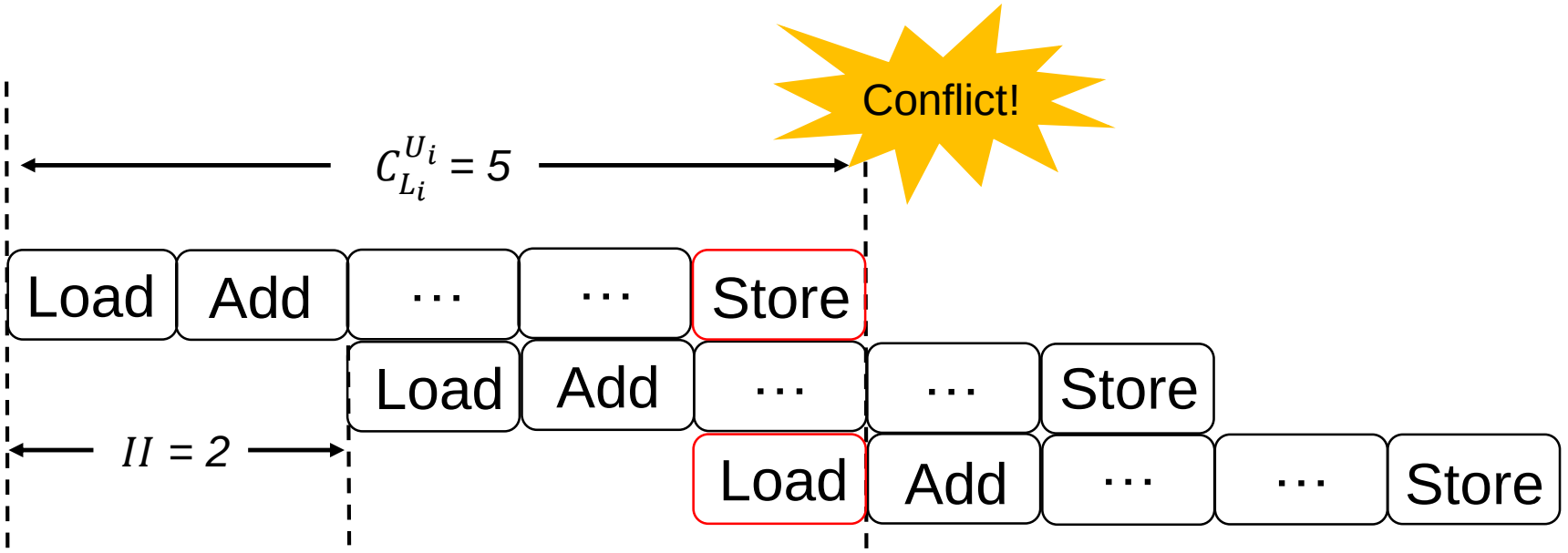


❷ Analytical Models: Performance Model

2. Loop Pipelining: pipeline depth, $Cycle_{L_i} = \boldsymbol{D_i} + II_i \cdot (Tc - 1)$

Load & Store operations: access **different** arrays, $D_i = C_{L_i}^{U_i}$

access the **same** array, $D_i = \left\lceil \frac{C_{L_i}^{U_i}}{II_i} \right\rceil \cdot II_i$

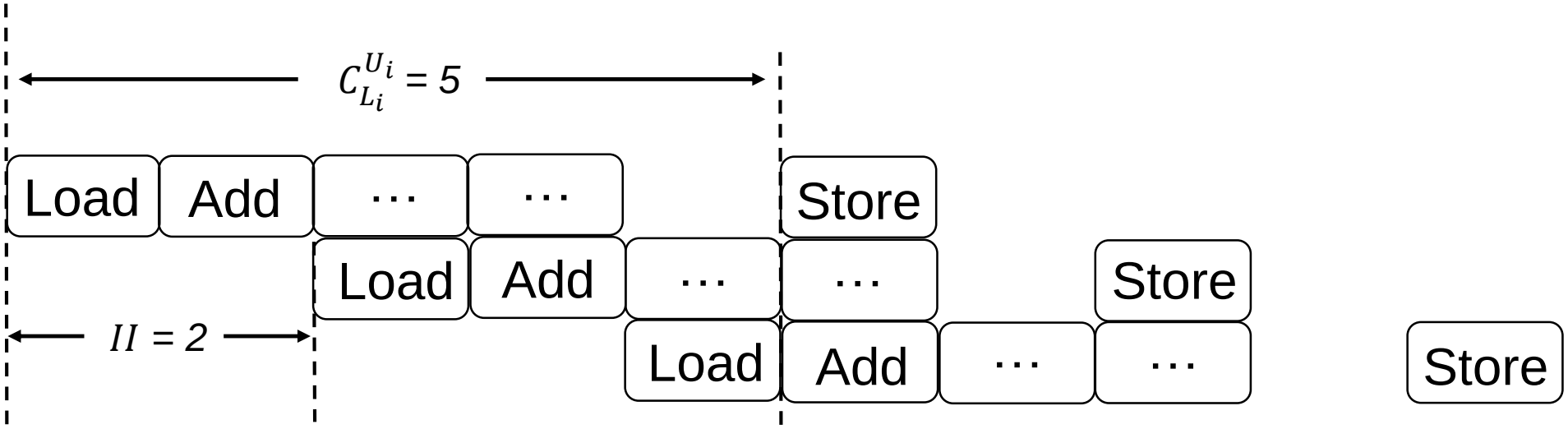


❷ Analytical Models: Performance Model

2. Loop Pipelining: pipeline depth, $Cycle_{L_i} = \boldsymbol{D_i} + II_i \cdot (Tc - 1)$

Load & Store operations: access **different** arrays, $D_i = C_{L_i}^{U_i}$

access the **same** array, $D_i = \left\lceil \frac{C_{L_i}^{U_i}}{II_i} \right\rceil \cdot II_i$

