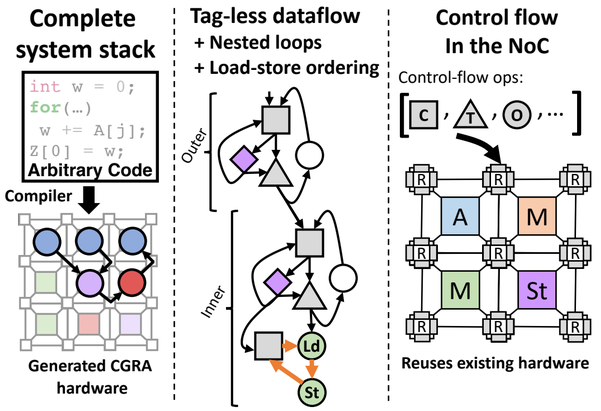
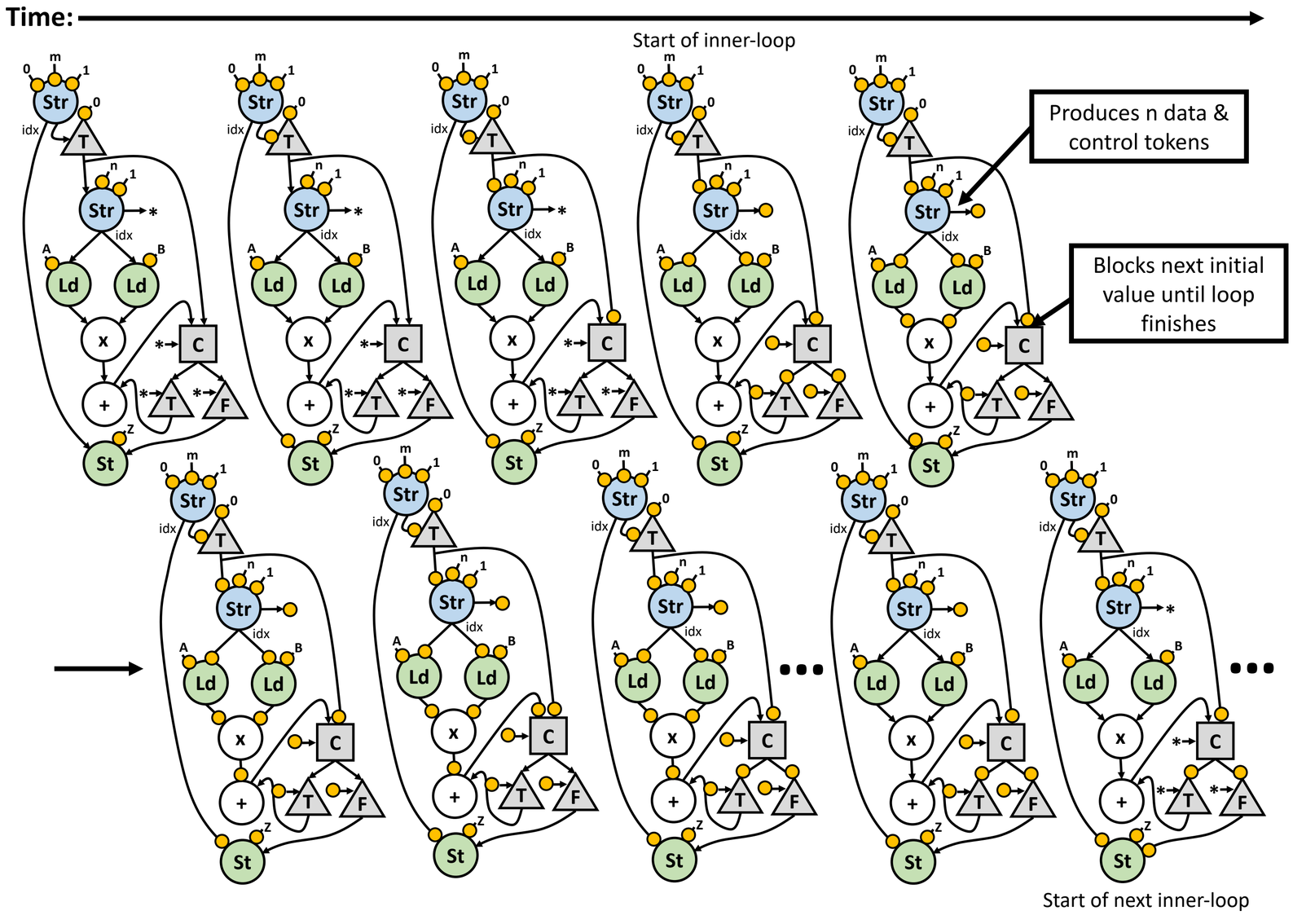
