# Array Scalarization In High Level Synthesis

Preeti Ranjan Panda[1] , **Namita Sharma**[1],
Arun Kumar Pilania[2], G. Krishnaiah[2],
Sreenivas Subramoney[2] and Ashok Jagannathan[2]

[1]Indian Institute of Technology Delhi
[2]Intel Technology India Pvt. Ltd.

# What is Array Scalarization?

- Arrays are usually mapped to memories (SRAM) in High Level Synthesis
  - Performance bottlenecks due to memory port constraints.
- Scalarization
  - Transforming an array into a group of scalar variables.
  - Entire array is stored in discrete registers.

***Question: How to automatically decide when to Scalarize?***

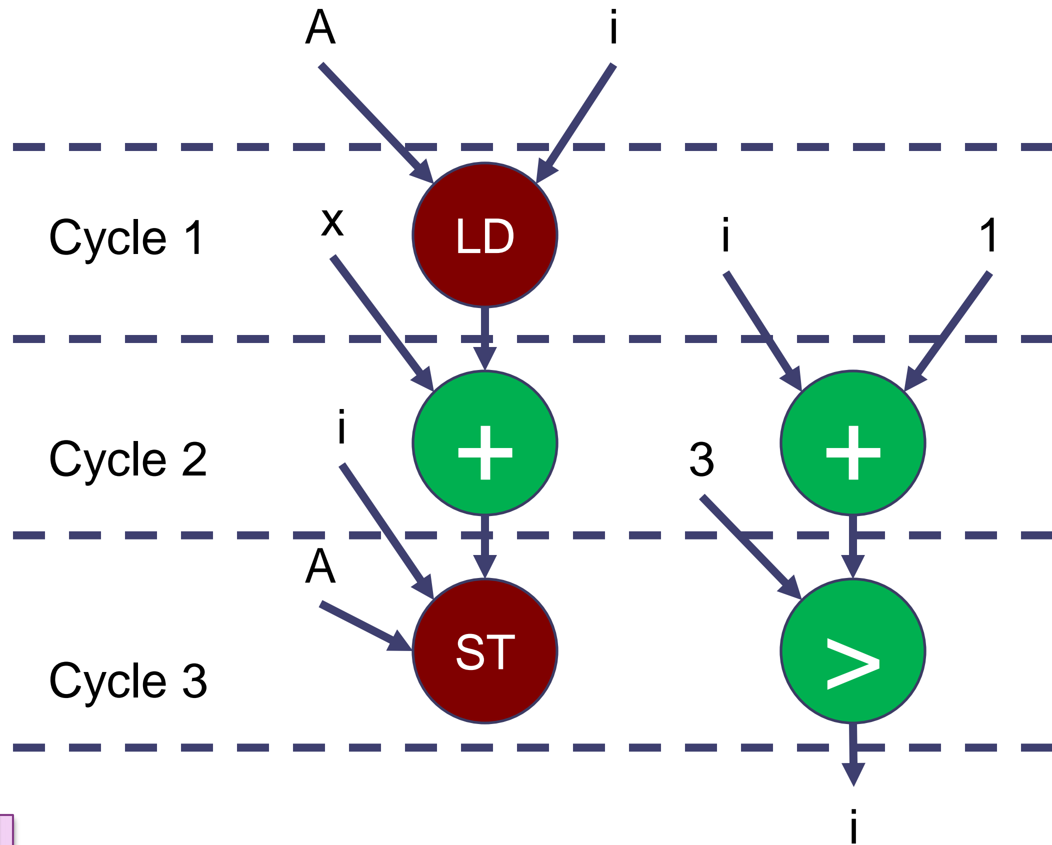
P. R. Panda et al., ASPDAC 2014

# Previous Work

- Commercial tools (Cadence, Synopsys) allow user pragmas for controlling scalarization
- In this paper: ***Automatic strategy for Scalarization***