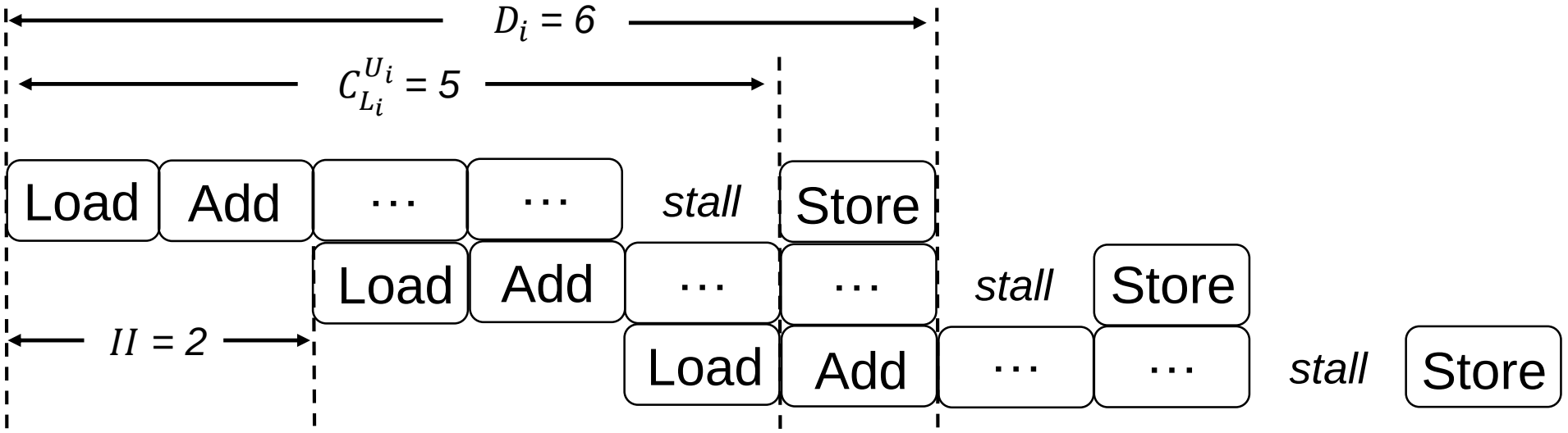


❷ Analytical Models: Performance Model

2. Loop Pipelining: pipeline depth, $Cycle_{L_i} = \boldsymbol{D_i} + II_i \cdot (Tc - 1)$

Load & Store operations: access **different** arrays, $D_i = C_{L_i}^{U_i}$

access the **same** array, $D_i = \left\lceil \frac{C_{L_i}^{U_i}}{II_i} \right\rceil \cdot II_i$



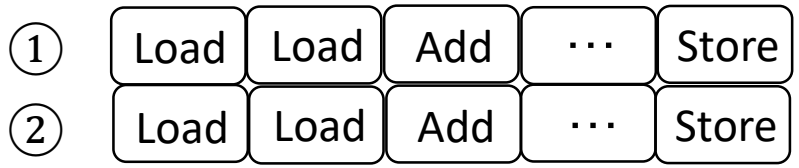
❷ Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, $Cycle_{L_i} = D_i + \boldsymbol{II_i} \cdot (Tc - 1)$

$$II_{i,\min} = \max\left(II^{res}_{i,\min}, II^{rec}_{i,\min}\right)$$

Resource-constrained II

$$II^{res}_{i,\min} = \max_{m}\left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil\right)$$



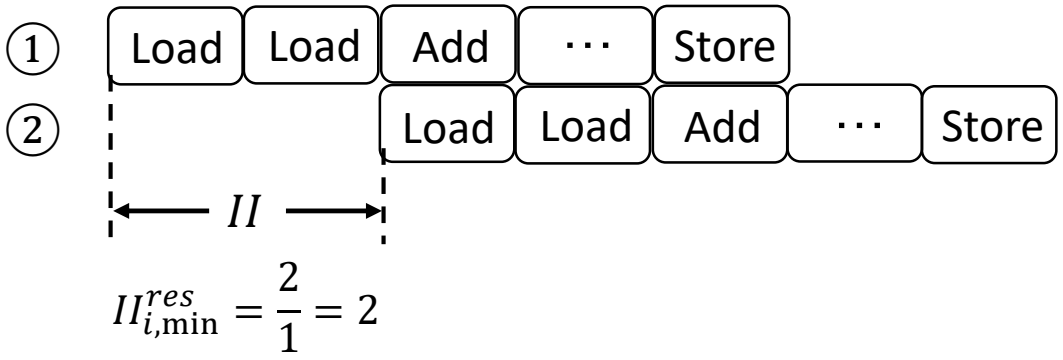
❷ Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, $Cycle_{L_i} = D_i + \boldsymbol{II_i} \cdot (Tc - 1)$

