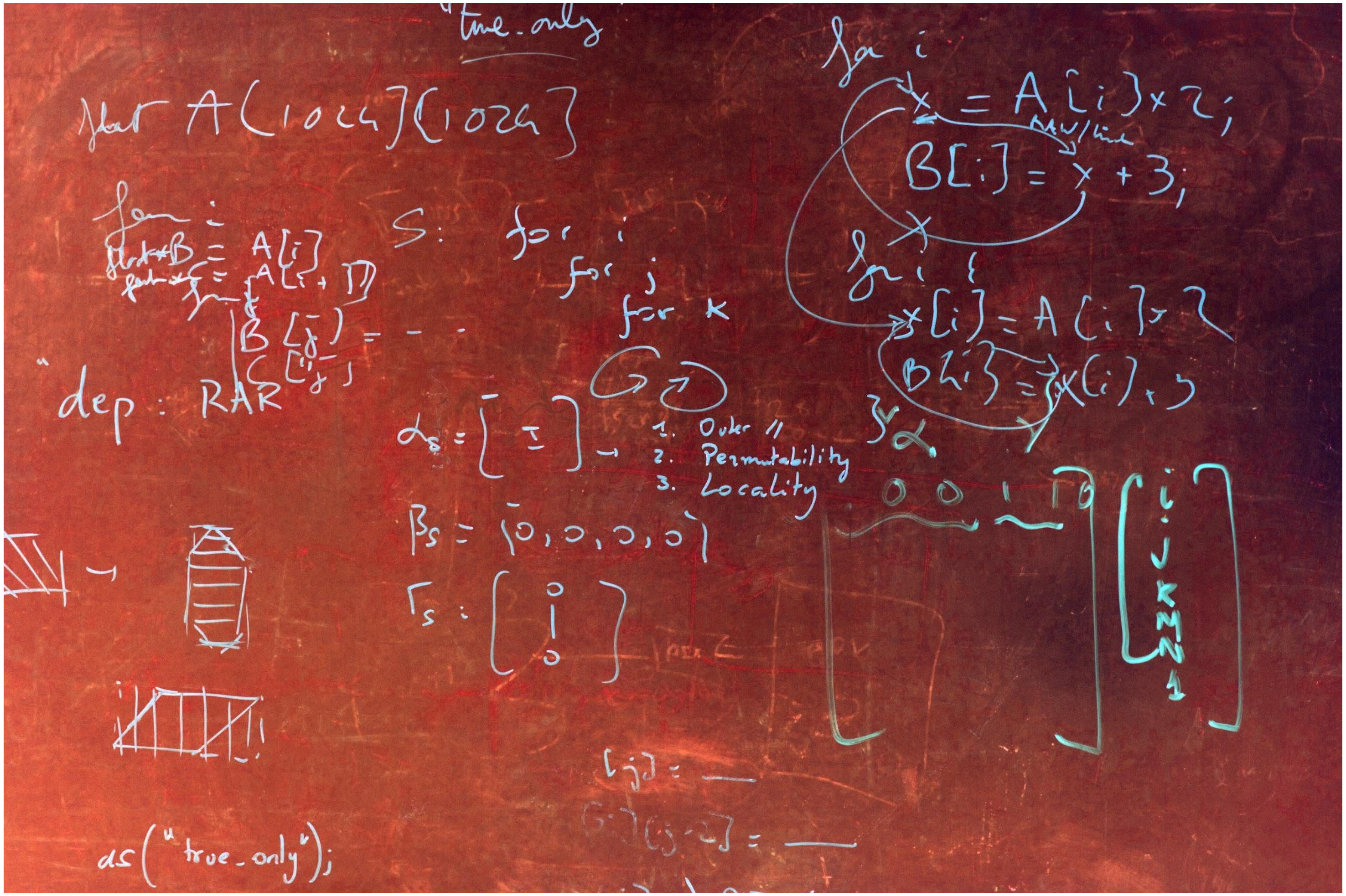
Late Divergence Analysis

How to detect machine regions where threads diverge



Late Divergence Analysis
How to detect machine regions where threads diverge

Are all threads enabled at this point?

Is only one threads enabled ?

Where did threads diverge?

Where will they converge?

Late Divergence Analysis

Outline

A motivating example

Existing LLVM infrastructure

Custom pipeline and data model

Extensions and applications

Summary

A motivating example

A motivating example

SIMT machine (eg GPU)

warp 1

warp 2

warp 3

```
loop:
  a[i] = foo(b) + bar(c)
  i++
  if i < count, br loop

foo:
  ...

bar:
  ...
```

```
t1: i = 5
t2: i = 0
...
```

