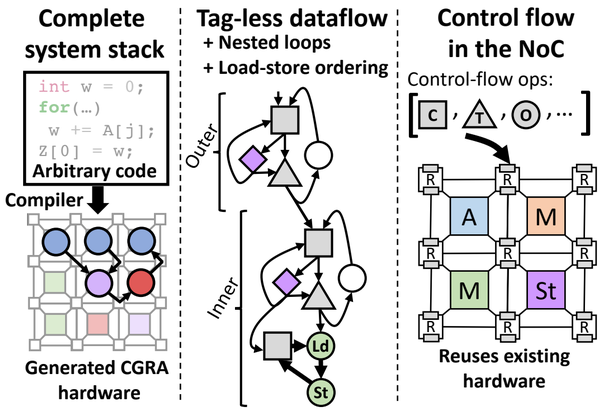
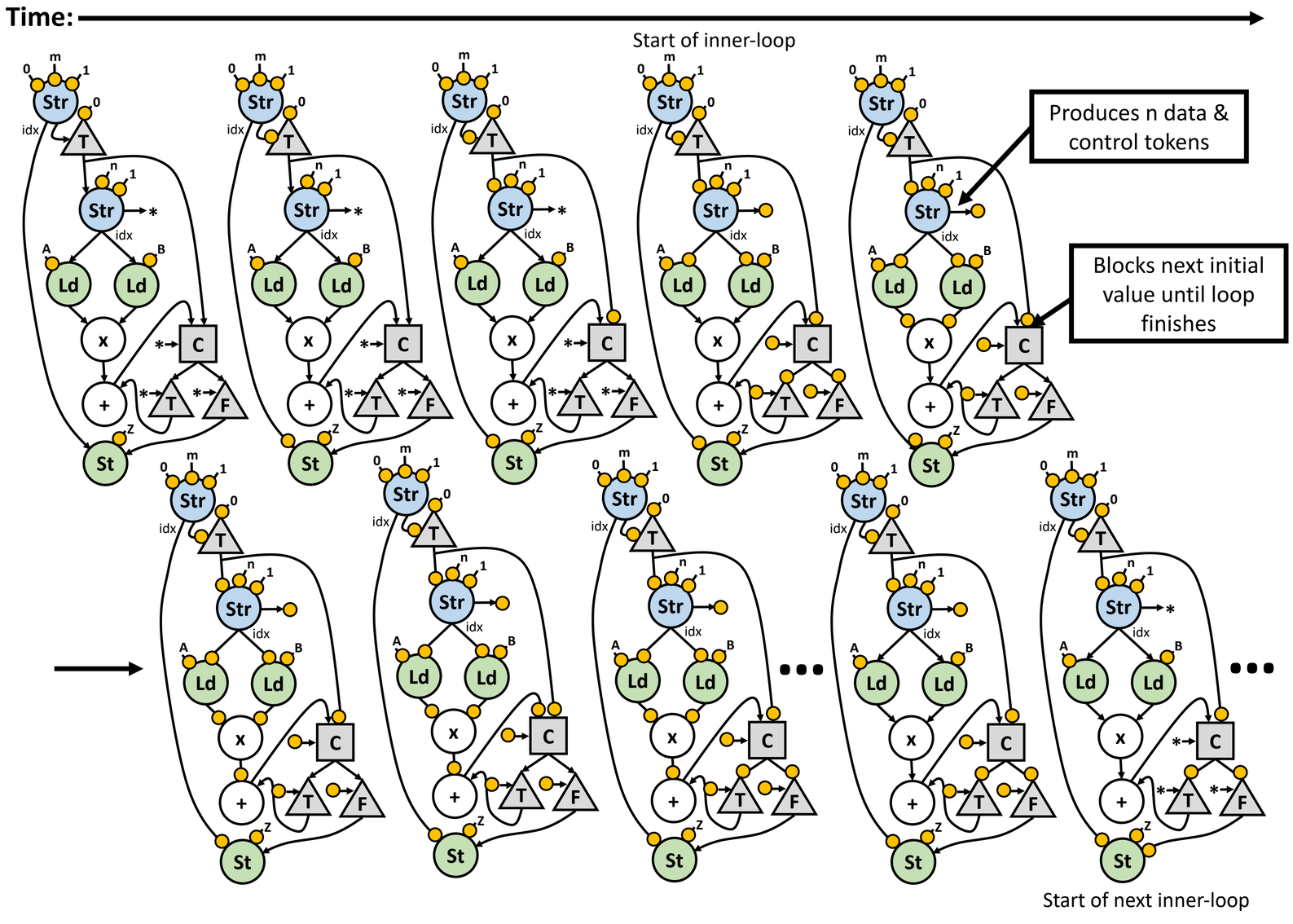
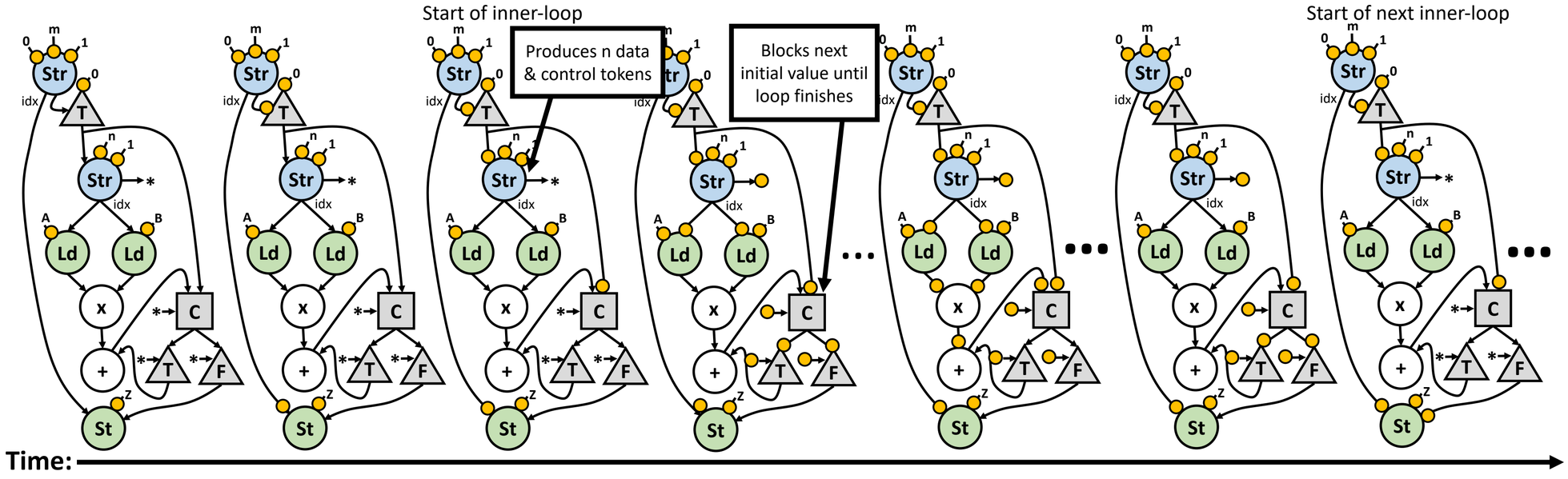