$$II_{i,\min} = \max\left(II^{res}_{i,\min}, II^{rec}_{i,\min}\right)$$

Resource-constrained II

$$II^{res}_{i,\min} = \max_m \left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)$$



$$II^{res}_{i,\min} = \frac{2}{1} = 2$$

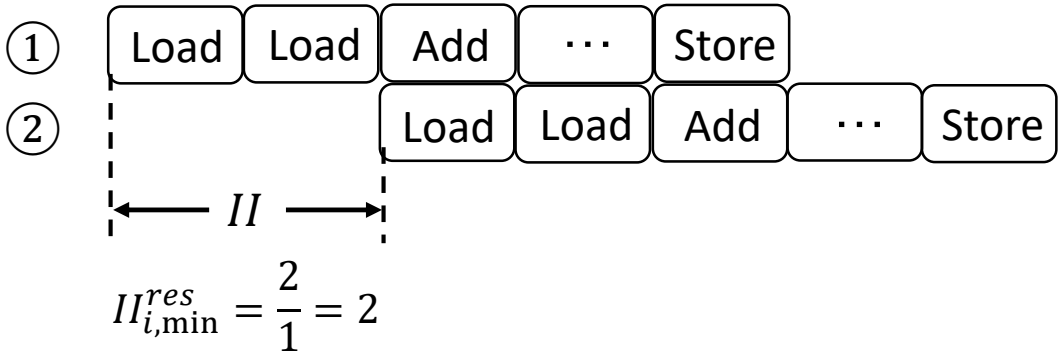
❷ Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, $Cycle_{L_i} = D_i + \boldsymbol{II_i} \cdot (Tc - 1)$

$$II_{i,\min} = \max\left(II^{res}_{i,\min}, II^{rec}_{i,\min}\right)$$

Resource-constrained II

$$II^{res}_{i,\min} = \max_m \left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)$$



$$II^{res}_{i,\min} = \frac{2}{1} = 2$$

Recurrence-constrained II

$$II^{rec}_{i,\min} = \max_p \left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right)$$

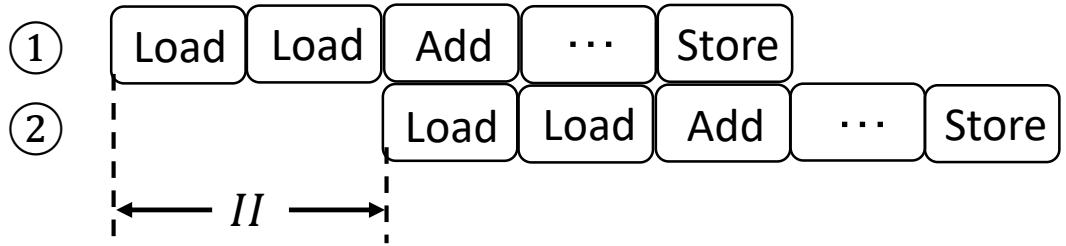
❷ Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, $Cycle_{L_i} = D_i + \boldsymbol{II_i} \cdot (Tc - 1)$

$$II_{i,\min} = \max\left(II^{res}_{i,\min}, II^{rec}_{i,\min}\right)$$

Resource-constrained II

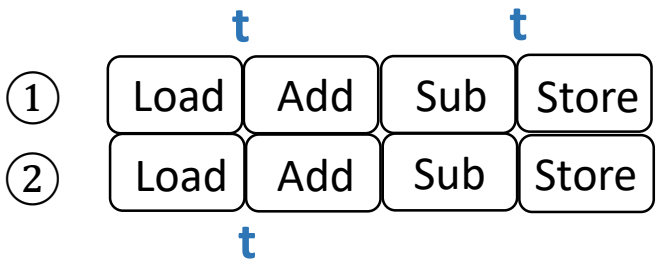
$$II^{res}_{i,\min} = \max_m \left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)$$



$$II^{res}_{i,\min} = \frac{2}{1} = 2$$

Recurrence-constrained II

$$II^{rec}_{i,\min} = \max_p \left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right)$$



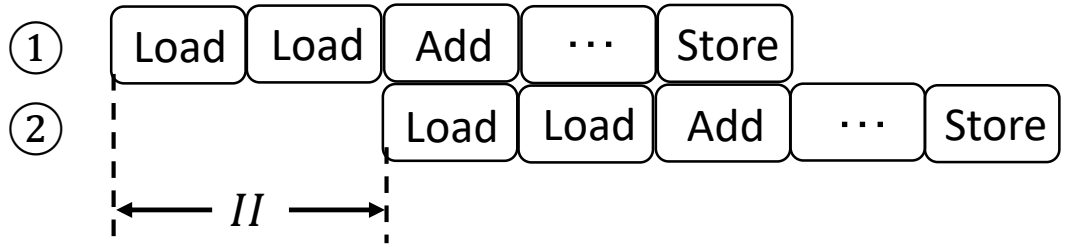
❷ Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, $Cycle_{L_i} = D_i + \boldsymbol{II_i} \cdot (Tc - 1)$

$$II_{i,\min} = \max\left(II^{res}_{i,\min}, II^{rec}_{i,\min}\right)$$

Resource-constrained II

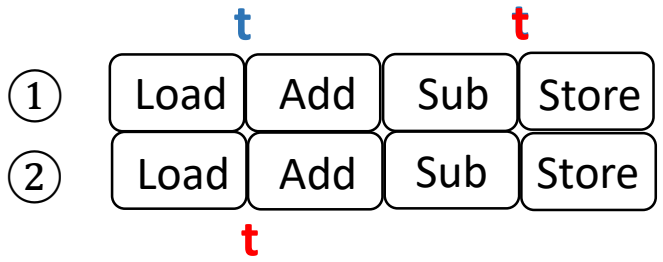
$$II^{res}_{i,\min} = \max_m \left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)$$



$$II^{res}_{i,\min} = \frac{2}{1} = 2$$

Recurrence-constrained II

$$II^{rec}_{i,\min} = \max_p \left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right)$$