Predicate flags control enabled threads

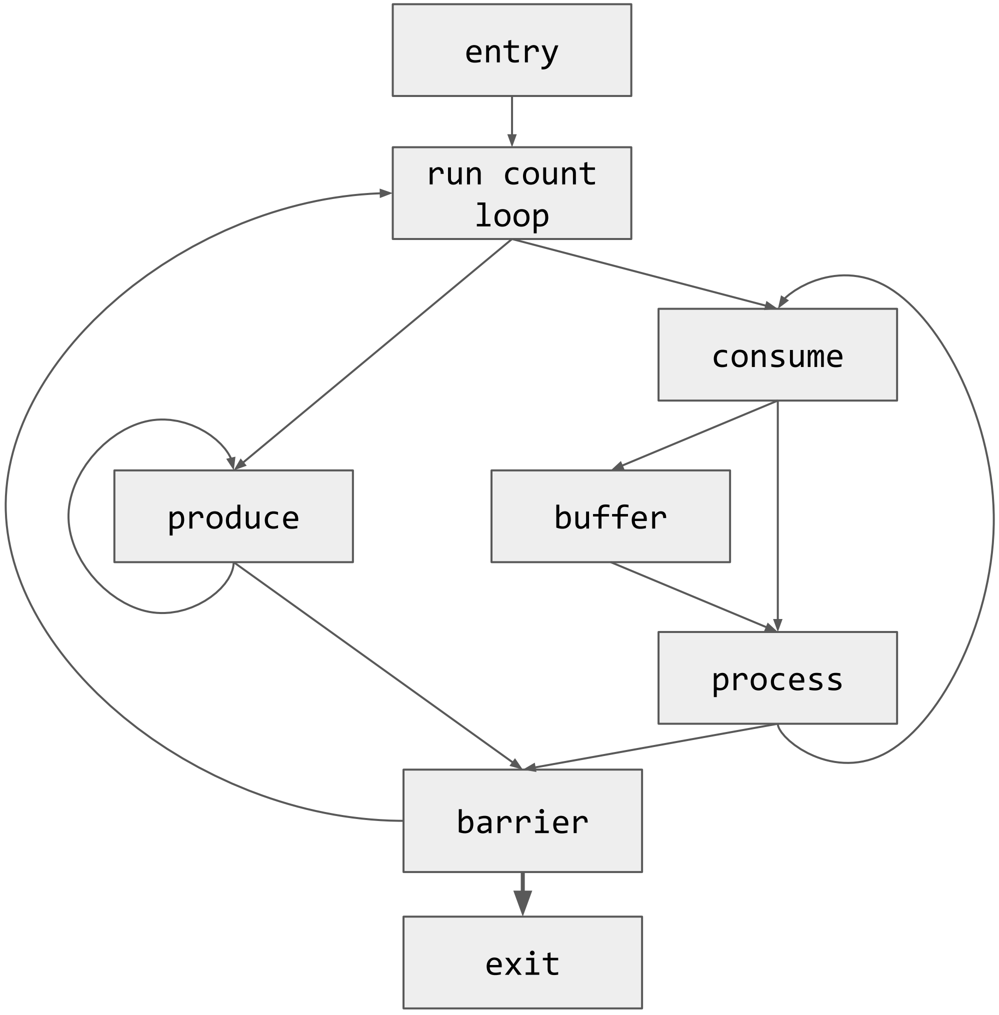
A motivating example

Producer / consumer algorithm on a SIMT machine

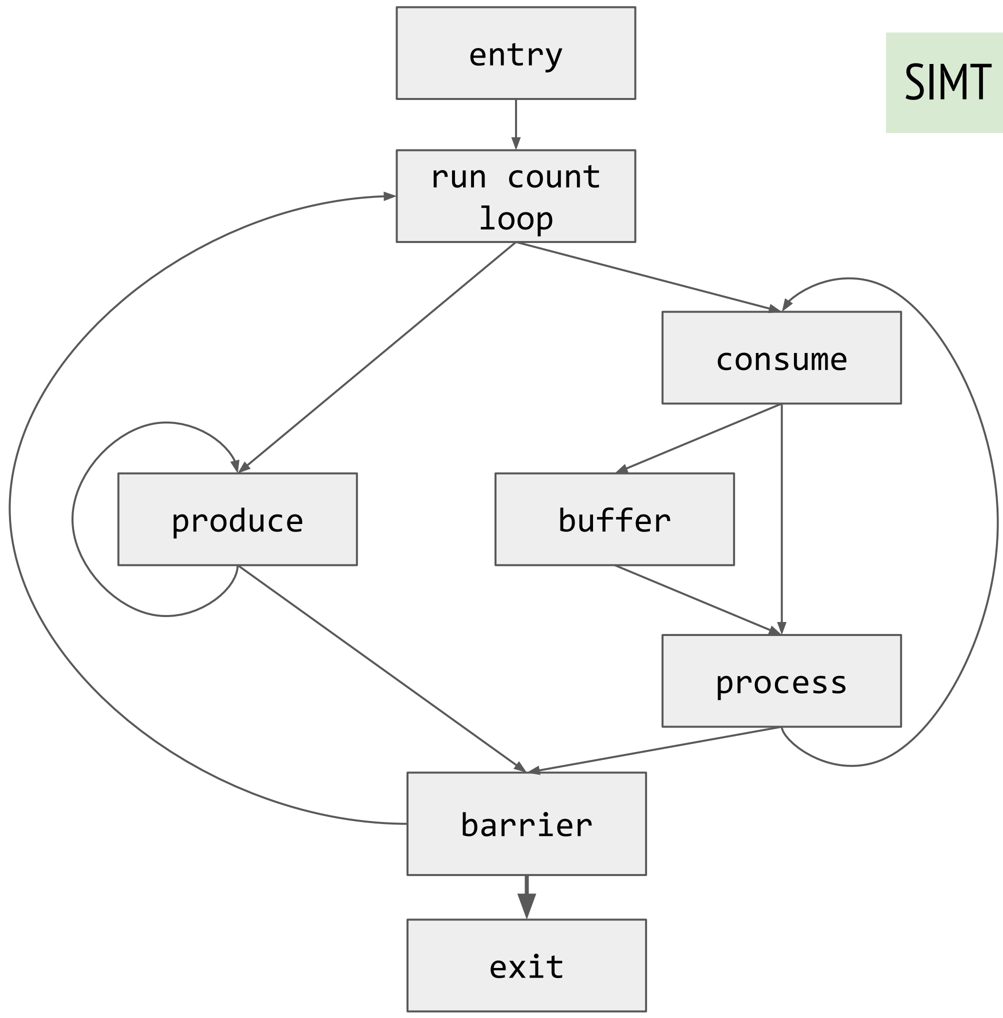
```
void run() {
 for (unsigned i = 0; i < RUN_COUNT; i++) {

   if (is_producer_warp()) {
     while(more_to_produce()) produce();

   } else {
     while(more_to_consume()) {
       if (empty_buffer()) buffer_data();
       process_data();
     }
   }

   barrier();
 }
}
```