# Illustration

*for (i = 0; i < 4; i++)*
*A[i] = A[i] + x;*

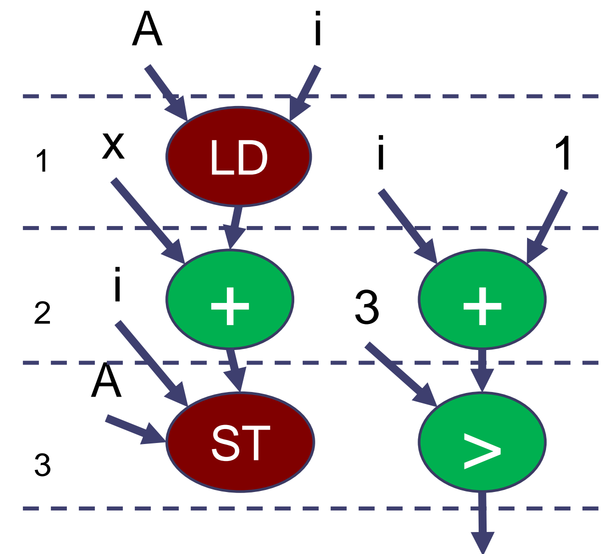



Latency = 4 x 3 = 12 cycles

P. R. Panda et al., ASPDAC 2014

# Loop Unrolling

```
for (i = 0; i < 4; i++)
  A[i] = A[i] + x;
```




Unroll (Resource: 2-port Mem)

```
A[0] = A[0] + x;
A[1] = A[1] + x;
A[2] = A[2] + x;
A[3] = A[3] + x;
```

Cycle 1

Cycle 2

Cycle 3

Cycle 4

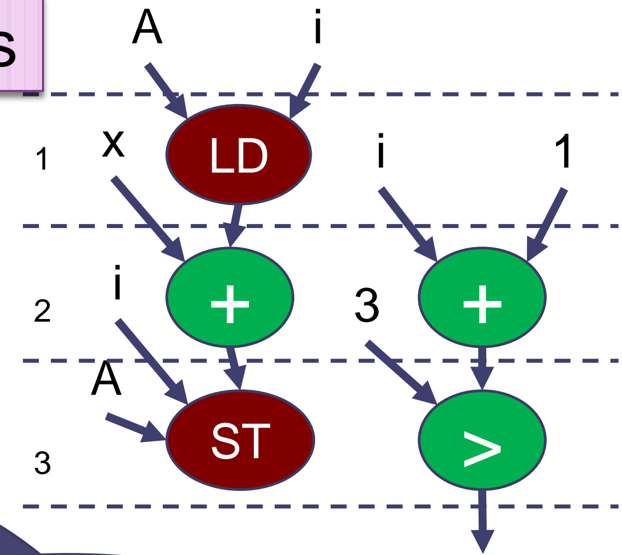
Latency = 4 cycles

P. R. Panda et al., ASPDAC 2014

# Scalarization

Latency = 12 cycles

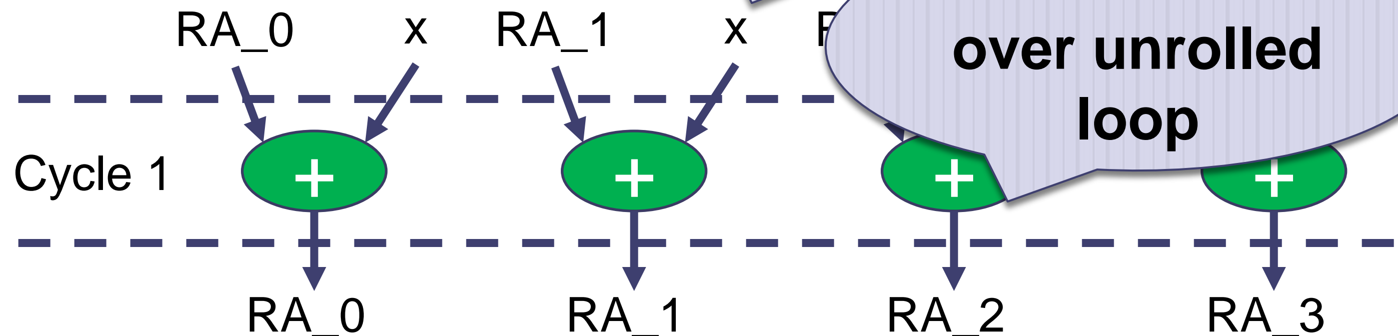
```
for (i = 0; i < 4; i++)
  A[i] = A[i] + x;
```




