An Out-of-Order Load-Store Queue for Spatial Computing

Lana Josipović, Philip Brisk, Paolo Ienne

École Polytechnique Fédérale de Lausanne
University of California, Riverside

October 2017
High-level Synthesis and Static Scheduling

- **High-level synthesis (HLS)** tools are the future of reconfigurable computing
  - Design circuits from high-level programming languages

- Classic HLS relies on **static schedules**
  - Each operation executes at a cycle fixed at synthesis time

- Scheduling dictated by compile-time information
  - Maximum parallelism when memory accesses are provably independent
  - Pessimistic assumptions when dependencies might occur
Overcoming the Limitations of Static Scheduling

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
Overcoming the Limitations of Static Scheduling

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```
Overcoming the Limitations of Static Scheduling

for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

- Static scheduling (any HLS tool)
  - Inferior when memory accesses cannot be disambiguated at compile time

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

RAW

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

RAW
Overcoming the Limitations of Static Scheduling

for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

• Static scheduling (any HLS tool)
  • Inferior when memory accesses cannot be disambiguated at compile time

Dynamic scheduling
  • Maximum parallelism: Only serialize memory accesses on actual dependencies
Dynamically Scheduled Circuits

- Dataflow, latency-insensitive, elastic \(^1, ^2, ^3\)
  - Each operation executes as soon as all of its inputs arrive
  - Operators pass on the data they produce

2. Cortadella et al. Synthesis of synchronous elastic architectures. DAC '06.
Dynamically Scheduled Circuits

- Dataflow, latency-insensitive, elastic $^{1,2,3}$
  - Each operation executes as soon as all of its inputs arrive
  - Operators pass on the data they produce

$^1$ Carloni et al. Theory of latency-insensitive design. TCAD '01.

$^2$ Cortadella et al. Synthesis of synchronous elastic architectures. DAC '06.

Dynamically Scheduled Circuits

- Dataflow, latency-insensitive, elastic \(^1, ^2, ^3\)
  - Each operation executes as soon as all of its inputs arrive
  - Operators pass on the data they produce

---

\(^1\) Carloni et al. Theory of latency-insensitive design. TCAD '01.

\(^2\) Cortadella et al. Synthesis of synchronous elastic architectures. DAC '06.

\(^3\) Vijayaraghavan et al. Bounded dataflow networks and latency-insensitive circuits. MEMOCODE '09.
Dynamically Scheduled Circuits

- Dataflow, latency-insensitive, elastic $^{1, 2, 3}$
- Each operation executes as soon as all of its inputs arrive
- Operators pass on the data they produce

1 Carloni et al. Theory of latency-insensitive design. TCAD '01.
2 Cortadella et al. Synthesis of synchronous elastic architectures. DAC '06.
Dynamically Scheduled Circuits

- Dataflow, latency-insensitive, elastic $^{1,2,3}$
  - Each operation executes as soon as all of its inputs arrive
  - Operators pass on the data they produce

What about memory?

---

$^1$ Carloni et al. Theory of latency-insensitive design. TCAD '01.
$^2$ Cortadella et al. Synthesis of synchronous elastic architectures. DAC '06.
Processor LSQ
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
Processor LSQ

loop:

lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
loop:

1. `lw $t1, 0($t0)`
2. `lw $t2, 0($t1)`
3. `mul $t2, $t2, $t3`
4. `sw $t2, 0($t0)`
5. `addi $t1, $t1, 4`
6. `bne $t5, $t1, loop`
Processor LSQ

loop:  
  lw  $t1, 0($t0)  
  lw  $t2, 0($t1)  
  mul  $t2, $t2, $t3  
  sw  $t2, 0($t0)  
  addi  $t1, $t1, 4  
  bne  $t5, $t1, loop
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
loop:  
lw  $t1, 0($t0)  
lw  $t2, 0($t1)  
mul  $t2, $t2, $t3  
sw  $t2, 0($t0)  
addi  $t1, $t1, 4  
bne  $t5, $t1, loop
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
loop:
  lw $t1, 0($t0)
  lw $t2, 0($t1)
  mul $t2, $t2, $t3
  sw $t2, 0($t0)
  addi $t1, $t1, 4
  bne $t5, $t1, loop
loop:  
  lw $t1, 0($t0)  
  lw $t2, 0($t1)  
  mul $t2, $t2, $t3  
  sw $t2, 0($t0)  
  addi $t1, $t1, 4  
  bne $t5, $t1, loop  