A motivating example

Program control flow



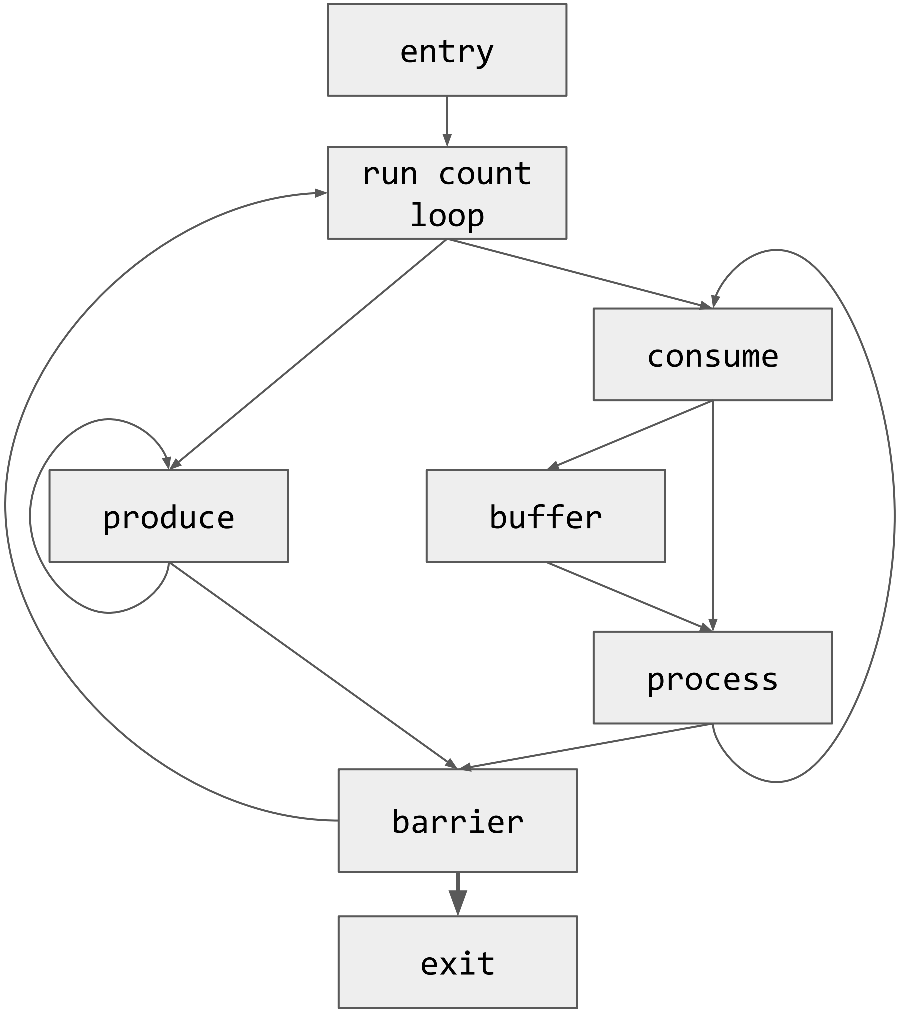
A motivating example
Program control flow



SIMT compatible code?

A motivating example

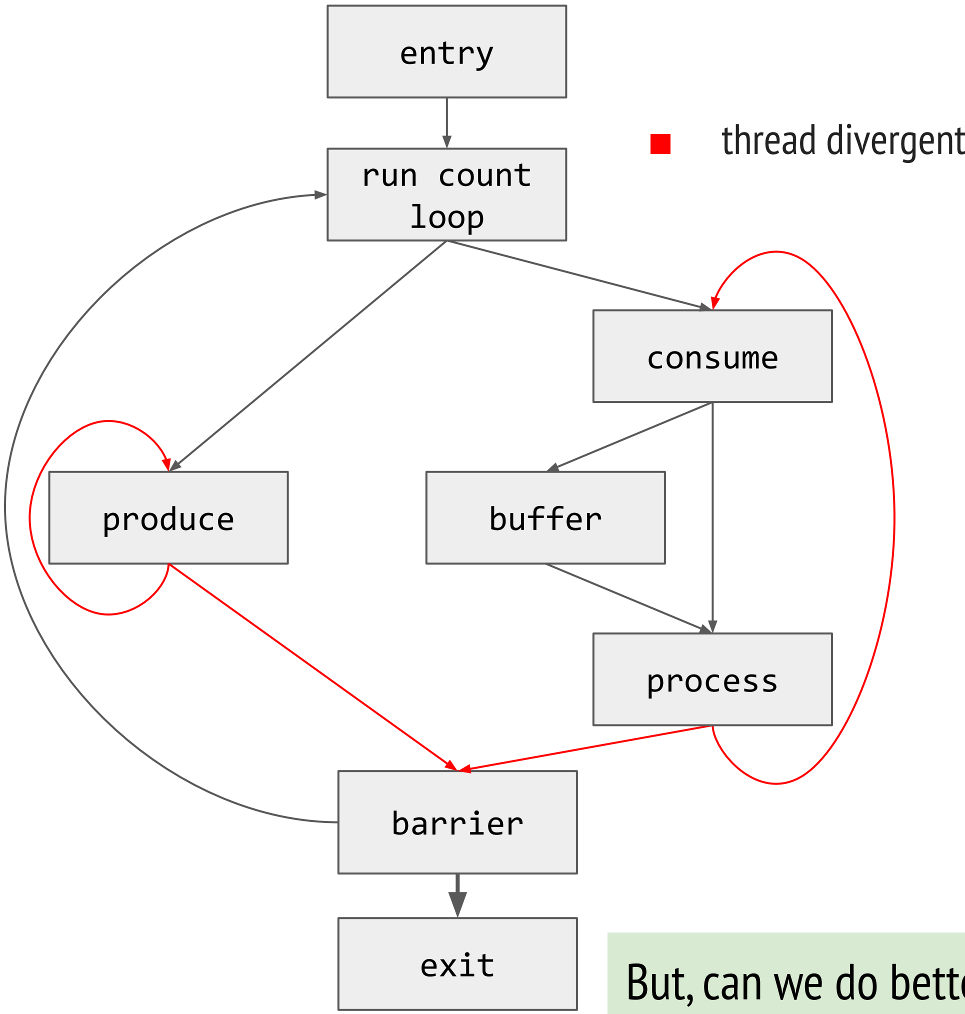
SIMT compatible code




```
entry:
  …
run_count_loop:
  …
   enabled = is_producer

produce:
  …
  enabled = more_to_produce
  if any thread enabled, br produce

consume:
  enabled = is_consumer
  …

buffer:
  enabled = empty_buffer
  …

process:
  enabled = is_consumer
  …
  enabled = more_to_consume
  if any thread enabled, br consume

barrier:
  enabled = i < RUN_COUNT
  …
  i += 1
  enabled = i < RUN_COUNT
  if any thread enabled, br run_count_loop

exit:
```