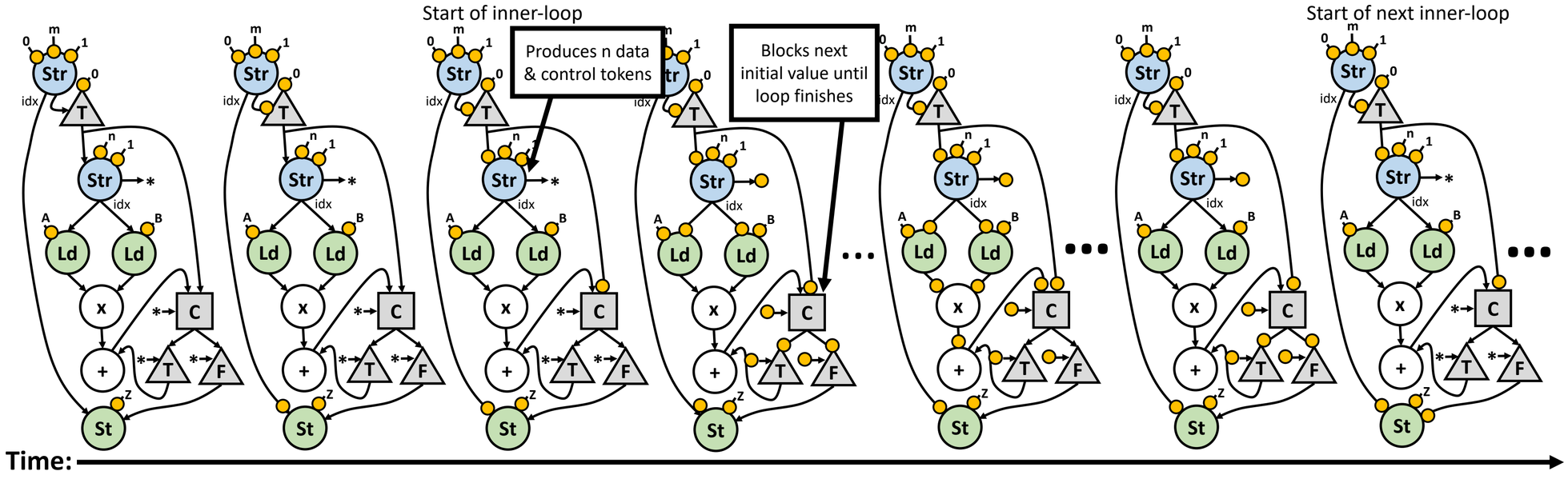
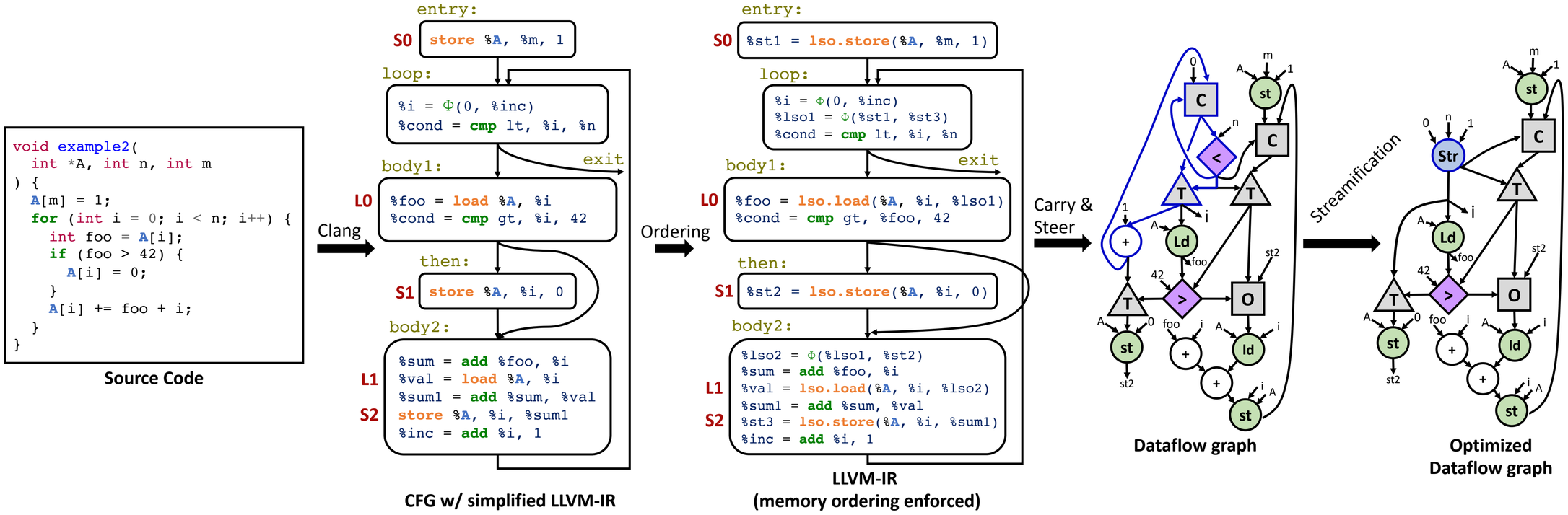