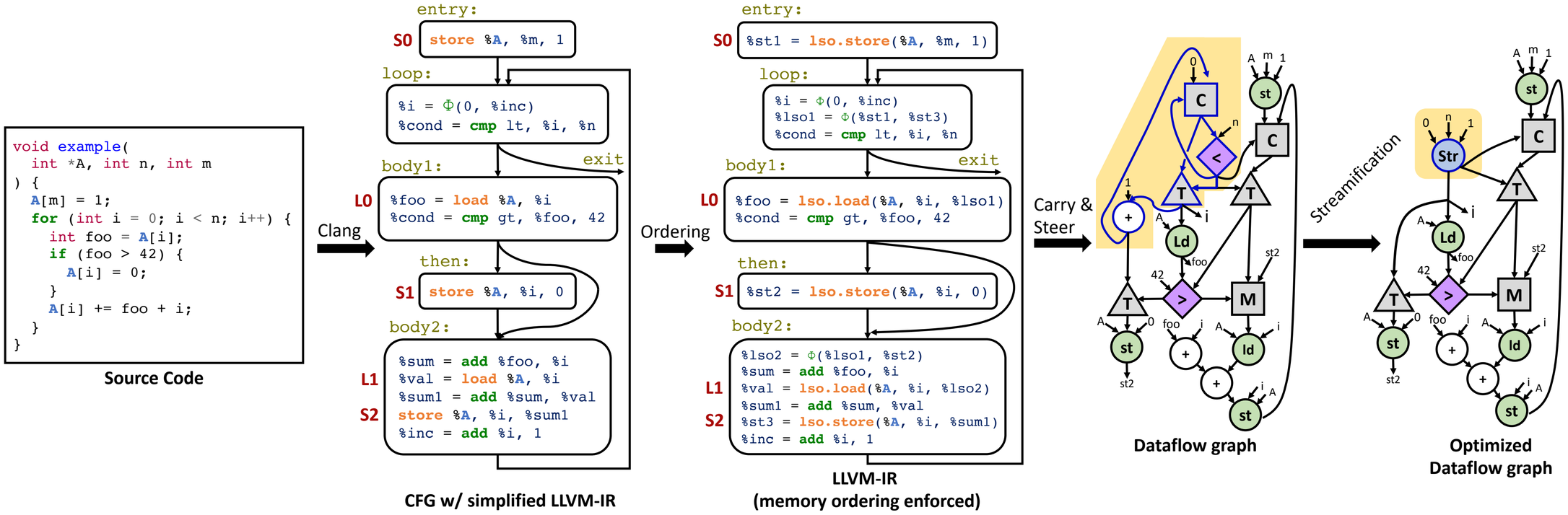
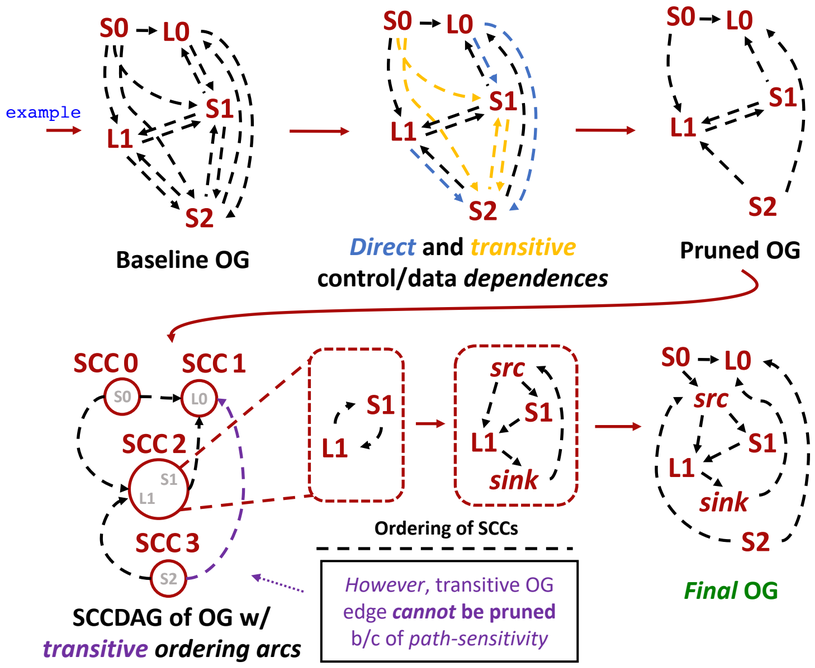
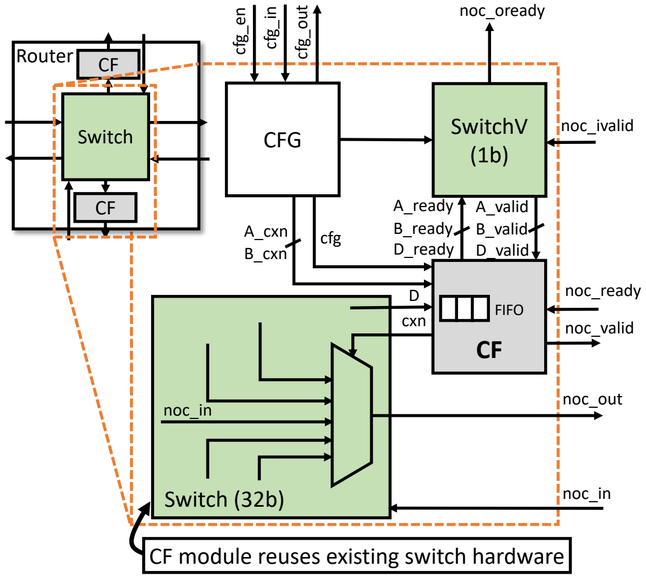