**Unroll + Scalarization**

92% reduction over rolled loop

```
RA_0 = RA_0 + x;
RA_1 = RA_1 + x;
RA_2 = RA_2 + x;
RA_3 = RA_3 + x;
```




Latency = 1cycle

P. R. Panda et al., ASPDAC 2014

# What makes **ARRAY SCALARIZATION** Difficult?

# Challenges

- Area Overheads
  - Discrete Registers => Extra area
  - **a, b** have same area
  - Scalarizing **b** results in more latency reduction

```
for (i = 0; i < 128; i++)
  a[i] = a[i] * p;
for (j =0; j < 128; j++)
  b[j] = b[j] << s;
for (k = 0; k < 128; k++)
  b[k] = b[k] + v;
```

- Sparse Array Accesses
  - large area penalty
  - not much performance gain

```
int MetricMap [128];
...
curr = MetricMap [3];
```

P. R. Panda et al., ASPDAC 2014

# Challenges

- Loop Carried Dependencies and Resource Constraints
  - Dependence => Sequentialization
  - Not much benefit from Scalarization

```
for (i = 0; i < 50; i++)
  A[i] = A[i] + A[i-1] * 7;
```

- Data dependent array indexing
  - Scalarizing **a** useful only if **b** is also scalarized
  - Scalarizing **b** is expensive
    - data dependent access => large 100-to-1 MUX connecting to all registers

```
int a[250], b[100];
...
for (i = 0; i < 150; i++) {
  index = calc (i) % 100;
  val = b[index];
  a [i] = a[i] << val;
}
```

P. R. Panda et al., ASPDAC 2014

# Problem Definition

Given a behavioral description with **loops** $L_1$**, L**$_2$**, ..., L**$_n$ accessing **arrays** $A_1$**, ...,**$A_m$, with

- each **loop** $L_j$ **accessing a subset** $S_j$ of the arrays
- array size for $A_i$ is Size($i$)
- an allowed area overhead constraint $OV$

compute the **Scalarization Vector** $SV$ **[1...**$m$**]**

$SV$ [$i$] = TRUE means $A_i$ is scalarized and = FALSE means $A_i$ is not scalarized

such that the overall latency (cycle count) reduction is maximized while the total area overhead < $OV$.