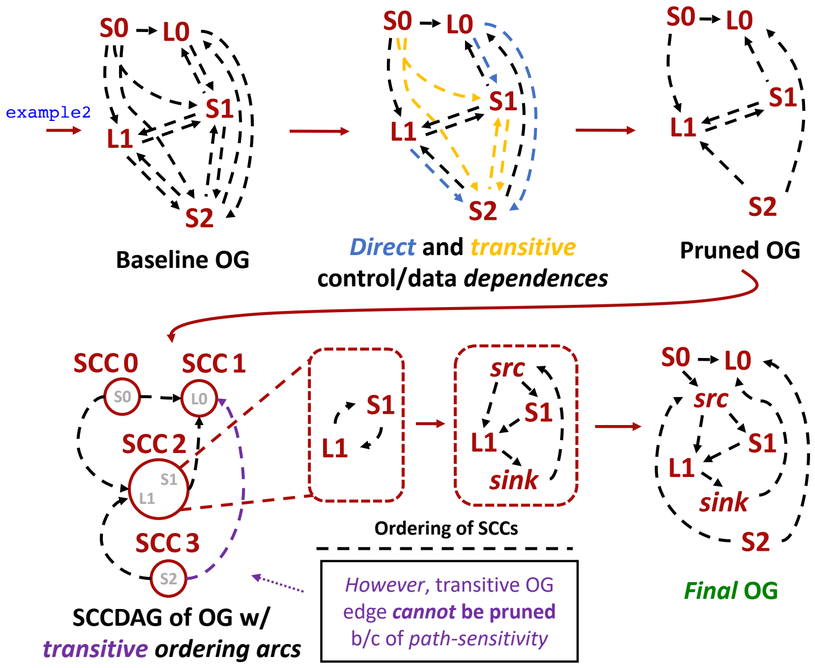
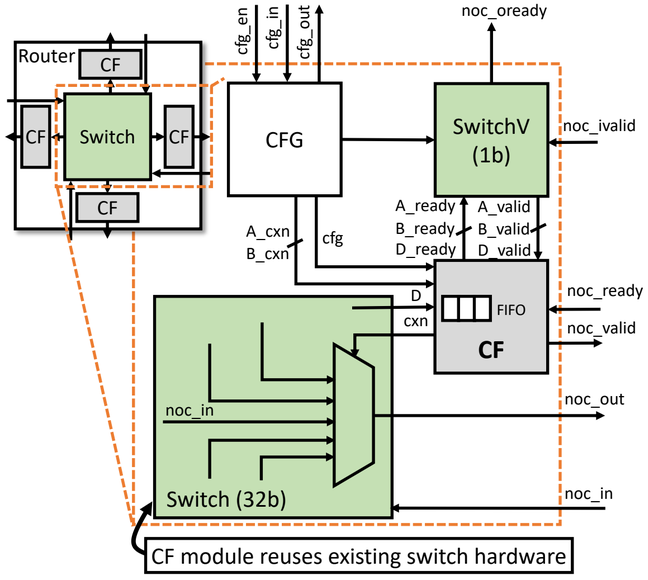