Complete system stack
int w = 0;
for(…)
w += A[j];
Z[0] = w;
Arbitrary code
Compiler
Generated CGRA hardware
Tag-less dataflow
+ Nested loops
+ Load-store ordering
Outer
Inner
Ld
St
Control flow in the NoC
Control-flow ops:
C
T
O
A
M
M
St
R
Reuses existing hardware

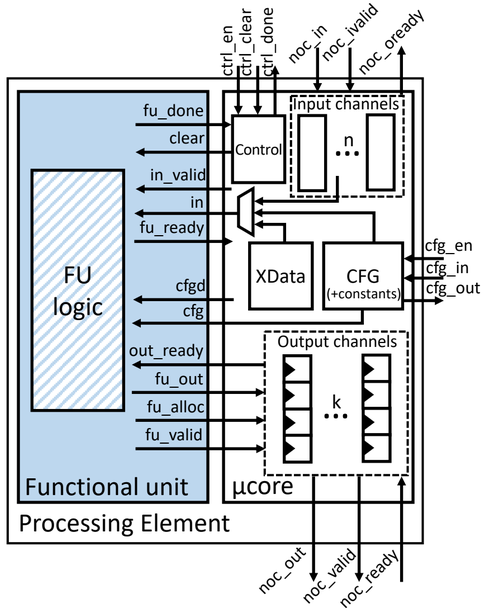

Time:
Start of inner-loop
Produces n data & control tokens
Blocks next initial value until loop finishes
Start of next inner-loop

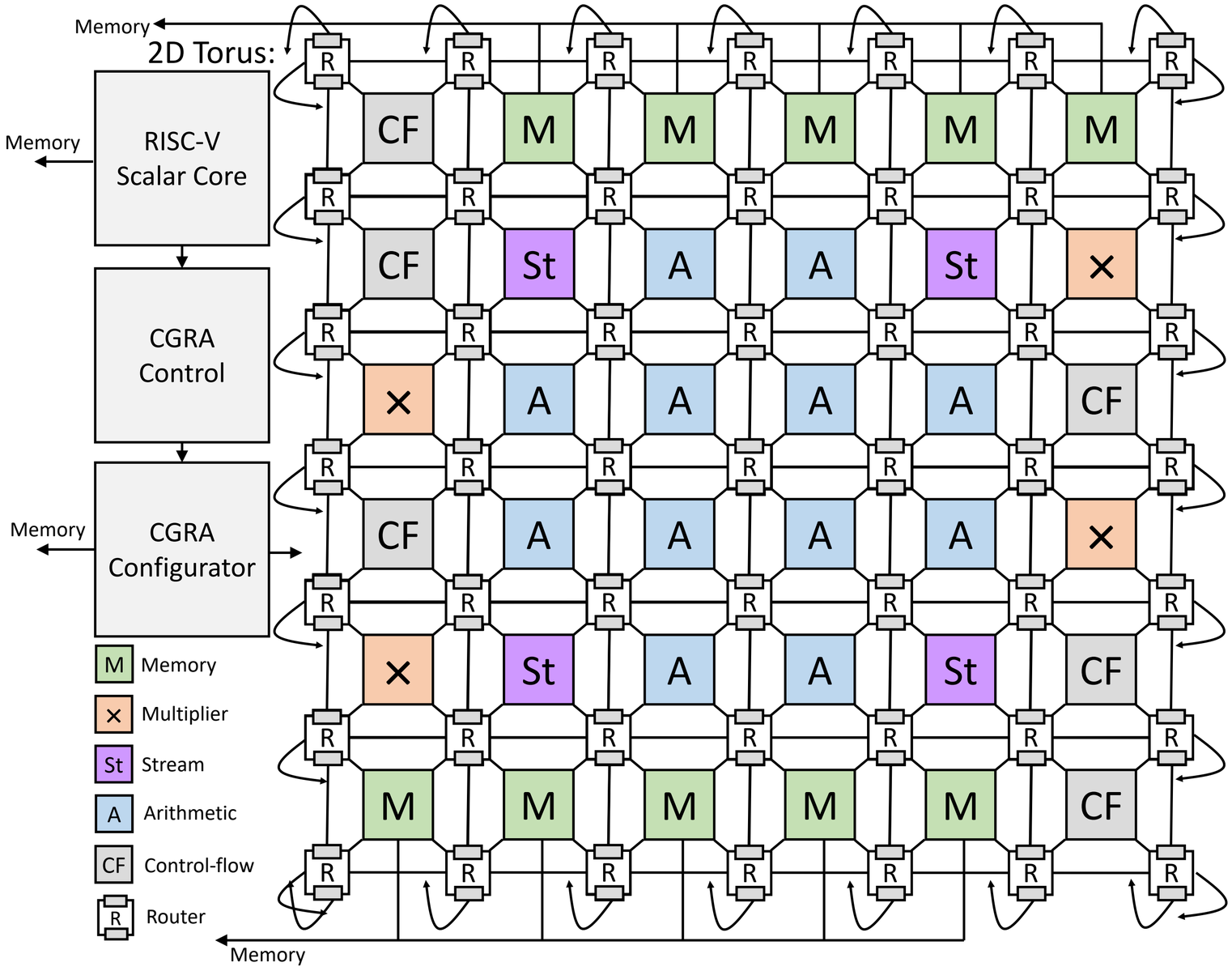

Start of inner-loop
Produces n data & control tokens
Blocks next initial value until loop finishes
Start of next inner-loop
Str
Ld
St
C
T
F
idx
Time:

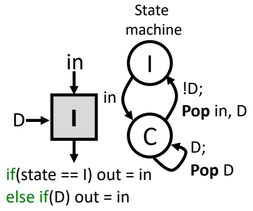

void example(
int *A, int n, int m
) {
A[m] = 1;
for (int i = 0; i < n; i++) {
int foo = A[i];
if (foo > 42) {
A[i] = 0;
}
A[i] += foo + i;
}
}
Source Code
Clang
entry:
S0 store %A, %m, 1
loop:
%i = Φ(0, %inc)
%cond = cmp lt, %i, %n
exit
body1:
L0 %foo = load %A, %i
%cond = cmp gt, %foo, 42
then:
S1 store %A, %i, 0
body2:
L1 %sum = add %foo, %i
%val = load %A, %i
%sum1 = add %sum, %val
S2 store %A, %i, %sum1
%inc = add %i, 1
CFG w/ simplified LLVM-IR
Ordering
entry:
S0 %st1 = lso.store(%A, %m, 1)
loop:
%i = Φ(0, %inc)
%lso1 = Φ(%st1, %st3)
%cond = cmp lt, %i, %n
exit
body1:
L0 %foo = lso.load(%A, %i, %lso1)
%cond = cmp gt, %foo, 42
then:
S1 %st2 = lso.store(%A, %i, 0)
body2:
L1 %lso2 = Φ(%lso1, %st2)
%sum = add %foo, %i
%val = lso.load(%A, %i, %lso2)
S2 %sum1 = add %sum, %val
%st3 = lso.store(%A, %i, %sum1)
%inc = add %i, 1
LLVM-IR
(memory ordering enforced)
Carry &
Steer
Dataflow graph
Streamification
Optimized
Dataflow graph

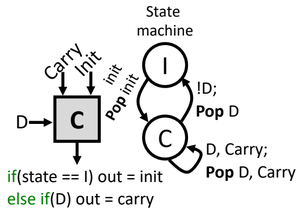

example
S0
L0
S1
L1
S2
Baseline OG
Direct and transitive
control/data dependences
Pruned OG
SCC 0
SCC 1
SCC 2
SCC 3
SCCDAG of OG w/
transitive ordering arcs
Ordering of SCCs
src
sink
However, transitive OG
edge cannot be pruned
b/c of path-sensitivity
Final OG

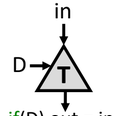

Router
CF
Switch
CF
cfg_en
cfg_in
cfg_out
CFG
noc_oready
SwitchV
(1b)
noc_ivalid
A_cxn
B_cxn
cfg
A_ready
B_ready
D_ready
A_valid
B_valid
D_valid
D
cxn
FIFO
CF
noc_ready
noc_valid
noc_out
noc_in
Switch (32b)
noc_in
CF module reuses existing switch hardware


ctrl_en
ctrl_clear
ctrl_done
noc_in
noc_ivalid
noc_oready
fu_done
clear
in_valid
in
fu_ready
cfgd
cfg
out_ready
fu_out
fu_alloc
fu_valid
FU
logic
Functional unit
Control
Input channels
n
XData
CFG
(+constants)
cfg_en
cfg_in
cfg_out
Output channels
k
µcore
Processing Element
noc_out
noc_valid
noc_ready

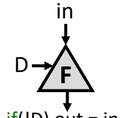

Memory
2D Torus:
Memory
RISC-V
Scalar Core
CGRA
Control
Memory
CGRA
Configurator
M Memory
× Multiplier
St Stream
A Arithmetic
CF Control-flow
R Router
Memory


State
machine
in
D
I
in
I
C
!D;
Pop in, D
D;
Pop D
if(state == I) out = in
else if(D) out = in

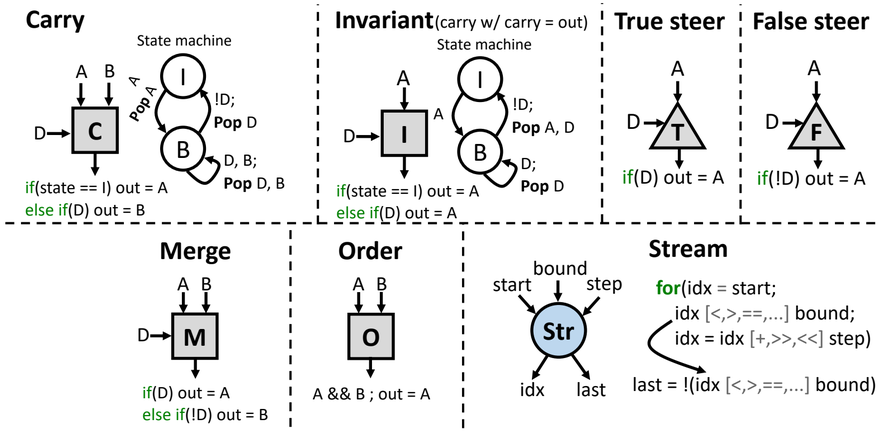

State
machine
Carry
Init
init
Pop init
D
C
I
C
!D;
Pop D
D, Carry;
Pop D, Carry
if(state == I) out = init
else if(D) out = carry