A motivating example

SIMT compatible code




```
entry:
  …
run_count_loop:
  …
   enabled = is_producer

produce:
  …
  enabled = more_to_produce
  if any thread enabled, br produce

consume:
  enabled = is_consumer
  …

buffer:
  enabled = empty_buffer
  …

process:
  enabled = is_consumer
  …
  enabled = more_to_consume
  if any thread enabled, br consume

barrier:
  enabled = i < RUN_COUNT
  …
  i += 1
  enabled = i < RUN_COUNT
  if any thread enabled, br run_count_loop

exit:
```

But, can we do better?

LLVM infrastructure

Existing infrastructure

LLVM IR divergence

Divergent values pass

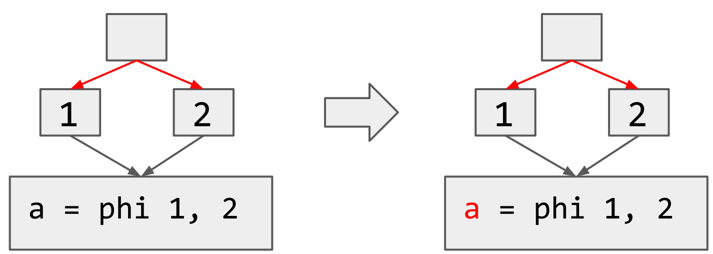
Targets specify divergence sources

- thread id, iteration variable, memory, etc.

Pass propagates divergence to dependent values

```
a = b + c
```

```
a = b + c
```



Existing infrastructure

Draw backs

IR only

Instruction level analysis

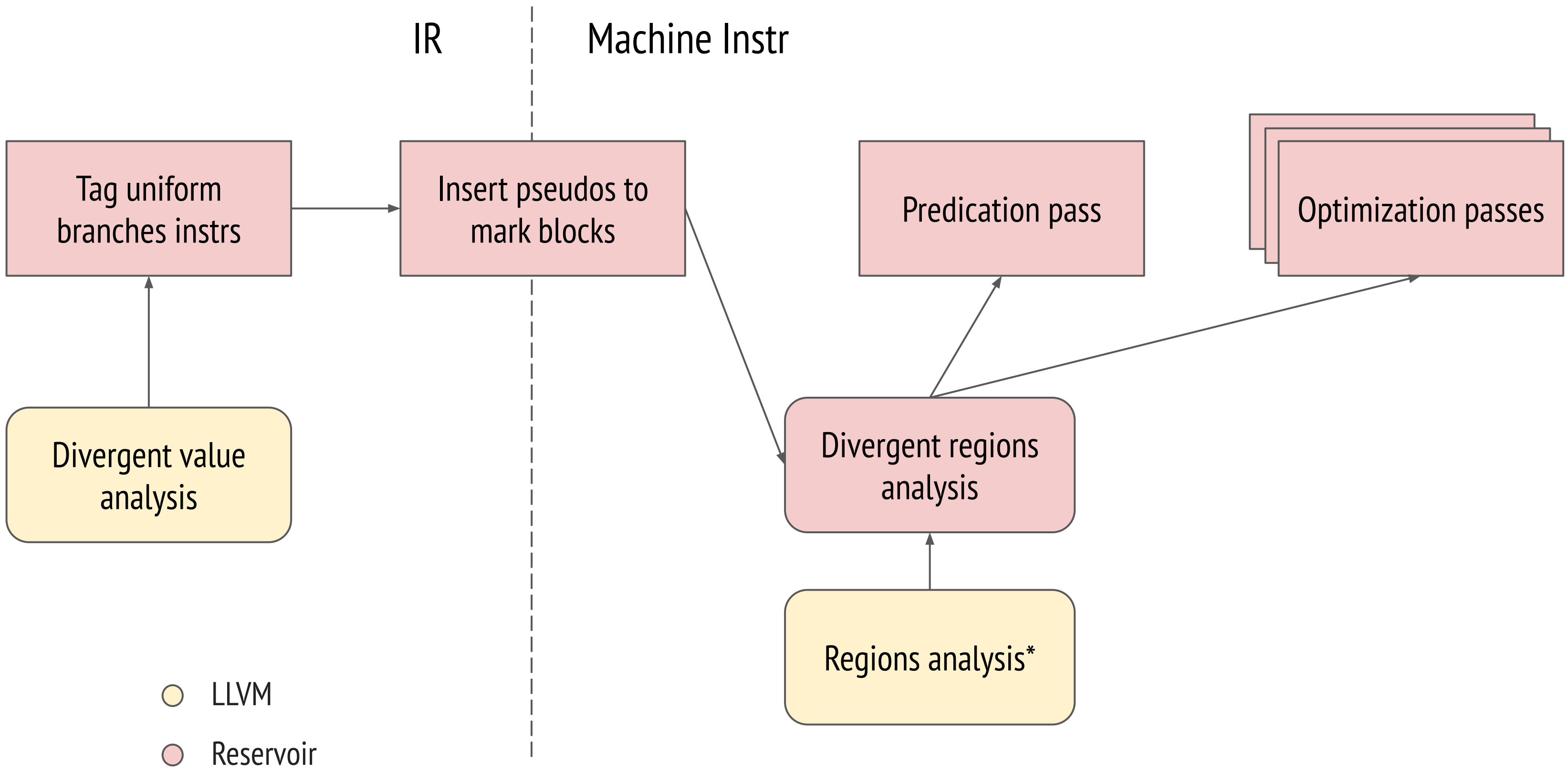
- what about block or region level?

Can’t answer the questions

- are all threads enabled at this point?
- is only one thread enabled ?