Complete system stack
int w = 0;
for(…)
w += A[j];
Z[0] = w;
Arbitrary Code
Compiler
Generated CGRA hardware
Tag-less dataflow
+ Nested loops
+ Load-store ordering
Outer
Inner
Ld
St
Control flow In the NoC
Control-flow ops:
C
T
O
…
R
A
M
M
St
Reuses existing hardware

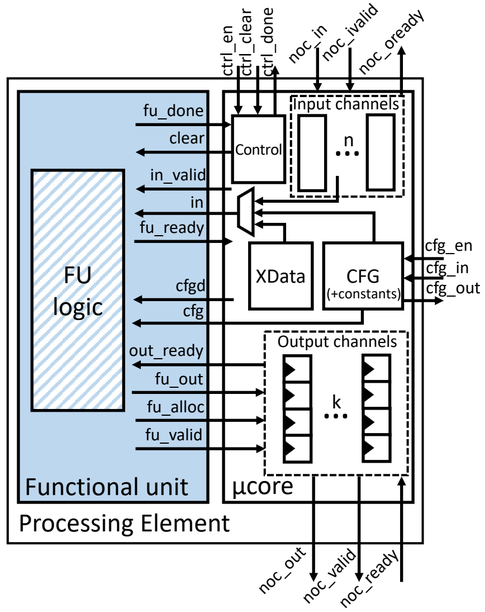

Time:
Start of inner-loop
Produces n data & control tokens
Blocks next initial value until loop finishes
Start of next inner-loop