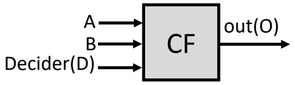


if(D) out = in

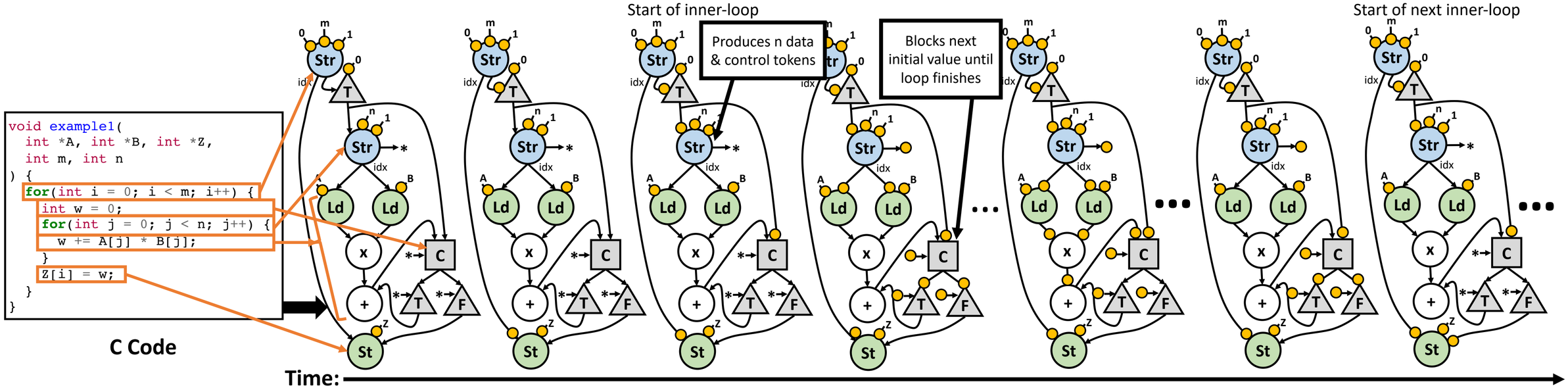


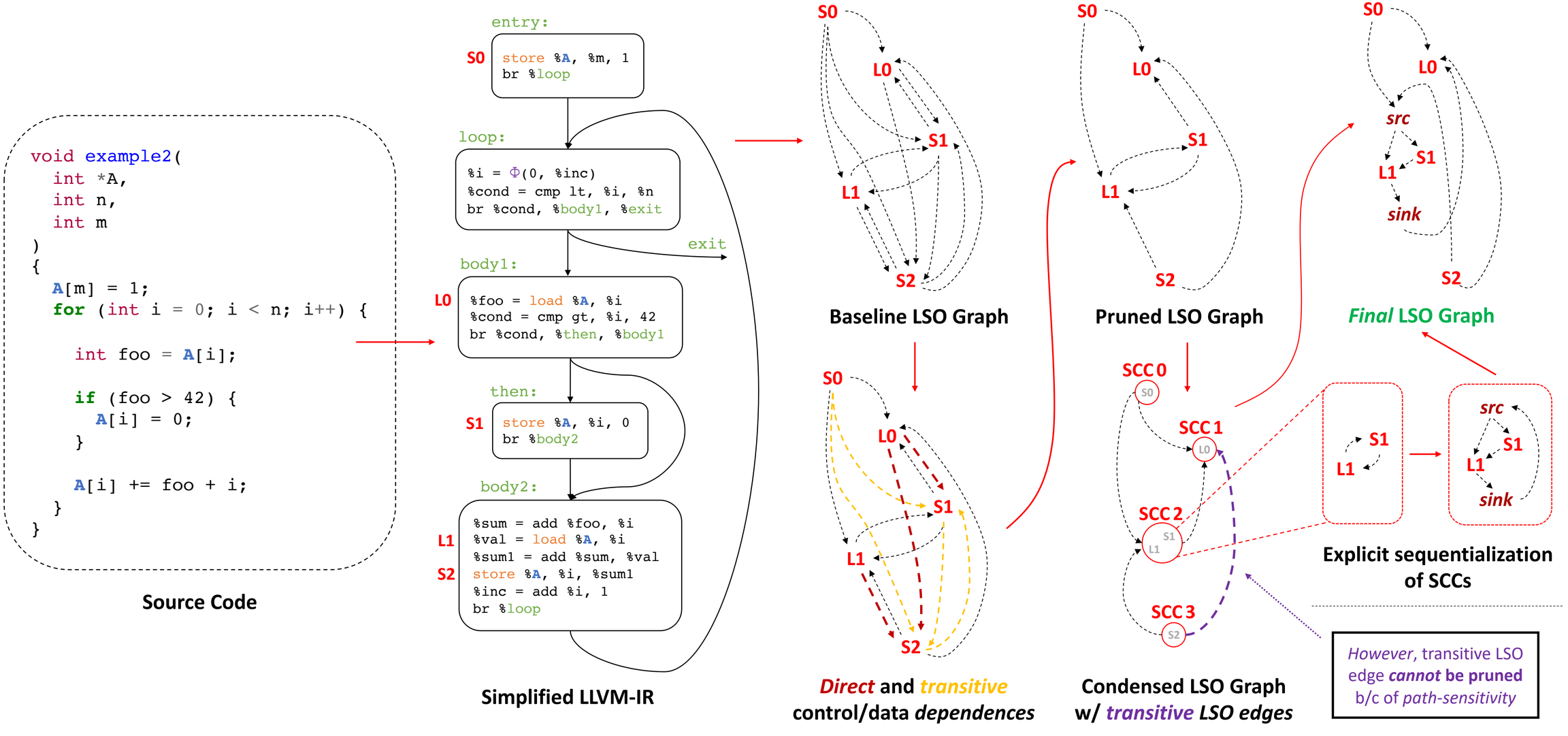
if(!D) out = in

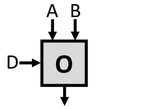

Carry
State machine
A
B
D
C
I
B
Pop A
!D;
Pop D
D, B;
Pop D, B
if(state == I) out = A
else if(D) out = B
Invariant(carry w/ carry = out)
State machine
A
D
I
I
B
!D;
Pop A, D
D;
Pop D
if(state == I) out = A
else if(D) out = A
True steer
A
D
T
if(D) out = A
False steer
A
D
F
if(!D) out = A
Merge
A
B
D
M
if(D) out = A
else if(!D) out = B
Order
A
B
O
A && B ; out = A
bound
start
step
Str
idx
last
Stream
for(idx = start;
idx [<,>,==,...] bound;
idx = idx [+,>>,<<] step)
last = !(idx [<,>,==,...] bound)


A
B
Decider(D)
CF
out(O)

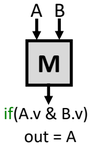
```
void example1(
  int *A, int *B, int *Z,
  int m, int n
) {
  for(int i = 0; i < m; i++) {
    int w = 0;
    for(int j = 0; j < n; j++) {
      w += A[j] * B[j];
    }
    Z[i] = w;
  }
}
```

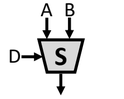

void example1(
int *A, int *B, int *Z,
int m, int n