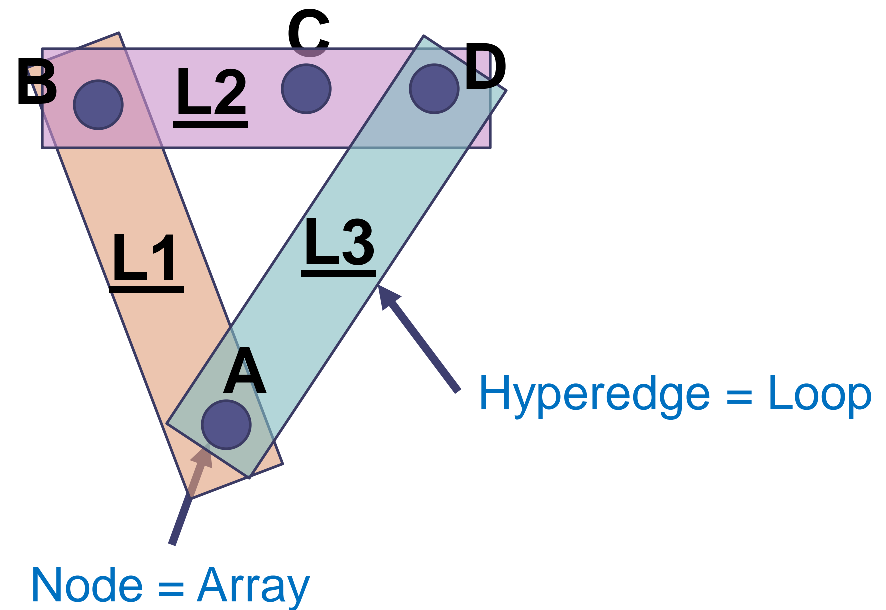
# APPROACH

# Array Access Graph

*L1*: for(i = 0; i<50; i++)
A[i] = B[i] + 3;

*L2*: for(i = 0; i<50; i++)
C[i] = D[i] + B[i];

*L3*: for(i = 0; i<50; i++)
D[i] = A[i] * 4;




P. R. Panda et al., ASPDAC 2014

# Cost Estimation

- Cost of scalarizing an array comprises of area overheads due to
  - Storing the array elements in discrete registers instead of SRAM cells

$$Storage_Area(i) = Size(i) * ElementSize(i) * (RArea - SArea)$$

  - MUX costs due to scalarization
    - Due to resource sharing by the scalarized registers = *Size(i)/R*
    - Data dependent accesses = *Size(i)*

$$Cost(A_i) = Storage_Area(i) + MUX_Area(i)$$

$$Cost(L_i) = \sum Cost(A_j),\ A_j \in S_i$$

P. R. Panda et al., ASPDAC 2014

# Latency Estimation

- Compute the ***Initiation Interval (II)*** for the loop which is a function of
  - Resource constraints
  - Loop carried dependencies

$$II = \max\left(\max\left[\frac{ops(i)*lat(i)}{FU(i)}\right]\forall i, \max\left[\frac{dep(t)}{it(t)}\right]\forall t\right)$$

$$Lat(i) = Body(i) + II*(iter(i)-1)$$

- Kurra et. al. , “ The impact of loop unrolling on controller delay in high level synthesis”, DATE 2007.

P. R. Panda et al., ASPDAC 2014

# Priority Function

$$\text{Priority}(L_i) = \frac{\text{Red}(L_i)}{\text{Cost}(L_i)}$$

$$\text{Red}(L_i) = \text{Body}(i) * \text{iter}(i) - \text{Lat}(i)$$

$$\text{Cost}(L_i) = \sum \text{Cost}(A_i), \quad A_j \in S_i$$

# Approach Overview

Begin

Build the **array access graph**

∀ **Edges**

Compute **Cost ($L_i$), Red($L_i$), Priority($L_i$)**
**Update** Candidate List ***CL***