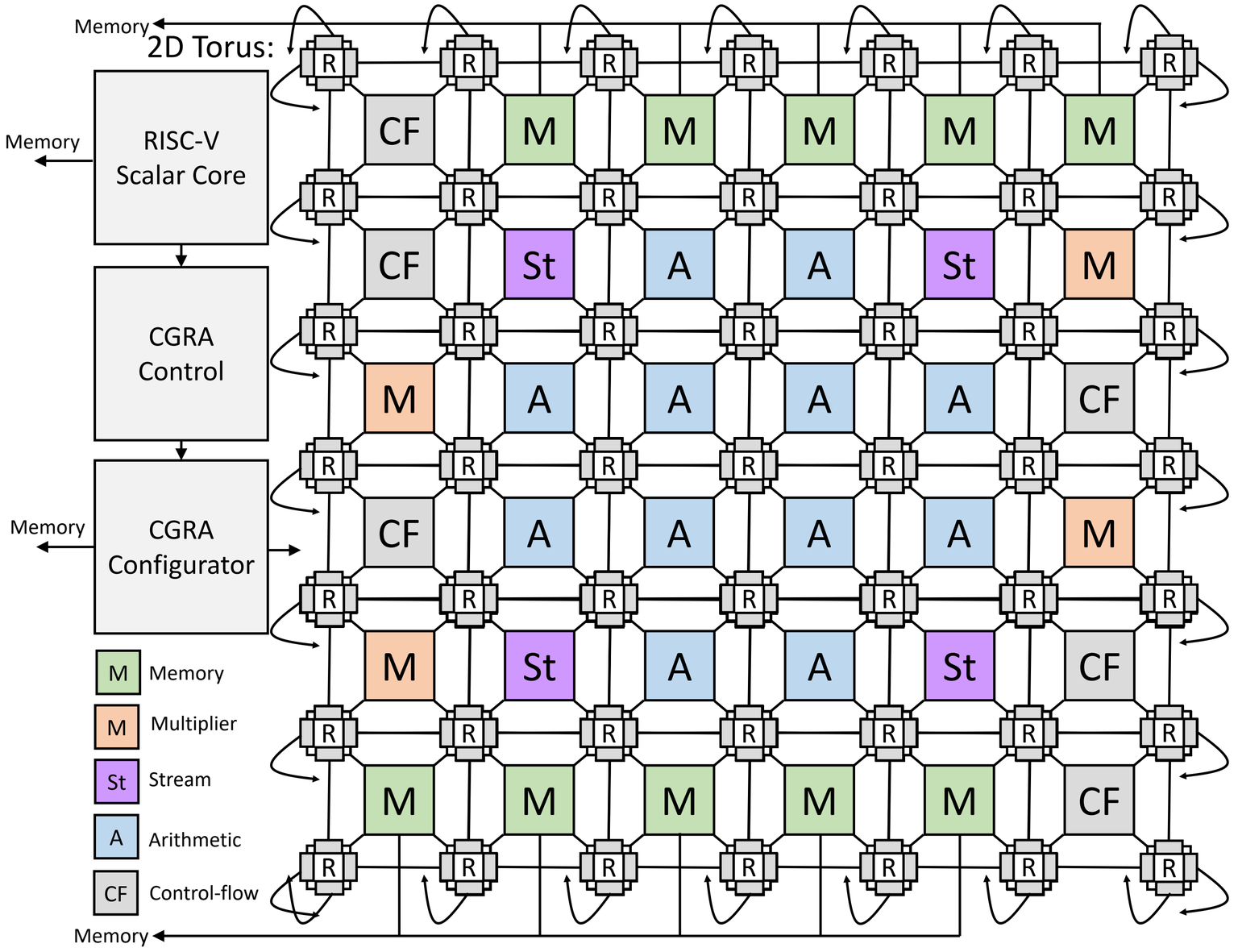

Start of inner-loop
Produces n data & control tokens
Blocks next initial value until loop finishes
Start of next inner-loop
Str
Ld
St
C
T
F
Time:

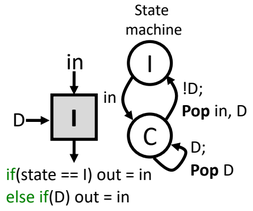

void example2(
int *A, int n, int m
) {
A[m] = 1;
for (int i = 0; i < n; i++) {
int foo = A[i];
if (foo > 42) {
A[i] = 0;
}
A[i] += foo + i;
}
}
Source Code
Clang
entry:
S0 store %A, %m, 1
loop:
%i = Φ(0, %inc)
%cond = cmp lt, %i, %n
exit
body1:
L0 %foo = load %A, %i
%cond = cmp gt, %i, 42
then:
S1 store %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
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

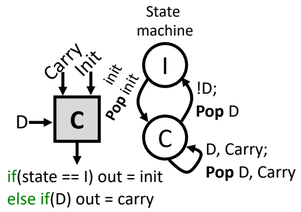

example2
S0
L0
S1
L1
S2
Baseline OG
Direct and transitive
control/data dependences
Pruned OG
SCC 0
SCC 1
SCC 2
SCC 3
SCCDAG of OG w/
transitive ordering arcs
src
sink
Ordering of SCCs
However, transitive OG
edge cannot be pruned
b/c of path-sensitivity
Final OG