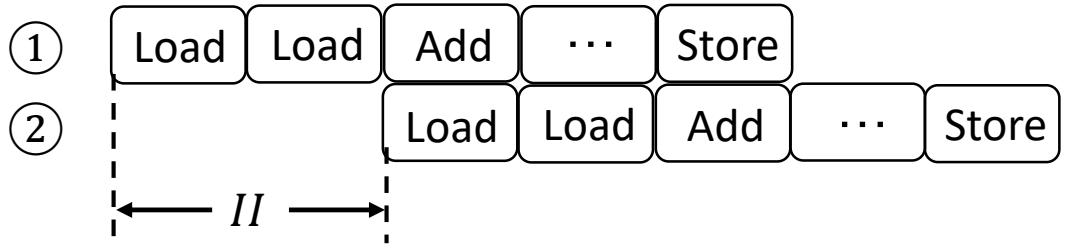
❷ Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, $Cycle_{L_i} = D_i + \boldsymbol{II_i} \cdot (Tc - 1)$

$$II_{i,\min} = \max\left(II^{res}_{i,\min}, II^{rec}_{i,\min}\right)$$

Resource-constrained II

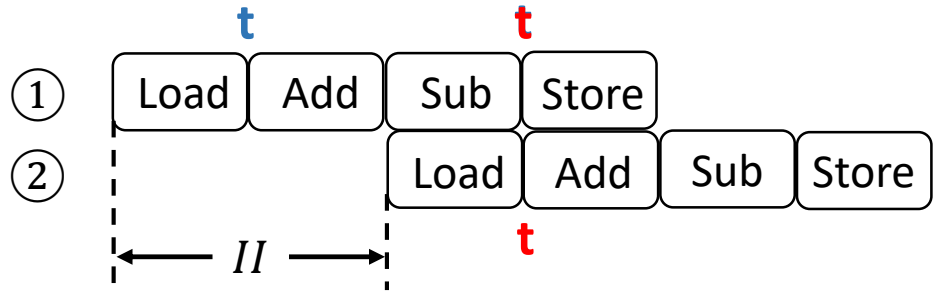
$$II^{res}_{i,\min} = \max_m \left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)$$



$$II^{res}_{i,\min} = \frac{2}{1} = 2$$

Recurrence-constrained II

$$II^{rec}_{i,\min} = \max_p \left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right)$$



$$II^{rec}_{i,\min} = \frac{2}{1} = 2$$

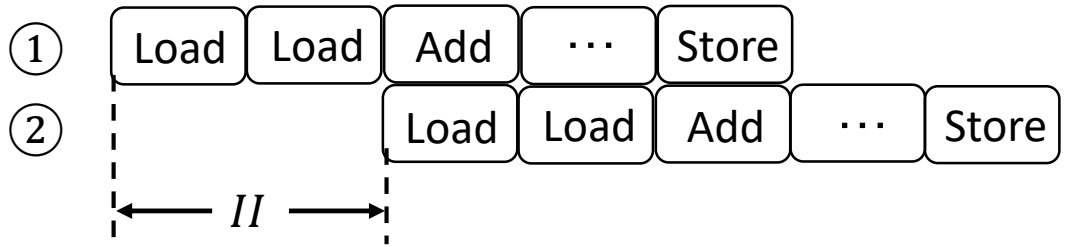
❷ Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, $Cycle_{L_i} = D_i + \boldsymbol{II_i} \cdot (Tc - 1)$

$$II_{i,\min} = \max\left(II^{res}_{i,\min}, II^{rec}_{i,\min}\right)$$

Resource-constrained II

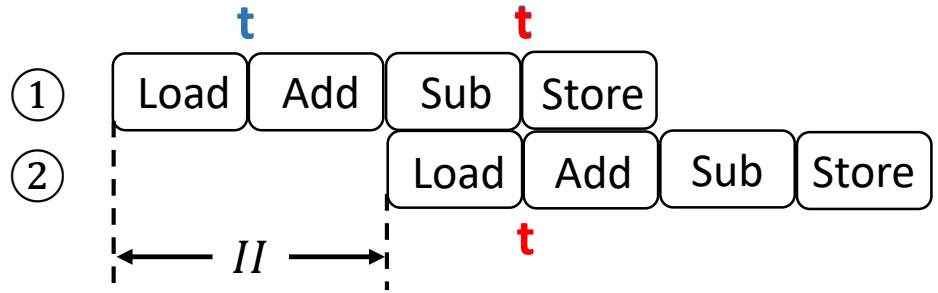
$$II^{res}_{i,\min} = \max_m \left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)$$



$$II^{res}_{i,\min} = \frac{2}{1} = 2$$

Recurrence-constrained II

$$II^{rec}_{i,\min} = \max_p \left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right)$$



$$II^{rec}_{i,\min} = \frac{2}{1} = 2$$

If the loop contains sub-functions, $II_i = \max\left(II_{i,\min}, II_{sub,\max}\right)$