) {
for(int i = 0; i < m; i++) {
int w = 0;
for(int j = 0; j < n; j++) {
w += A[j] * B[j];
}
Z[i] = w;
}
}
C Code
Start of inner-loop
Produces n data & control tokens
Blocks next initial value until loop finishes
Start of next inner-loop
Str
Ld
St
Time:


void example2(
int *A,
int n,
int m
)
{
A[m] = 1;
for (int i = 0; i < n; i++) {
int foo = A[i];
if (foo > 42) {
A[i] = 0;
}
A[i] += foo + i;
}
}
Source Code
entry:
S0
store %A, %m, 1
br %loop
loop:
%i = Φ(0, %inc)
%cond = cmp lt, %i, %n
br %cond, %body1, %exit
exit
body1:
L0
%foo = load %A, %i
%cond = cmp gt, %i, 42
br %cond, %then, %body1
then:
S1
store %A, %i, 0
br %body2
body2:
L1
%sum = add %foo, %i
%val = load %A, %i
%sum1 = add %sum, %val
S2
store %A, %i, %sum1
%inc = add %i, 1
br %loop
Simplified LLVM-IR
S0
L0
S1
L1
S2
Baseline LSO Graph
Direct and transitive
control/data dependences
Pruned LSO Graph
SCC 0
SCC 1
SCC 2
SCC 3
Condensed LSO Graph
w/ transitive LSO edges
Final LSO Graph
src
sink
Explicit sequentialization
of SCCs
However, transitive LSO
edge cannot be pruned
b/c of path-sensitivity