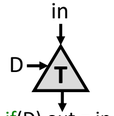

Router
CF
CF
Switch
CF
CF
cfg_en
cfg_in
cfg_out
CFG
noc_oready
SwitchV
(1b)
noc_ivalid
A_ready
B_ready
D_ready
A_valid
B_valid
D_valid
A_cxn
B_cxn
cfg
D
cxn
FIFO
CF
noc_ready
noc_valid
noc_out
noc_in
Switch (32b)
noc_in
CF module reuses existing switch hardware


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
Control
Input channels
n
XData
CFG
(+constants)
cfg_en
cfg_in
cfg_out
Output channels
k
FU
logic
Functional unit
µcore
Processing Element
noc_out
noc_valid
noc_ready

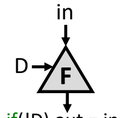

2D Torus:
Memory
Memory
RISC-V
Scalar Core
CGRA
Control
Memory
CGRA
Configurator
R
CF
M
St
A
M Memory
M Multiplier
St Stream
A Arithmetic
CF Control-flow
Memory


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

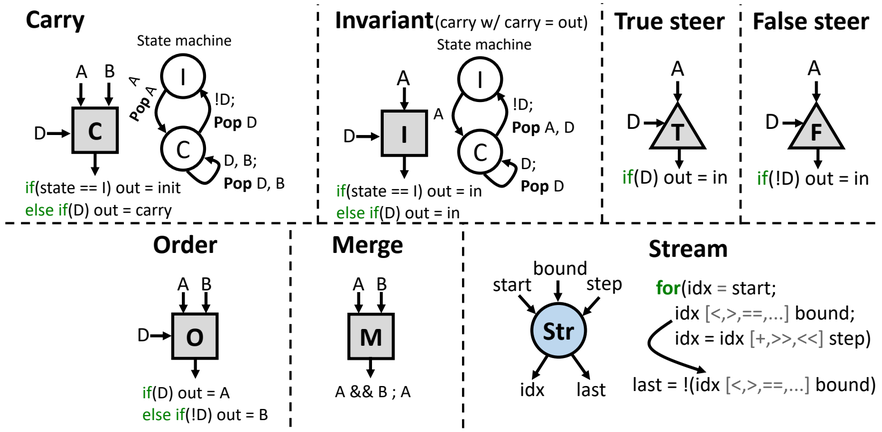

State
machine
Carry
Init
init
Pop init
D
C
I
C
!D;
Pop D
D, Carry;
Pop D, Carry
if(state == I) out = init
else if(D) out = carry