***CL***!=ϕ && OV > 0

False

End

True

$E_i$ = Candidate List ***CL*** → ***head***

**OV > Cost ($L_i$)**

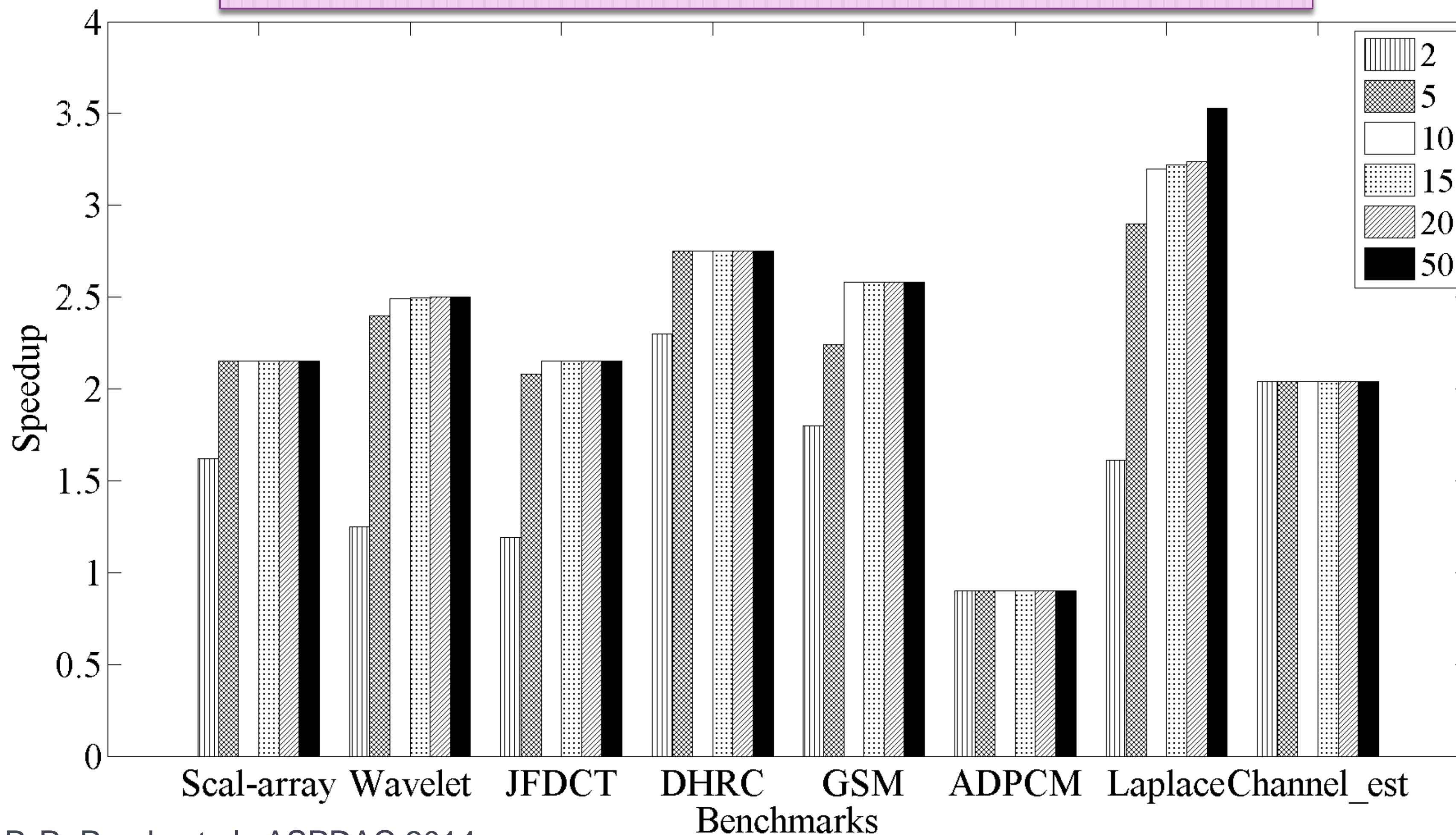
False

True

**SV[j]= TRUE**
**Update** Cost ($L_j$), Priority($L_j$) ∀ j
Candidate List ***CL & OV***

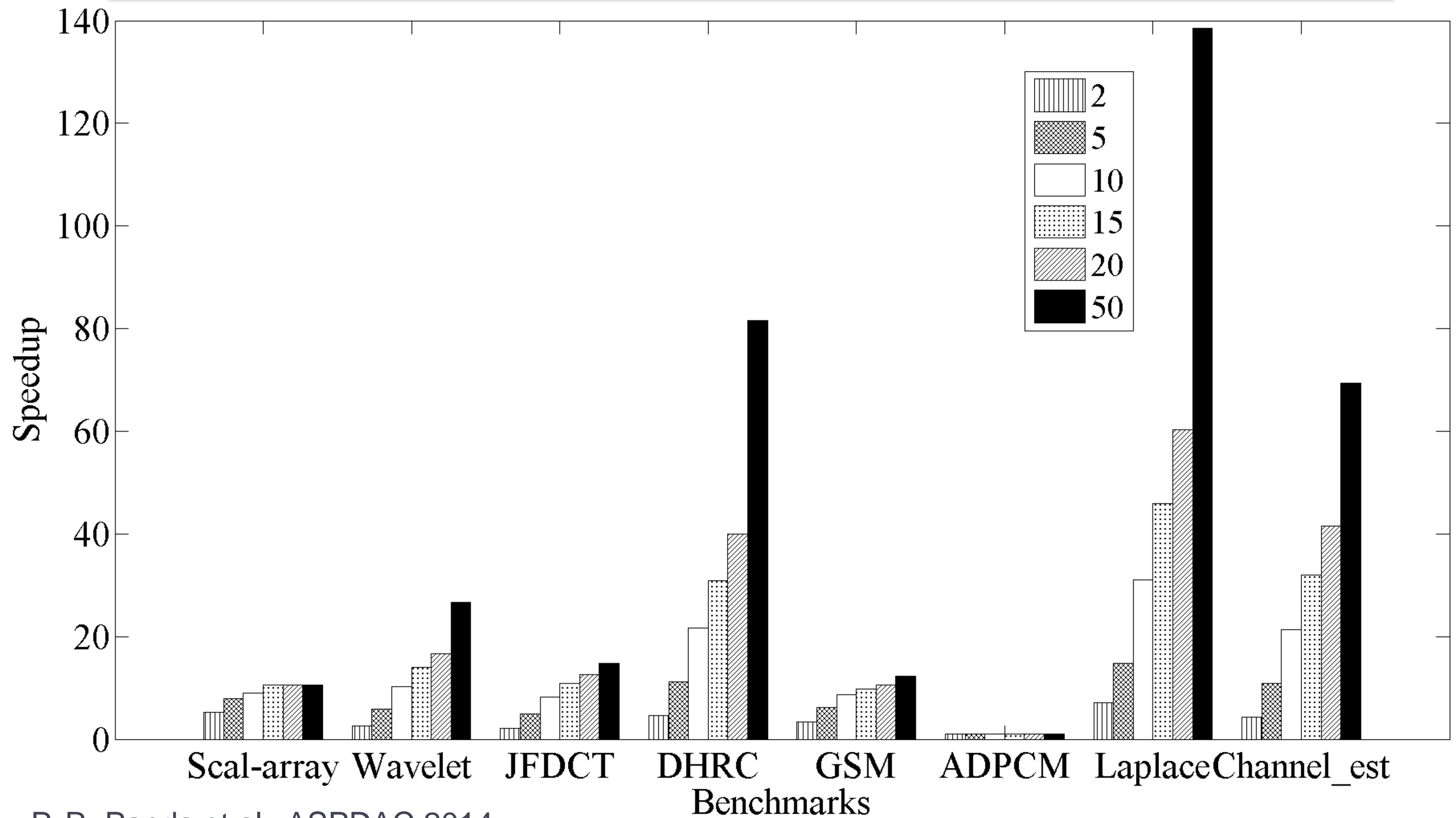
P. R. Panda et al., ASPDAC 2014