Custom pipeline

Custom pipeline
Overview



Custom pipeline

Region analysis

LLVM regions

- Program Tree Structure [Johnson, Pearson and Pingali 1994]

Bounded by entry and exit blocks

Custom pipeline

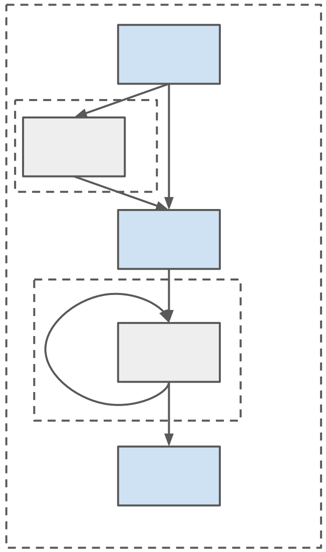
Region analysis

LLVM regions

- Program Tree Structure [Johnson, Pearson and Pingali 1994]

Bounded by entry and exit blocks

Control dependent equivalence



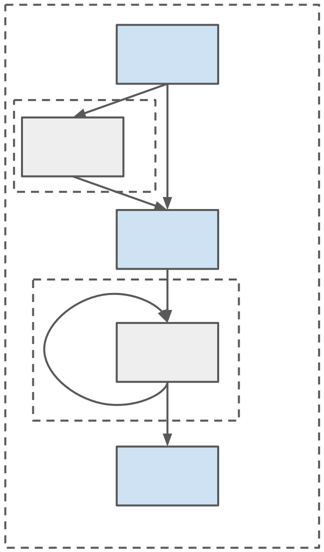
Custom pipeline
Region analysis

LLVM regions

- Program Tree Structure [Johnson, Pearson and Pingali 1994]

Bounded by entry and exit blocks

Control dependent equivalence



Needed custom regions for increase granularity, modified CFGs

Detail appendix - A1

Custom pipeline and data model

Divergent regions

Threads diverge from immediate dominator region

Custom pipeline and data model

Divergent regions

Threads diverge from immediate dominator region

Propagation:

- Incoming divergent edge(s) -> divergent

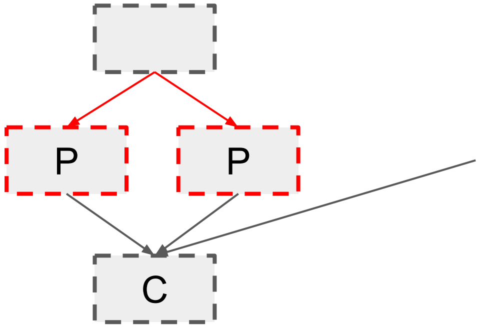
Custom pipeline and data model
Divergent regions

Threads diverge from immediate dominator region

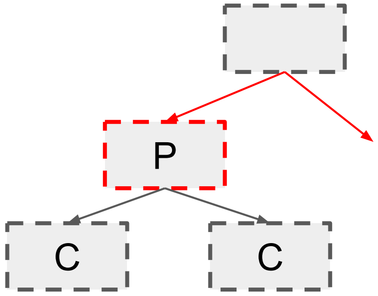
Propagation:

- Incoming divergent edge(s) -> divergent
- Parent divergent -> child divergent, unless

child post-dominates divergent source(s)



parent dominates child



Custom pipeline and data model

Divergent regions

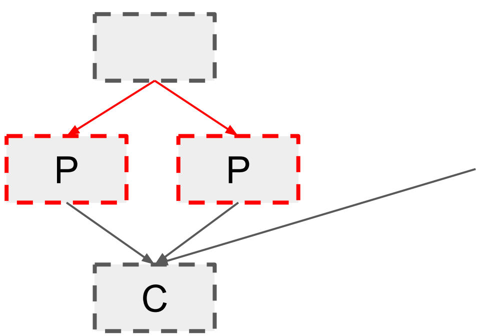
Threads diverge from immediate dominator region

Propagation:

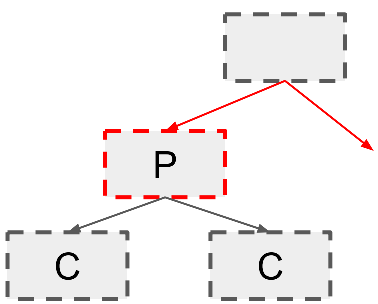
- Incoming divergent edge(s) -> divergent
- Parent divergent -> child divergent, unless

Propagation example in appendix - A2

child post-dominates divergent source(s)