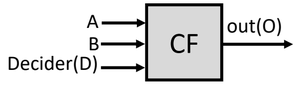


if(D) out = in

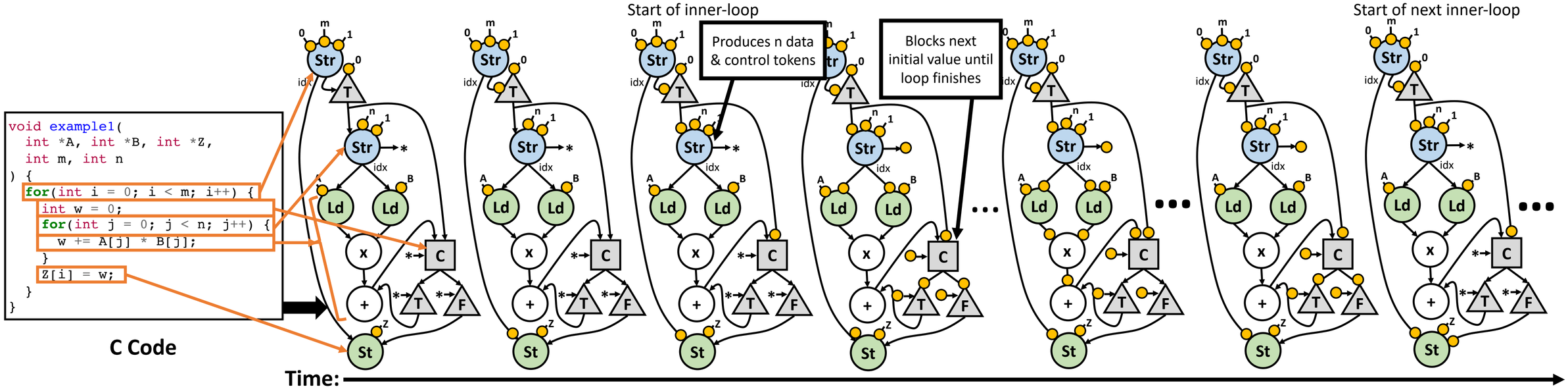


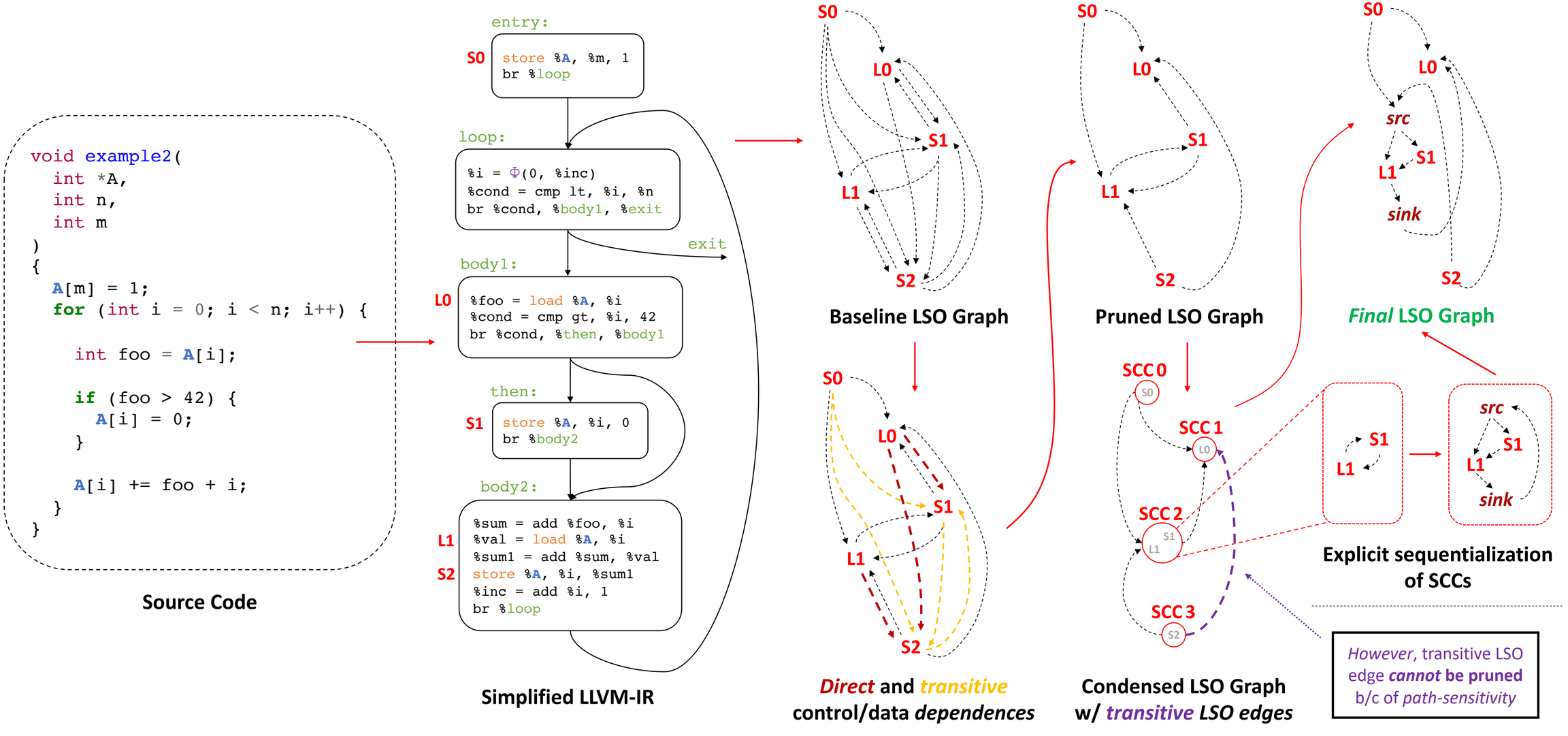
if(!D) out = in

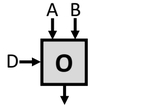

Carry
State machine
A
B
D
C
I
C
A
Pop A
!D;
Pop D
D, B;
Pop D, B
if(state == I) out = init
else if(D) out = carry
Invariant(carry w/ carry = out)
State machine
A
D
I
I
C
A
!D;
Pop A, D
D;
Pop D
if(state == I) out = in
else if(D) out = in
True steer
A
D
T
if(D) out = in
False steer
A
D
F
if(!D) out =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


A
B
Decider(D)
CF
out(O)

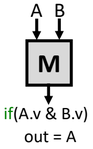
```
void example1(
  int *A, int *B, int *Z,
  int m, int n
) {
  for(int i = 0; i < m; i++) {
    int w = 0;
    for(int j = 0; j < n; j++) {
      w += A[j] * B[j];
    }
    Z[i] = w;
  }
}
```

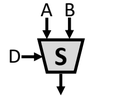

void example1(
int *A, int *B, int *Z,
int m, int n
) {
for(int i = 0; i < m; i++) {
int w = 0;
for(int j = 0; j < n; j++) {
w += A[j] * B[j];
}
Z[i] = w;
}
}
C Code
Start of inner-loop
Produces n data & control tokens
Blocks next initial value until loop finishes
Start of next inner-loop
Str
Ld
St
Time:


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
src
sink
Final LSO Graph
Explicit sequentialization
of SCCs
However, transitive LSO
edge cannot be pruned
b/c of path-sensitivity