Processor LSQ

Processor datapath

LSQ
  LD b[0]
  LD a[b[0]]
  ST a[0], ?
  LD b[1]
  LD a[b[1]]
  ST a[1], ?
  LD b[2]
  LD a[b[2]]
  ST a[2], ?
Processor LSQ

loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
Processor LSQ

loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
loop: lw $t1, 0($t0)
    lw $t2, 0($t1)
    mul $t2, $t2, $t3
    sw $t2, 0($t0)
    addi $t1, $t1, 4
    bne $t5, $t1, loop
Processor LSQ

loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
Processor LSQ

loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
Processor LSQ

loop: lw $t1, 0($t0)
     lw $t2, 0($t1)
     mul $t2, $t2, $t3
     sw $t2, 0($t0)
     addi $t1, $t1, 4
     bne $t5, $t1, loop
loop:  
  lw $t1, 0($t0)  
  lw $t2, 0($t1)  
  mul $t2, $t2, $t3  
  sw $t2, 0($t0)  
  addi $t1, $t1, 4  
  bne $t5, $t1, loop
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

const

RD a[b[i]]

WR a[i]

LD b[0]

LD a[b[0]]

LD b[1]
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

LD b[0]
LD a[b[0]]
LD b[1]
LD a[b[1]]
LD b[2]
LD a[b[2]]
...

LSQ
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit:
- RD b[i]
- const
- RD a[b[i]]
- *
- WR a[i] (3)

LSQ:
- LD b[0]
- LD a[b[0]]
- LD b[1]
- LD a[b[1]]
- LD b[2]
- LD a[b[2]]
- ...
- ST a[0], ?
- ST a[1], ?
- ST a[2], ?
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

RD b[i]

RD a[b[i]]

const

* 

WR a[i]

LSQ

LD b[0]

LD a[b[0]]

LD b[1]

LD a[b[1]]

LD b[2]

LD a[b[2]]

...

ST a[0], ?

ST a[1], ?

ST a[2], ?

C0

C1

C2

C3

C4

C5

C6

C7

C8

C9

C10

C11

C12

C13

C14

C15

C16

C17

rd b[0]
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

LD b[0]
LD a[5]
LD b[1]
LD a[b[1]]
LD b[2]
LD a[b[2]]
...
ST a[0], ?
ST a[1], ?
ST a[2], ?
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

RD b[i]

cost

RD a[b[i]]

*

WR a[i]

LSQ

LD b[0]
LD a[5]
LD b[1]
LD a[b[1]]
LD b[2]
LD a[b[2]]

...:

ST a[0], ?
ST a[1], ?
ST a[2], ?

C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16 C17

rd b[0] rd a[b[0]]
rd b[1]
rd b[2]
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

LD b[0]
LD a[5]
LD b[1]
LD a[b[2]]
LD a[6]
LD b[2]
LD a[b[2]]

LSQ

ST a[0], ?
ST a[1], ?
ST a[2], ?

rd b[0]
rd a[b[0]]
rd b[1]
rd b[2]
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

RD b[i]

RD a[b[i]]

const

* 

WR a[i]

LSQ

LD b[0]
LD a[5]
LD b[1]
LD a[6]
LD b[2]
LD a[b[2]]
...
ST a[0], ?
ST a[1], ?
ST a[2], ?
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

- **RD b[i]**
- **RD a[b[i]]**
- **const**
- **WR a[i]**

LSQ

- LD b[0]
- LD a[5]
- LD b[1]
- LD a[6]
- LD b[2]
- LD a[1]
-...
- ST a[0], ?
- ST a[1], ?
- ST a[2], ?

Timeline:

- C0
  - rd b[0]
  - rd a[b[0]]
- C1
  - rd b[1]
  - rd a[b[1]]
- C2
  - rd b[2]
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit

Dataflow circuit

LD b[0]
LD a[5]
LD b[1]
LD a[6]
LD b[2]
LD a[1]

ST a[0], 9
ST a[1], ?
ST a[2], ?

Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit
Using a Processor LSQ in a Spatial Circuit

**Dataflow circuit**

- RD b[i]
- RD a[b[i]]
- const
- *
- WR a[i]

**LSQ**

- LD b[0]
- LD a[5]
- LD b[1]
- LD a[6]
- LD b[2]
- LD a[1]
- ...
- ST a[0], 9
- ST a[1], 4
- ST a[2]

RAW dependency not honored!
Inadequacy of Processor LSQs

- Traditional processor LSQs allocate memory instructions in program order

```
loop: lw $t1, 0($t0)
      lw $t2, 0($t1)
      mul $t2, $t2, $t3
      sw $t2, 0($t0)
      addi $t1, $t1, 4
      bne $t5, $t1, loop
```
Inadequacy of Processor LSQs

- Traditional processor LSQs allocate memory instructions in program order

  loop: lw $t1, 0($t0)
  lw $t2, 0($t1)
  mul $t2, $t2, $t3
  sw $t2, 0($t0)
  addi $t1, $t1, 4
  bne $t5, $t1, loop

- Spatial circuits do not have instructions or a program order

How to supply program order to the LSQ?
Outline

- Introduction
- Program order in dynamic circuits
- LSQ implementation
- Evaluation
- Conclusion
Typical LSQ Organization

Generic LSQ entry:
1. Opcode (LD or ST)
2. Memory address to access
3. Data
4. Completion flag
Generic LSQ entry:
1. Opcode (LD or ST)
2. Memory address to access
3. Data
4. Completion flag

Requirement for execution:
The entry does not depend on any not yet completed entries earlier in the queue (closer to the head)
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```plaintext
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
```
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];

RAW

<table>
<thead>
<tr>
<th>H</th>
<th>LD</th>
<th>ST</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
<td>0</td>
</tr>
</tbody>
</table>
‘Lazy’ Allocation

• Allocate when address/data of a memory access is known
• Force allocation requests to arrive in program order

for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```plaintext
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];  
1: b[1]=6 → rd a[6]; wr a[1];  
2: b[2]=1 → rd a[1], wr a[2];
‘Lazy’ Allocation

- Allocate when address/data of a memory access is known
- Force allocation requests to arrive in program order

Limited advantages of dynamic scheduling and of the LSQ\(^1,\)\(^2\)

---

\(^1\) Budiu et al. Dataflow: A complement to superscalar. ISPASS ’05.

‘Eager’ Allocation

• Statically allocate all entries before execution


‘Eager’ Allocation

- Statically allocate all entries before execution

```plaintext
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

Control flow needs to be statically knowable
Unrealistically large LSQ
### Overview of Allocation Strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Applicable to any program</th>
<th>Fast out-of-order execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Lazy’ allocation</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>‘Eager’ allocation</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>?</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

Groups correspond to basic blocks.
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];

RAW
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

```
0: b[0]=5 \rightarrow rd a[5]; \text{wr} a[0];
1: b[1]=6 \rightarrow rd a[6]; \text{wr} a[1];
2: b[2]=1 \rightarrow rd a[1]; \text{wr} a[2];
```
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

```plaintext
0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
```
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

- Case 0: b[0]=5 → rd a[5]; wr a[0];
- Case 1: b[1]=6 → rd a[6]; wr a[1];
- Case 2: b[2]=1 → rd a[1], wr a[2];
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];

Diagram showing the allocation process and code snippet.
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made
- Order of accesses within a group is statically determinable

<table>
<thead>
<tr>
<th>H</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD @</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD @</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST @</td>
<td>?</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD @</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD ?</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST @</td>
<td>?</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD @</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD ?</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST @</td>
<td>?</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```plaintext
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

```
0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
```
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made
- Order of accesses within a group is statically determinable

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made
- Order of accesses within a group is statically determinable

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made
- Order of accesses within a group is statically determinable

```c
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1], wr a[2];
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```plaintext
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

0: \( b[0] = 5 \rightarrow \text{rd } a[5]; \text{wr } a[0]; \)
1: \( b[1] = 6 \rightarrow \text{rd } a[6]; \text{wr } a[1]; \)
2: \( b[2] = 1 \rightarrow \text{rd } a[1]; \text{wr } a[2]; \)
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

- $b[0]=5 \rightarrow$ read $a[5]$; write $a[0]$;

Diagram showing allocation and access order.
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made.
- Order of accesses within a group is statically determinable.