parent dominates child



Custom pipeline and data model

Region properties

Divergent

- threads **can** diverge from immediate dominator region

Uniform

- threads **never** diverge from immediate dominator region

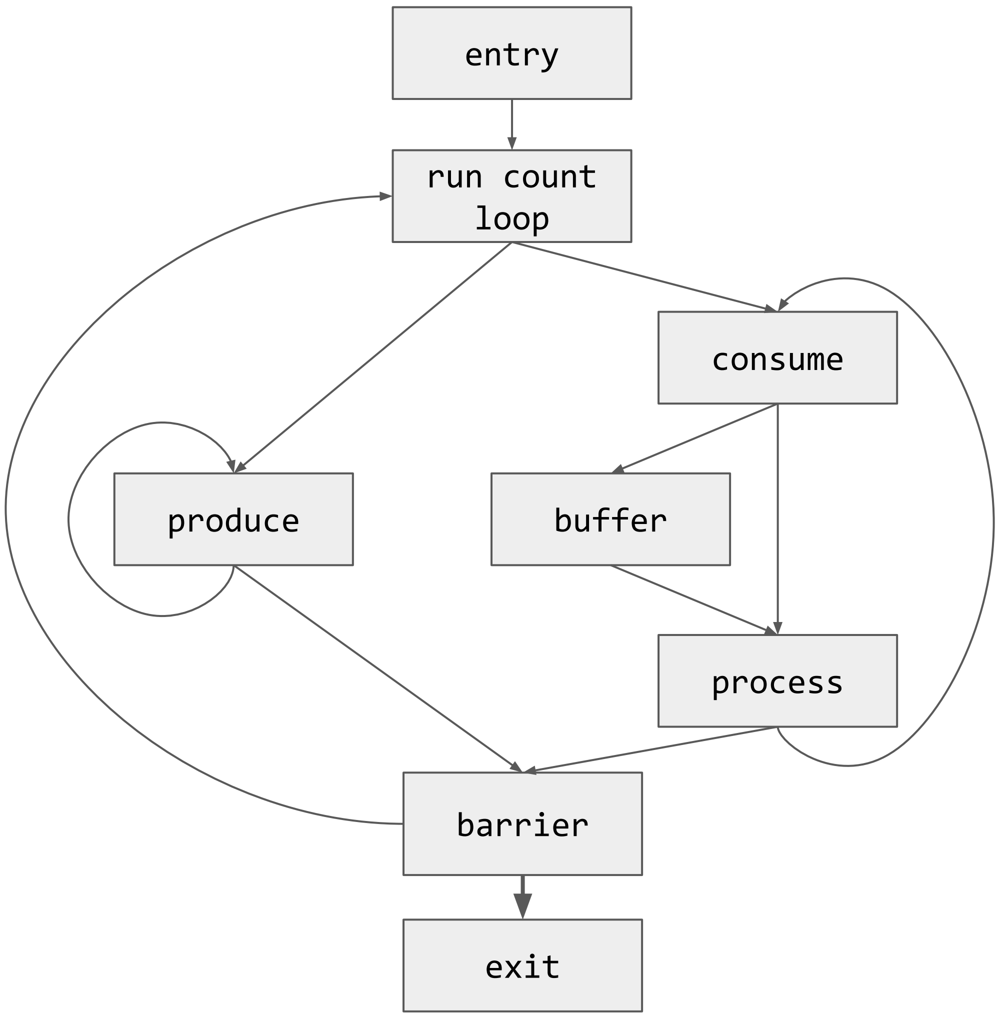
Function uniform

- every dominator is uniform
- threads **never** diverge from function entry

Example revisited

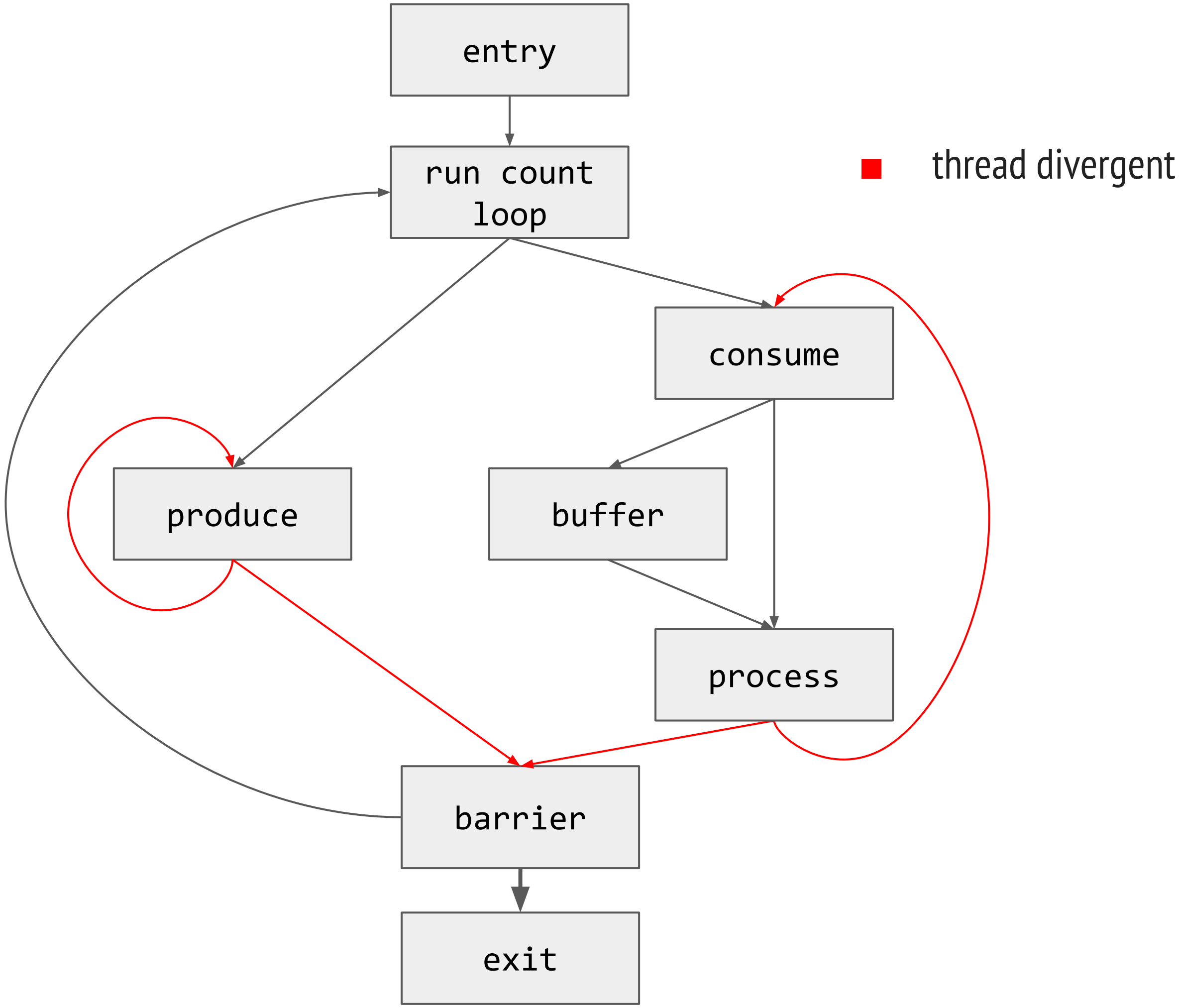
Example revisited

Program control flow



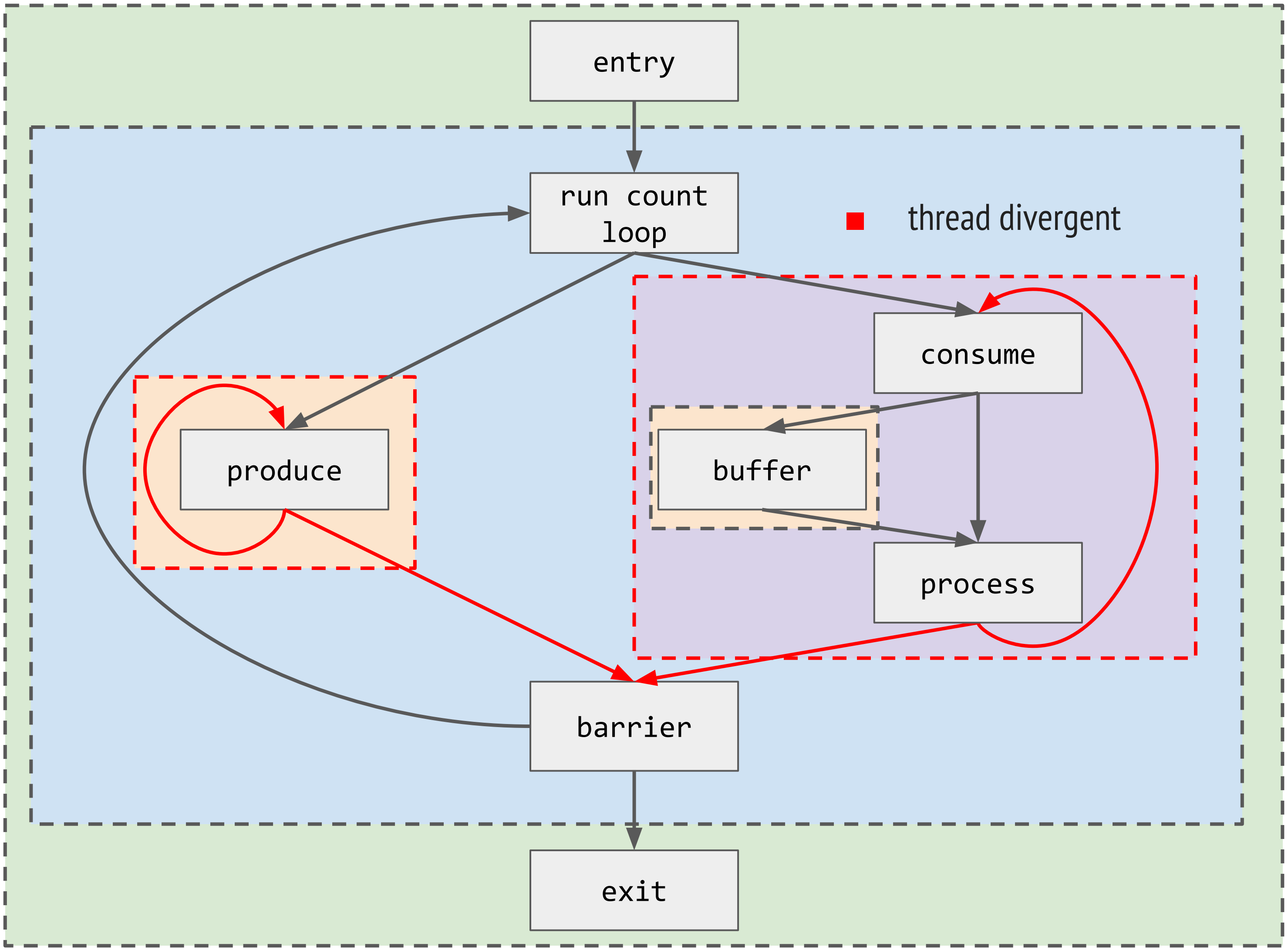
Example revisited

Program control flow



Example revisited

Divergent regions



Example revisited

Emitting SIMT compatible code

Only divergent regions need predicates

Uniform branches can be left in

Example revisited

Emitting SIMT compatible code

Only divergent regions need predicates

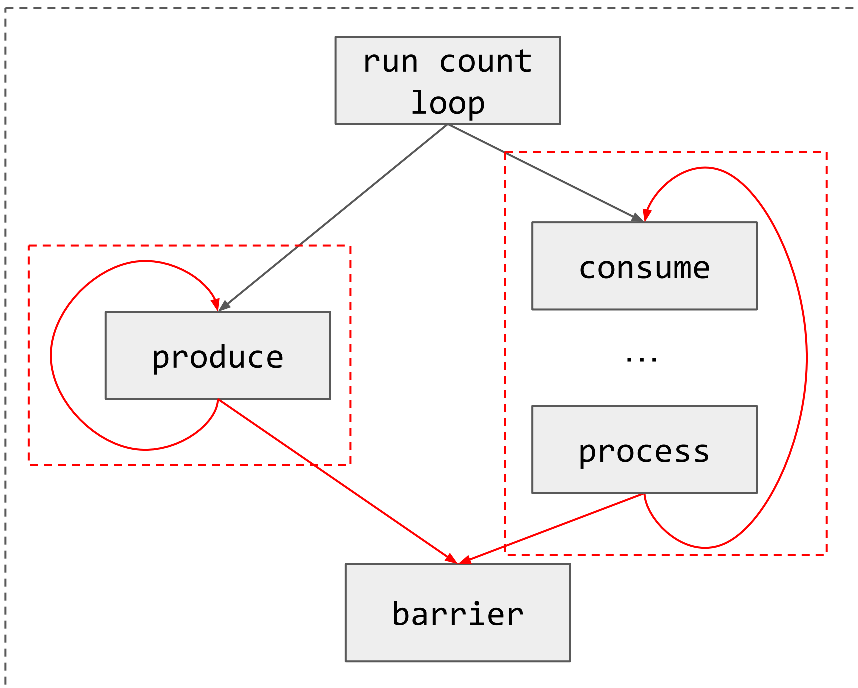
Uniform branches can be left in*

Example in appendix - A3

Example revisited