❷ Analytical Models: Performance Model

2. Loop Pipelining: trip count, $Cycle_{L_i} = D_i + II_i \cdot (\boldsymbol{Tc} - 1)$

Perfect nested loop:

```
for(int i = 0; i < 8; i++)
{
    for(int j = 0; j < 8; j++)
    {
        #pragma HLS PIPELINE II=1
        #pragma HLS UNROLL factor=2
        a[j] += b[j] * j;
    }

}
```

Outer loops are **flattened** to feed the inner loop with more data.

$$\boldsymbol{Tc} = \frac{B_i}{U_i} \cdot B_{i-1}B_{i-2} \cdots B_k$$
$$= \frac{8}{2} \cdot 8 = 32$$

Non-perfect nested loop:

```
for(int i = 0; i < 8; i++)
{
    for(int j = 0; j < 8; j++)
    {
        #pragma HLS PIPELINE II=1
        #pragma HLS UNROLL factor=2
        a[i] += b[j] * j;
    }
    c[i] *= a[i];
}
```

Codes between loop statements **stop** outer loops from **flattening** with the inner loop.

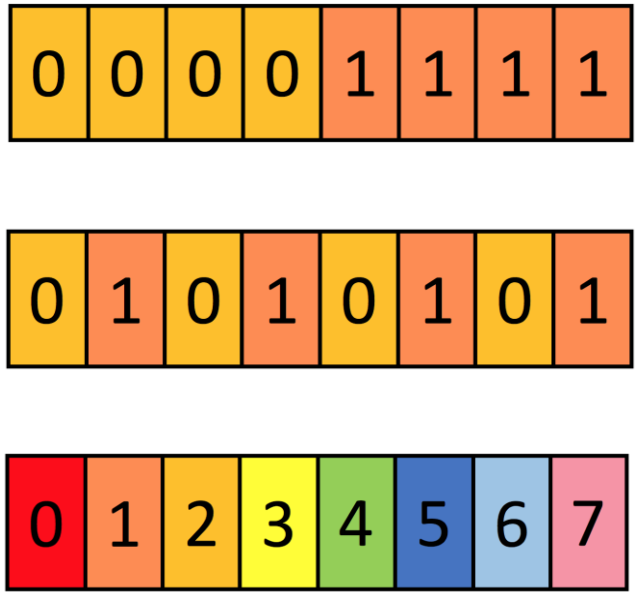
$$\boldsymbol{Tc} = \frac{B_i}{U_i} = \frac{8}{2} = 4$$

❷ Analytical Models: Performance Model

3. Array Partitioning

Partition number
in dimension i

Partition number
considering n-dimension array partitioning



Block: $P_i = \lfloor index_i / \lceil size_i / f_i \rceil \rfloor$

Cyclic: $P_i = (index_i) \bmod (f_i)$

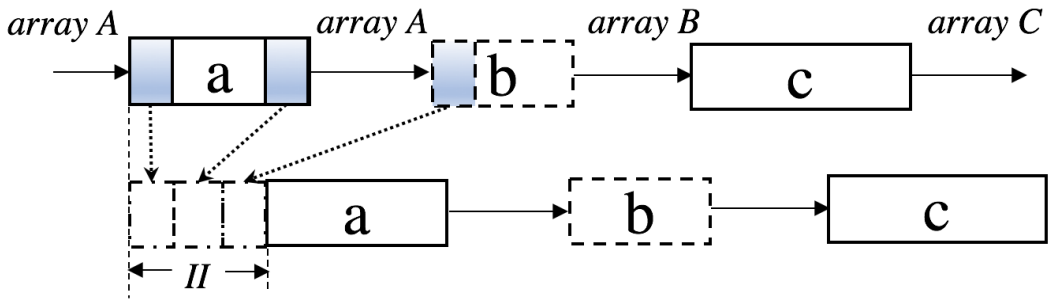
Complete: $P_i = index_i$

$$P = P_1 + \sum_{i=2}^{n} (P_i \cdot \prod_{k=1}^{i-1} f_k)$$

Partition number shows which partition this array element is located in.

❷ Analytical Models: Performance Model

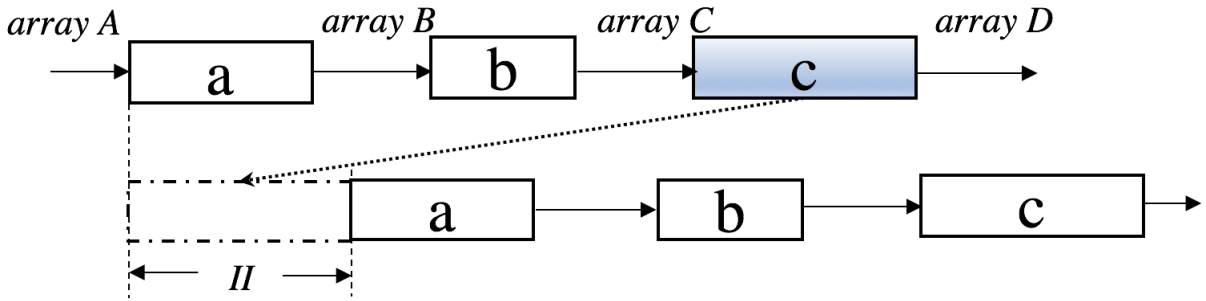
4. Function Pipelining "Fine-grain" pipelining: operators



$$II = \max\left(II^{res}_{\min}, II^{sub}_{\max}\right)$$

- II^{res}_{min}: **resource-constrained** minimum II,
- II^{sub}_{max}: the maximum function II among all **sub-functions**.