```plaintext
for (i=0; i<N; i++) {
    a[i] = const * a[b[i]];
}
```

```
0: b[0]=5 → rd a[5]; wr a[0];
1: b[1]=6 → rd a[6]; wr a[1];
2: b[2]=1 → rd a[1]; wr a[2];
```
Group Allocation

- Allocate groups of accesses as soon as a control flow decision is made
- Order of accesses within a group is statically determinable

Fast out-of-order execution
Applicable to any program
Outline

- Introduction
- Program order in dynamic circuits
- LSQ implementation
- Evaluation
- Conclusion
LSQ Implementation

- Overall structure:
LSQ Implementation

- Overall structure:

As in a processor LSQ
LSQ Implementation

- Overall structure:
Group Allocator

- The group allocator allocates entries and ensures allocation respects the sequential program order
The group allocator allocates entries and ensures allocation respects the sequential program order.

ROM outputs:
- Number of loads, number of stores
Group Allocator

- The group allocator allocates entries and ensures allocation respects the sequential program order

ROM outputs:
- Number of loads, number of stores
- LD/ST ordering
Group Allocator

- The group allocator allocates entries and ensures allocation respects the sequential program order.

ROM outputs:
- Number of loads, number of stores
- LD/ST ordering
- LD/ST port connectivity
Group Allocator

- The group allocator allocates entries and ensures allocation respects the sequential program order
The group allocator allocates entries and ensures allocation respects the sequential program order.

- Group Allocator

ROM

Allocate: 2 LDs + 1 ST
Ordering: LD, LD, ST
Connectivity: Port1, Port2, Port3
Group Allocator

- The group allocator allocates entries and ensures allocation respects the sequential program order

Key difference from processor LSQ

Allocate: 2 LDs + 1 ST
Ordering: LD, LD, ST
Connectivity: Port1, Port2, Port3
Outline

- Introduction
- Program order in dynamic circuits
- LSQ implementation
- Evaluation
- Conclusion
Results: Throughput

- Applications with irregular memory access patterns
- LSQ designs dynamically resolve dependencies and produce high-throughput pipelines

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average loop initiation interval (II)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vivado HLS</td>
</tr>
<tr>
<td>Histogram</td>
<td>8</td>
</tr>
<tr>
<td>Matching</td>
<td>11.8</td>
</tr>
<tr>
<td>Matrix Power</td>
<td>14</td>
</tr>
</tbody>
</table>
Results: LSQ Depth

- Resource and clock degradation with queue depth
- In line with previous efforts to implement LSQs on FPGAs\(^1\)

<table>
<thead>
<tr>
<th>Groups</th>
<th>Depth</th>
<th>Ports</th>
<th>CP (ns)</th>
<th>Resources (slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3.5</td>
<td>181</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3.9</td>
<td>353</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>2</td>
<td>5.7</td>
<td>1,144</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>2</td>
<td>7.2</td>
<td>2,998</td>
</tr>
</tbody>
</table>

Results: Benchmarks

- Dynamically scheduled designs are Pareto dominant over the statically scheduled designs

Up to 3x improvement in execution time
(up to 2x with moderate resource cost)
Results: Benchmarks

- Dynamically scheduled designs are Pareto dominant over the statically scheduled designs

General purpose applications require dynamic scheduling \(^1\,^2\)

---

\(^1\) Liu et al. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. FCCM ’15.

\(^2\) Dai et al. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. FPGA ‘17.
Results: Benchmarks

- Dynamically scheduled designs are Pareto dominant over the statically scheduled designs

Our LSQ enables dynamically scheduled designs to compete with the capabilities of OoO processors
Thank you!
The approach can effectively implement designs with larger number of groups.

Applicable to wide range of applications.

<table>
<thead>
<tr>
<th>Groups</th>
<th>Depth</th>
<th>Ports</th>
<th>CP (ns)</th>
<th>Resources (slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>5.7</td>
<td>1,190</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8</td>
<td>5.9</td>
<td>1,326</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>5.7</td>
<td>1,511</td>
</tr>
</tbody>
</table>
Results: Ports

- No influence on cycle time
- Overhead of a single port is minor compared to the overall resources of the LSQ

<table>
<thead>
<tr>
<th>Groups</th>
<th>Depth</th>
<th>Ports</th>
<th>CP (ns)</th>
<th>Resources (slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>2</td>
<td>5.7</td>
<td>1,144</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>4</td>
<td>5.7</td>
<td>1,228</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>6</td>
<td>5.7</td>
<td>1,297</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>5.7</td>
<td>1,190</td>
</tr>
</tbody>
</table>