Emitting SIMT compatible code

Only divergent regions need predicates

Uniform branches can be left in



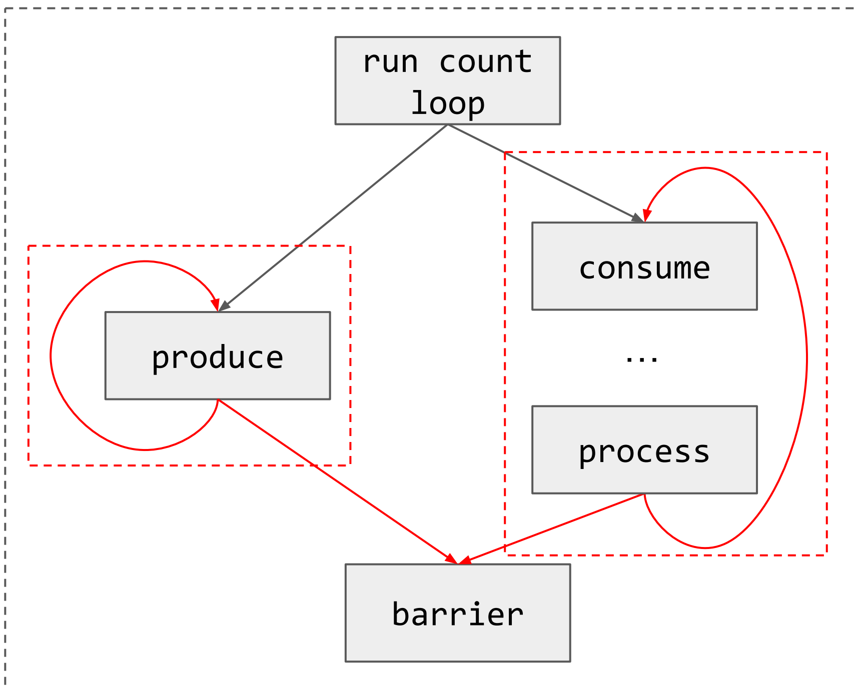

```
run_count_loop:
  …
  enabled = is_producer

produce:
  …
  enabled = more_to_produce
  if any thread enabled, br produce

consume:
  enabled = is_consumer
  …

process:
  …
  enabled = more_to_consume
  if any thread enabled, br consume

barrier:
  …
```

Example revisited

Emitting SIMT compatible code

Only divergent regions need predicates

Uniform branches can be left in



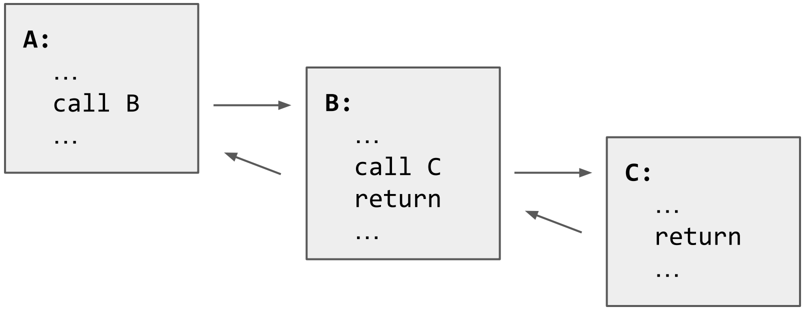

```
run_count_loop:
  …
  if is_consumer, br consume

produce:
  …
  enabled = more_to_produce
  if any thread enabled, br produce
  br barrier

consume:
  …

process:
  …
  enabled = more_to_consume
  if any thread enabled, br consume

barrier:
  …
```

Extensions / applications

Extensions and applications

Applications

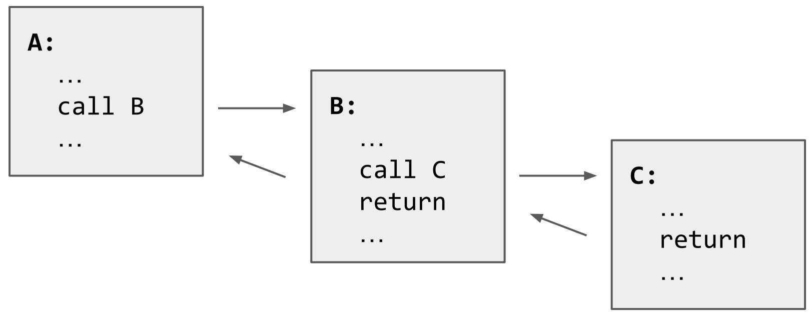