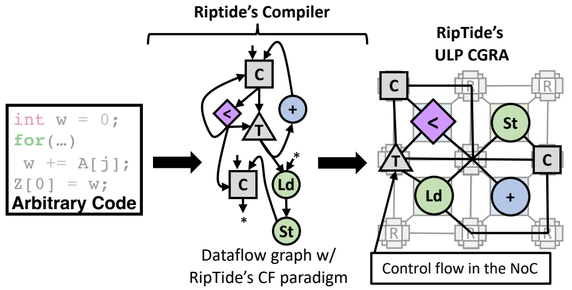


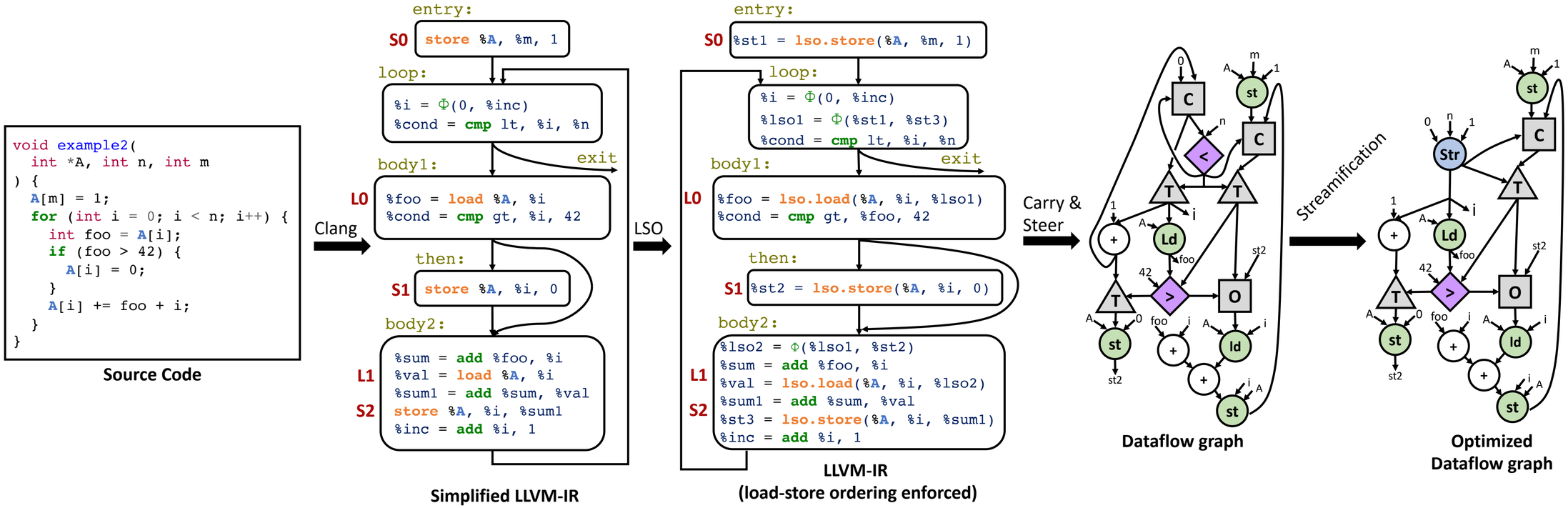
if(D) out = A
else if(!D) out = B

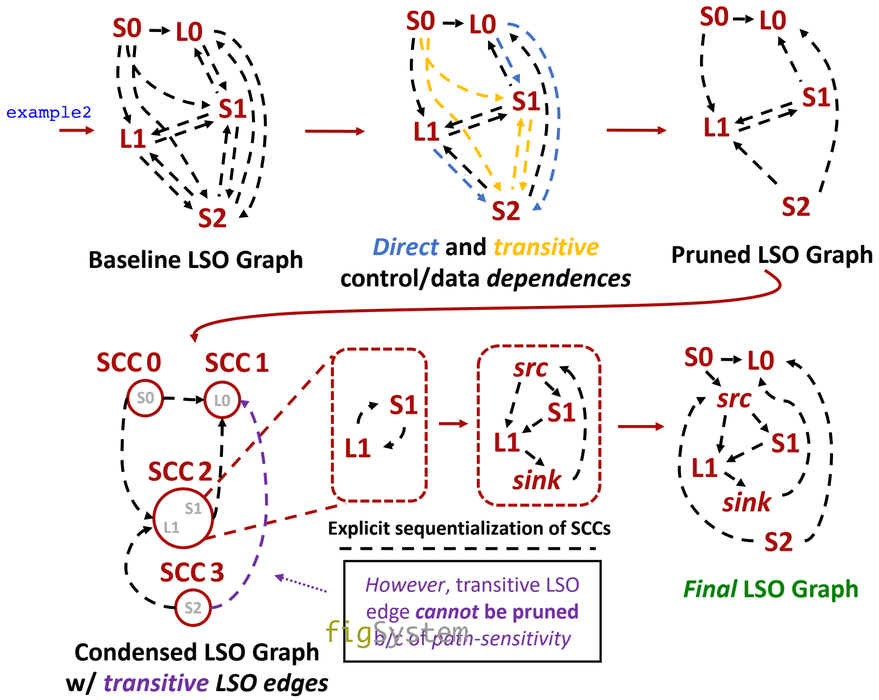

A
B
M
if(A.v & B.v)
out = A



out = (D) ? A : D


Riptide's Compiler
RipTide's
ULP CGRA
int w = 0;
for(…)
w += A[j];
Z[0] = w;
Arbitrary Code
C
<
T
+
C
Ld
St
*
*
Dataflow graph w/
RipTide's CF paradigm
C
<
St
T
C
Ld
+
R
R
Control flow in the NoC


void example2(
int *A, int n, int m
) {
A[m] = 1;
for (int i = 0; i < n; i++) {
int foo = A[i];
if (foo > 42) {
A[i] = 0;
}
A[i] += foo + i;
}
}
Source Code
Clang
entry:
S0 store %A, %m, 1
loop:
%i = Φ(0, %inc)
%cond = cmp lt, %i, %n
exit
body1:
L0 %foo = load %A, %i
%cond = cmp gt, %i, 42
then:
S1 store %A, %i, 0
body2:
L1 %sum = add %foo, %i
%val = load %A, %i
%sum1 = add %sum, %val
S2 store %A, %i, %sum1
%inc = add %i, 1
Simplified LLVM-IR
LSO
entry:
S0 %st1 = lso.store(%A, %m, 1)
loop:
%i = Φ(0, %inc)
%lso1 = Φ(%st1, %st3)
%cond = cmp lt, %i, %n
exit
body1:
L0 %foo = lso.load(%A, %i, %lso1)
%cond = cmp gt, %foo, 42
then:
S1 %st2 = lso.store(%A, %i, 0)
body2:
L1 %lso2 = Φ(%lso1, %st2)
%sum = add %foo, %i
%val = lso.load(%A, %i, %lso2)
S2 %sum1 = add %sum, %val
%st3 = lso.store(%A, %i, %sum1)
%inc = add %i, 1
LLVM-IR
(load-store ordering enforced)
Carry &
Steer
Dataflow graph
Streamification
Optimized
Dataflow graph


example2
S0
L0
S1
L1
S2
Baseline LSO Graph
Direct and transitive
control/data dependences
Pruned LSO Graph
SCC 0
SCC 1
SCC 2
SCC 3
Condensed LSO Graph
w/ transitive LSO edges
src
sink
Explicit sequentialization of SCCs
However, transitive LSO
edge cannot be pruned
b/c of path-sensitivity
Final LSO Graph
figSystem

Loop annotations (in other works)

```
// REVEL-like pragmas
#pragma config
#pragma stream
#pragma dataflow in(...)
for (...)
```

```
// Annotations for 4D-CGRA, etc.
#pragma accelerate
for (...)
...
```

RipTide Code (Native C)

```
#define u16 uint16_t
#define u32 uint32_t
#define res restrict // RipTide annotation

void simple_bfs(
  u16 * res rows, u16 * res cols, // Graph in CSR
  u16 * res queue, u16 * res visited, // Helpers
  u16 * res walk  // Output
) {
  while (!stack_empty()) {
    // Record next vertex in @walk
    u16 next = pop();
    add_to_walk(next);

    // Add neighbors to @queue
    for (u32 i = rows[next];
         i < rows[next + 1]; i++) {
      u32 dst = cols[i];
      if (!visited[dst]) {
        push(dst);
        visited[dst] = 1;
      }
    }
  }
}
```