# Experimental Results



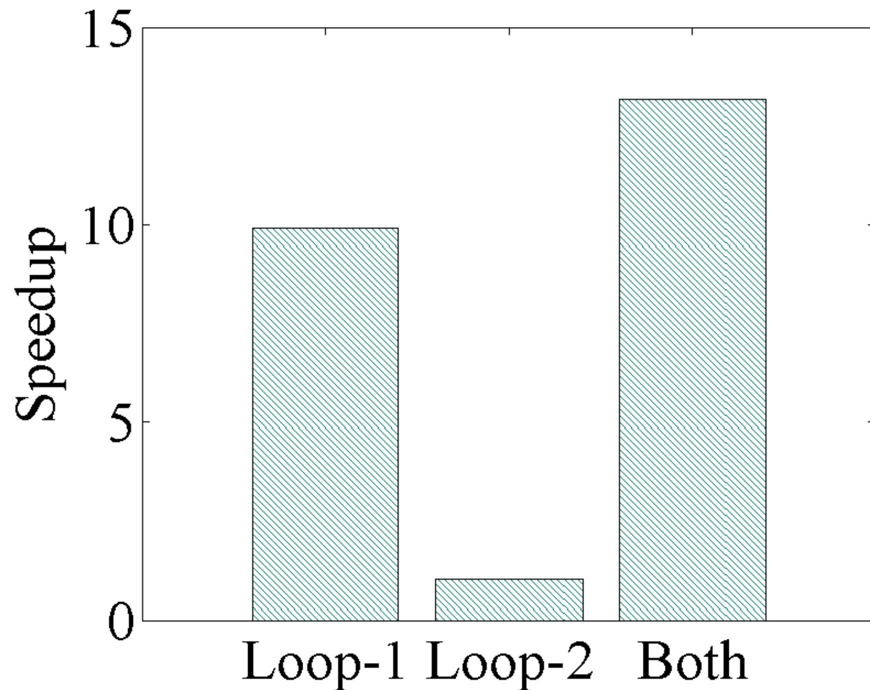

P. R. Panda et al., ASPDAC 2014

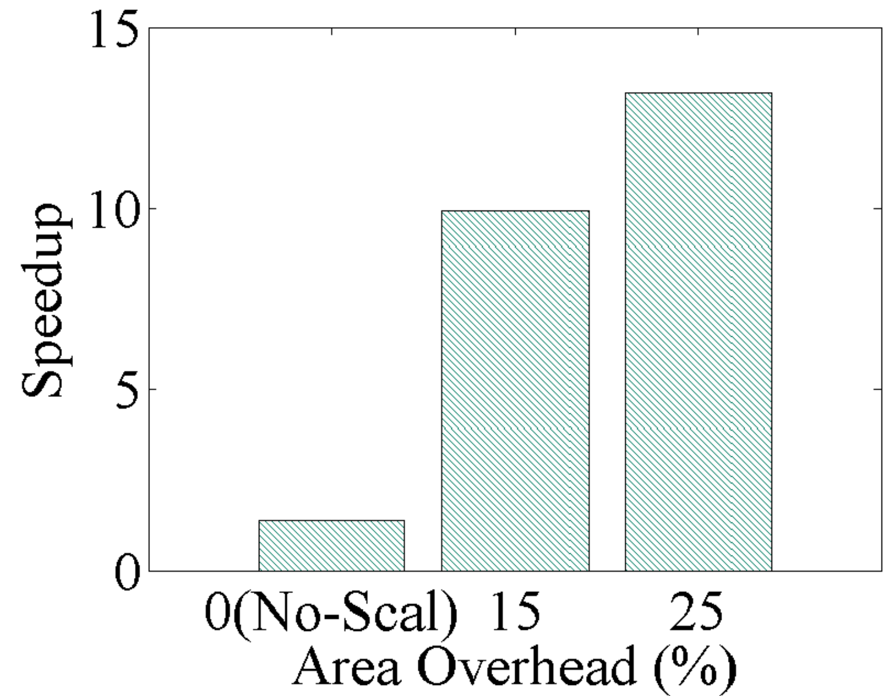
# Experimental Results




P. R. Panda et al., ASPDAC 2014

# Experimental Results




P. R. Panda et al., ASPDAC 2014

# Conclusions

- Investigated *Array Scalarization*
  - a behavioral synthesis optimization
  - selectively transforms arrays accessed in loops to scalars
    - performance  improvement under  area overhead constraint.
- Automated strategy
  - prioritize the loops based on:
    - expected performance benefits
    - resulting area overhead

P. R. Panda et al., ASPDAC 2014

# Thank You!