5. Dataflow "Coarse-grain" pipelining: functions and loops



$$II = II^{sub}_{\max} = \max_{i}\left(II^{sub}_{i}\right)$$

- II^{sub}_{max} is the maximum II among all **sub-functions** and **sub-loops**.

❷ Analytical Models: Resource Model

1. DSP Estimation

- Sharable: the **maximum** number of **parallel** operators
- If a loop is **pipelined**, $N_{\min}^{op} = \left\lceil \frac{N_{op}}{II} \right\rceil$

2. BRAM Estimation

$$R_{bram} = \left\lceil \frac{\#bits}{width} \right\rceil \cdot \left\lceil \frac{\#element}{depth} \right\rceil \cdot \#partition \cdot d$$

- R_{bram}: the number of **blocks** on BRAM.

❷ Analytical Models: Resource Model

1. DSP Estimation

- Sharable: the **maximum** number of **parallel** operators
- If a loop is **pipelined**, $N_{\min}^{op} = \left\lceil \frac{N_{op}}{II} \right\rceil$

2. BRAM Estimation

$$R_{bram} = \left\lceil \frac{\#bits}{width} \right\rceil \cdot \left\lceil \frac{\#element}{depth} \right\rceil \cdot \#partition \cdot d$$

Width ratio

Depth ratio

- *#bit*: the width of each array element. e.g., int: 32 bits
- *width* is the **bandwidth** of the selected block configuration.
- *#element*: the number of **elements** in one memory partition.
- *depth* is the **depth** of the selected block configuration.

❷ Analytical Models: Resource Model

1. DSP Estimation

- Sharable: the **maximum** number of **parallel** operators
- If a loop is **pipelined**, $N_{\min}^{op} = \left\lceil \frac{N_{op}}{II} \right\rceil$

2. BRAM Estimation

$$R_{bram} = \left\lceil \frac{\#bits}{width} \right\rceil \cdot \left\lceil \frac{\#element}{depth} \right\rceil \cdot \#partition \cdot d$$

Width ratio

Depth ratio

- *#partition*: the number of memory partitions.
- d: equals 2 if dataflow is applied; equals 1 if not.

Framework

1. RDC
2. Models
3. **MGDSE**

uided Design Space Exploration

ge: Redundancy Elimination

e **redundant** design points based on the rules of HLS tools.

tion: pipelined,
s: unrolled completely, cannot be pipelined.
lated to LU, LP will be removed.

stage: Guided Search

he performance of the **current** design point

the **next** design point

ation **metrics** are proposed

❸ Metric-Guided Design Space Exploration

1. The **first** stage: Redundancy Elimination

- To **remove** the **redundant** design points based on the rules of HLS tools.

  E.g., *top-function: pipelined,*
  sub-loops: unrolled completely, cannot be pipelined.
  Points related to LU, LP will be removed.

2. The **second** stage: Guided Search

- To **evaluate** the performance of the **current** design point
- To **determine** the **next** design point

➡ **Three** evaluation **metrics** are proposed

3 Metric-Guided Design Space Exploration

Three evaluation metrics:

$$M_{diff} = C^{sub}_{\max} - C^{sub}_{\text{s,max}}$$

- Find the **bottleneck**: the **longest** sub-function/sub-loop is assumed to have the **greatest** influence.

$$M_{res} = \max\left(\frac{DSP_{used}}{DSP_{total}}, \frac{BRAM_{used}}{BRAM_{total}}\right)$$