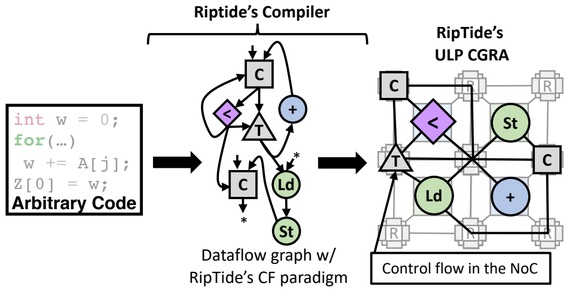


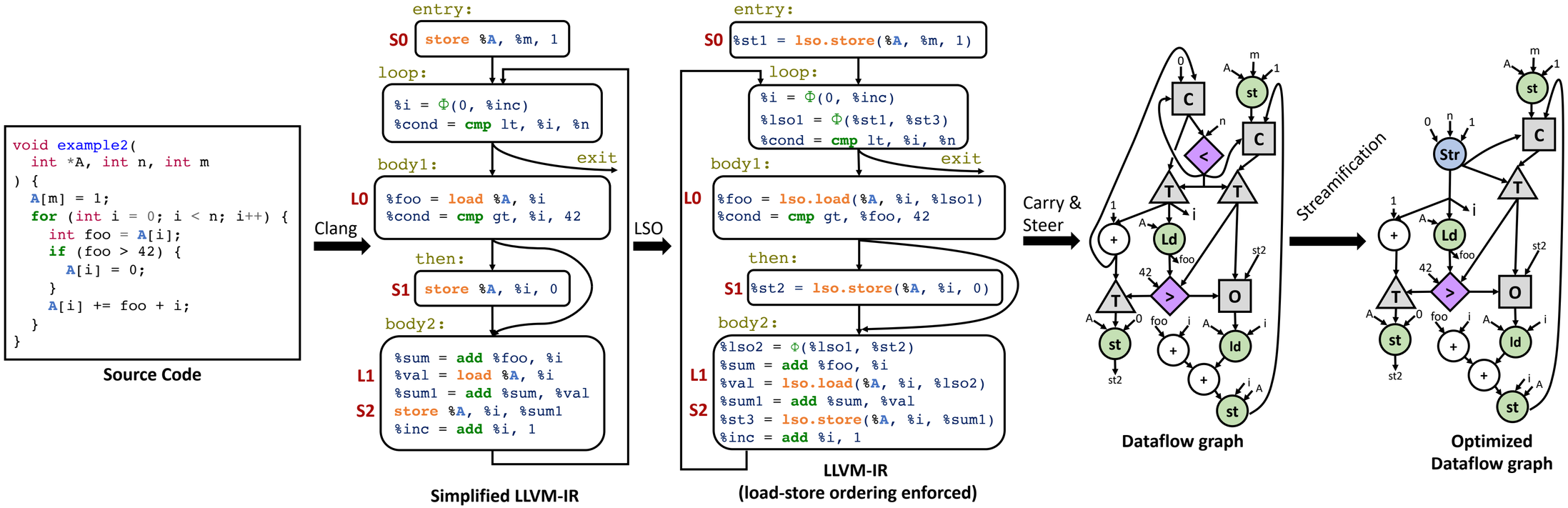
if(D) out = A
else if(!D) out = B

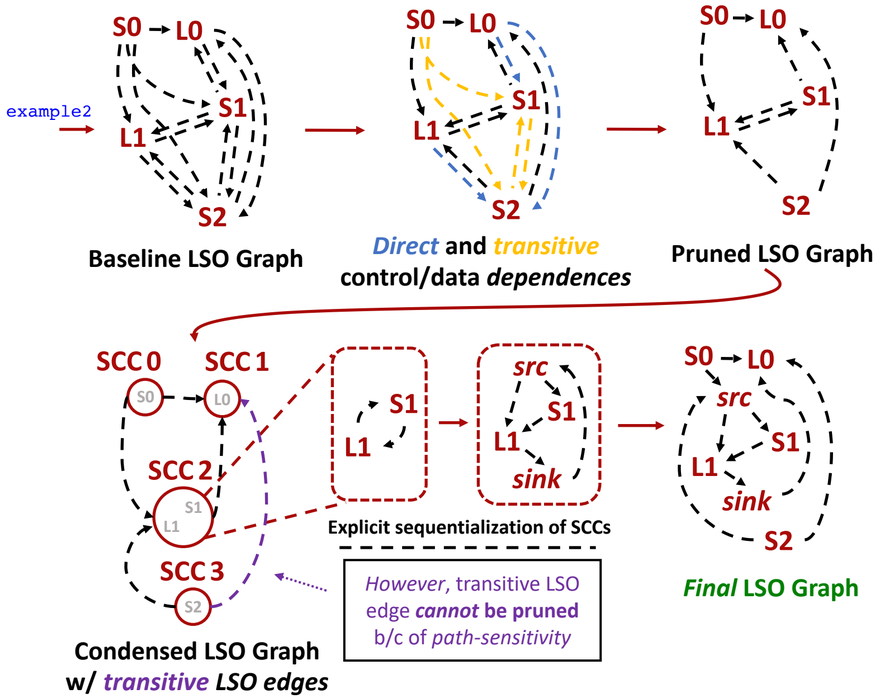

A
B
M
if(A.v & B.v)
out = A



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
*
*
C
St
C
T
Ld
+
R
R
Dataflow graph w/
RipTide's CF paradigm
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
Explicit sequentialization of SCCs
src
sink
However, transitive LSO
edge cannot be pruned
b/c of path-sensitivity
Final LSO Graph