SNAFU assembly excerpt

```
// Config to offload loop
u16 tmp;
u32 start = rows[next];
u32 stop = rows[next + 1];
if ((stop – start) <= 0) continue;
vcfg(((stop – start), _kernel);
vtfr(cols + start, BFS_SNAFU_VTFR0);
vtfr(visited, BFS_SNAFU_VTFR1);
...
vfence();

// SNAFU assembly for loop
vlh(v1, cols);
vlxh(v2, visited, v1);
vseqi(v0, v2, 0);
vsxh(visited, v0.m, v1);
vpresum(v4, v0);
...
```

a.

```
1 void foo (...) {
2   ...
3   #pragma target
4   for (i = 0..n)
5     ... = a[i] ...
6   ...
7 }
```

b.

```
1 void foo (...) {
2   ...
3   #pragma target
4   for (i = 0..n)
5     for (j = 0..n)
6       ... = a[i][j] ...
7   ...
8 }
```

e.

```
vcfg(...);
… // Config
vlh(v1, b);
vlxh(v2, a, v1);
veq(v0, v2, 1);
… // Ops for push(i)
```

c.

```
1  void foo (...) {
2    ...
3    #pragma config ...
4    for (i = 0..n)
5      ...
6      #pragma stream
7      #pragma dataflow in(...)
8      for (j = i..n)
9        ... = a[i][j] ...
10   ...
11 }
```

d.

```
1  void foo (int * restrict a,
2            int * restrict b, ...) {
2    ...
4    while (!queue.empty()) {
5      n = queue.pop()
6      for (i = a[n]..a[n + 1]))
7        if (b[a[i]]) queue.push(i)
8        ...
9    }
10   ...
11 }
```

```
void foo (...) {
  #pragma target
  for (i = 0..n)
    ... = a[i]
}
```

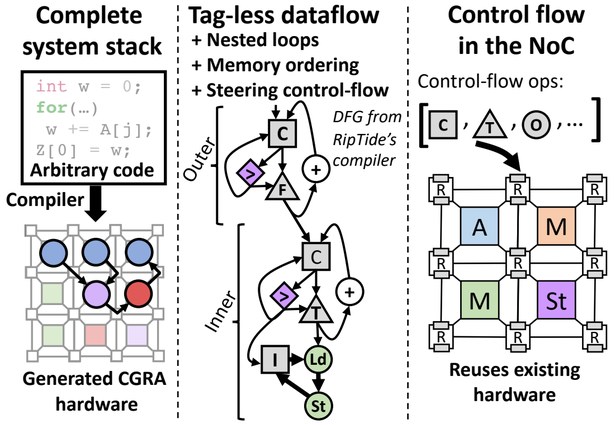
```
void foo (...) {
  ...
  #pragma target
  for (i = 0..n)
    for (j = 0..n)
      ... = a[i][j]
  ...
}
```

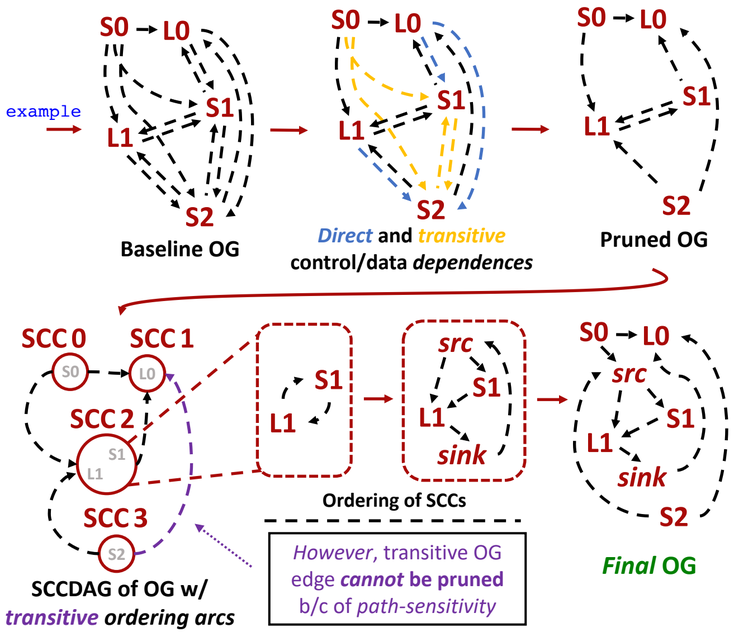
```
void foo (...) {
   #pragma config ...
   for (i = 0..n) ...
     #pragma stream
     #pragma dataflow
     for (j = i..n)
       ... = a[i][j]
}
```

```
#riptide void foo
  (int * restrict a, b) {
   while (!q.empty()) {
     n = q.pop()
     for (i in 0..n)
       if (b[a[i]]) ...
   }
}
```

```
void foo (...) {
  for (i = 0..n) ...
     vlh v1, a + i
     vlh v2, b
     vadd v3, v1, v2
     vsh b + i, v3
    ...
}
```

```
void foo (int * a,
          int * b) {
   while (!q.empty()) {
     n = q.pop()
     for (i in 0..n)
       if (b[a[i]]) ...
   }
}
```


Complete system stack
int w = 0;
for(…)
w += A[j];
Z[0] = w;
Arbitrary code
Compiler
Generated CGRA hardware
Tag-less dataflow
+ Nested loops
+ Memory ordering
+ Steering control-flow
DFG from RipTide's compiler
Outer
Inner
C
F
T
I
Ld
St
Control flow in the NoC
Control-flow ops:
A
M
M
St
R
Reuses existing hardware


example
S0
L0
S1
L1
S2
Baseline OG
Direct and transitive
control/data dependences
Pruned OG
SCC 0
SCC 1
SCC 2
SCC 3
SCCDAG of OG w/
transitive ordering arcs
Ordering of SCCs
src
sink
However, transitive OG
edge cannot be pruned
b/c of path-sensitivity
Final OG

```
void foo (...) {
  ...
  for (i = 0..n) {
    while (a[i] != 0)
      a[i] = ...
  ...
  }
}
```