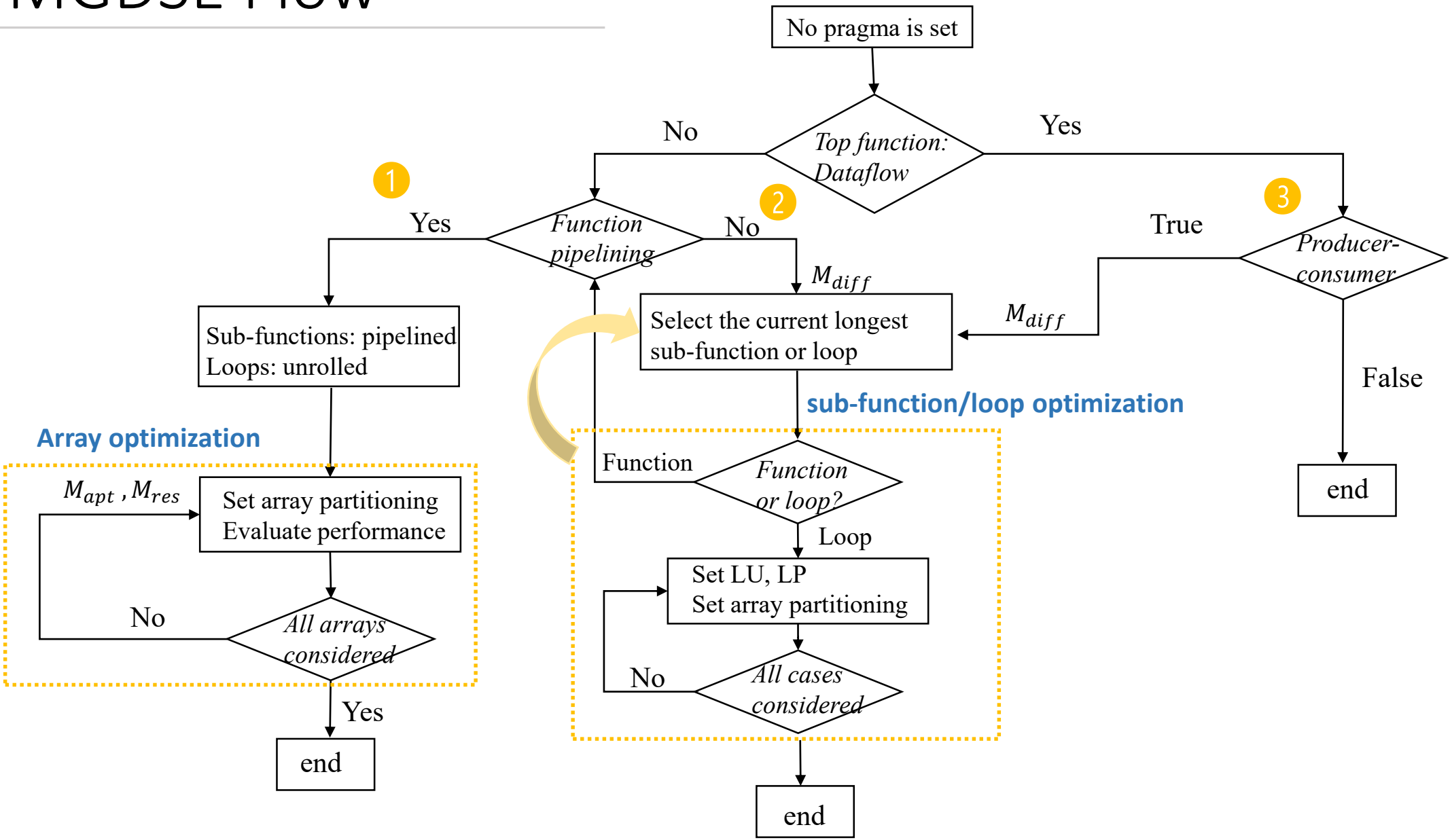
- Check the **resource** constraints: whether the resource usage exceeds the available resources on FPGA.

$$M_{apt,l} = \frac{\#loads}{\max\limits_{j,k}(index^j_i - index^k_i + 1)}$$

$$M_{apt,s} = \frac{\#stores}{\max\limits_{j,k}(index^j_i - index^k_i + 1)}$$

- Decide how to **partition**: which option is better, block or cyclic.

MGDSE Flow


No pragma is set
Top function: Dataflow
No
Yes
1
Yes
Function pipelining
2
No
M_{diff}
3
True
Producer-consumer
False
M_{diff}
Select the current longest sub-function or loop
sub-function/loop optimization
Sub-functions: pipelined
Loops: unrolled
Array optimization
M_{apt} , M_{res}
Set array partitioning
Evaluate performance
No
All arrays considered
Yes
end
Function
Function or loop?
Loop
Set LU, LP
Set array partitioning
No
All cases considered
end
end

Framework

1. RDC
2. Models
3. MGDSE

Results

n Accuracy

om Polybench

	Lin_analyzer [1] (%)	COMBA(%)
ATAX	2.68	0.71
BICG	1.24	0.33
GEMM	3.25	0.46
GESUMMV	5.15	1.25
MM	2.69	0.83
MVT	2.05	0.48
SYR2K	1.32	0.47
SYRK	2.78	0.44

➢ The estimation **error** is **reduced!**

plications

	COMBA (%)
JPEG	1.54
Seidel	0.91
Rician	1.12

➢ COMBA also **works** for **more complicated** applications!

DSE Results: Comparison

Benchmark	Design Space		Performance Speed-up		MGDSE Time (s)	
	Lin-analyzer[1]	COMBA	Lin-analyzer[1]	COMBA	Lin-analyzer[1]	COMBA
ATAX	85	**1.31×10^8**	8.35	**125.45**	8.85	**41.01**
BICG	95	**5.76×10^8**	15.88	**201.38**	22.60	**89.20**
GEMM	85	**1.05×10^{10}**	8.15	**261.63**	185.37	**65.83**
GESUMMV	85	**8.39×10^8**	15.42	**83.88**	12.05	**77.40**
MM	85	**6.34×10^{13}**	15.22	**277.03**	161.37	**292.15**
MVT	95	**1.05×10^{10}**	15.30	**189.18**	14.88	**57.72**
SYR2K	85	**1.05×10^{10}**	7.27	**123.96**	250.01	**223.00**
SYRK	85	**1.64×10^8**	8.12	**462.65**	168.78	**86.34**

Larger design space

Better speed-up

Efficient design space exploration

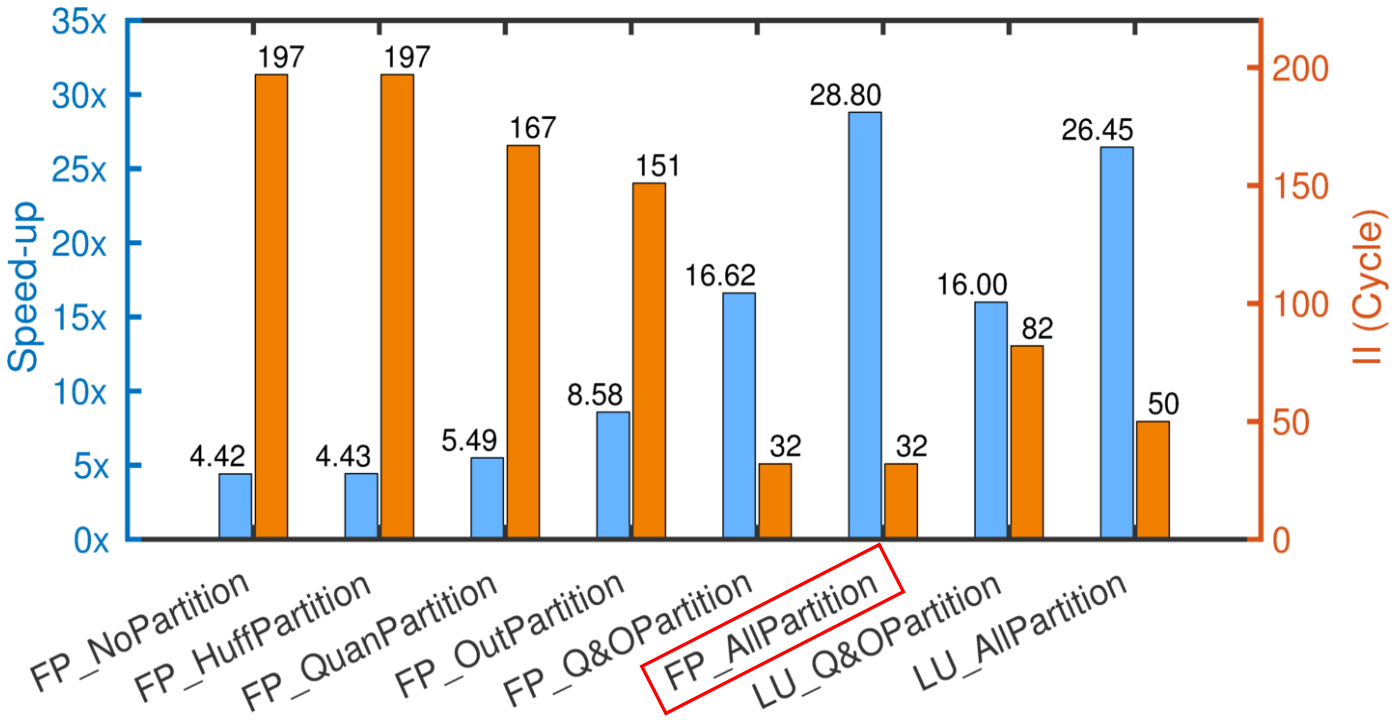
DSE Results: Optimizations of Polybench

Benchmark	Array size	Optimizations		
		Pipeline	Unroll	Partition (array name: (type, factor, dimension))
ATAX	16	top-loop	2	A:(block,8,2); x:(complete,16,1); y:(block,8,1); tmp:(not partitioned)
BICG	32	top-loop	2	A:(block,16,2) (block,2,1); s:(block,16,1); p:(block,8,1); r,q:(not partitioned)
GEMM	16	top-loop	2	B:(block,8,2) (block,2,1); C:(cyclic,2,2); A:(not partitioned)
GESUMMV	16	top-loop	2	A,B:(block,8,2) (block,2,1); x:(complete,16,1); y,tmp:(not partitioned)
MM	8	top-loops(at the same level)	2,2	A,D,tmp:(complete,8,2); B,C:(complete,8,2) (complete,8,1)
MVT	16	top-function	16,16	y1,y2:(complete,16,1); A:(block,8,2) (block,2,1); x1,x2:(not partitioned)
SYR2K	16	top-loop	1	A,B:(complete,16,2); C:(not partitioned)
SYRK	16	top-loop	4	A,C:(complete,16,2)

For different code structures, **the optimization schemes** are different!

Case Study: JPEG

```
void decode_block (int Quant[64], int Out[64], int Huff[64]){
    IZigzagMatrix(Huff, Quant); //one loop with bound 64
    IQuantize(Quant);           //one loop with bound 64
    ChenIDct(Quant,Out);        //three loops with bounds 8,8,64
    PostshiftIDctMatrix(Out);   //one loop with bound 64
    BoundIDctMatrix(Out);       //one loop with bound 64
}
```



Design space: 7.61×10^{12}

MGDSE time: 613s

Speed-up: 28.8x

The optimal configuration found by COMBA: "***FP_AllPartition***", which is to

- **pipeline** the top-function,
- **partition** the three arrays **completely**.

Conclusion

The proposed model-based framework, COMBA:

- **Optimizes** different applications with **various** code structures.
- Performs **better** than the state-of-the-art on simple benchmarks.
- Optimizes **more complicated** applications efficiently, **within minutes.**

Our tool is available at: *https://github.com/zjru/COMBA*

Thank you for listening!

Q & A