- hardware loop candidates
- tail call optimization candidates



Extensions and applications

Applications

- hardware loop candidates
- tail call optimization candidates



Extension

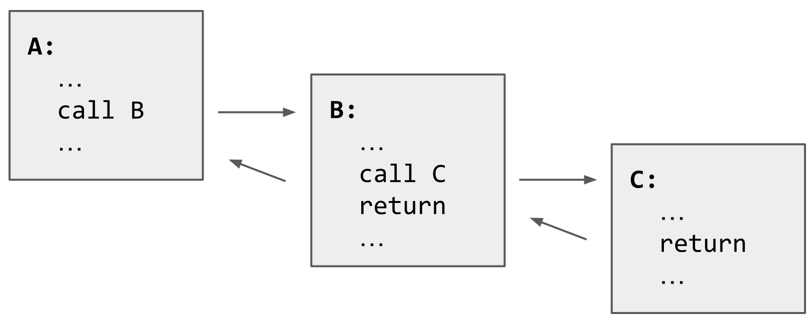
- thread uniqueness

```
while(more_to_produce())
 produce();

if (lead_produce(warp_id, thread_id))
 notify_completion();
```

Extensions and applications

Applications

- hardware loop candidates
- tail call optimization candidates



Extension

- thread uniqueness

```
while(more_to_produce())
 produce();

if (lead_produce(warp_id, thread_id))
 notify_completion();
```

Details in appendix - A5

Summary

Identifies SESE regions that are

- divergent, uniform, function uniform, thread unique

Analysis combines

- IR divergence, region detection, divergence / uniqueness propagation

Use cases

- predication, tail call optimizations, hardware loops, more?

Appendix

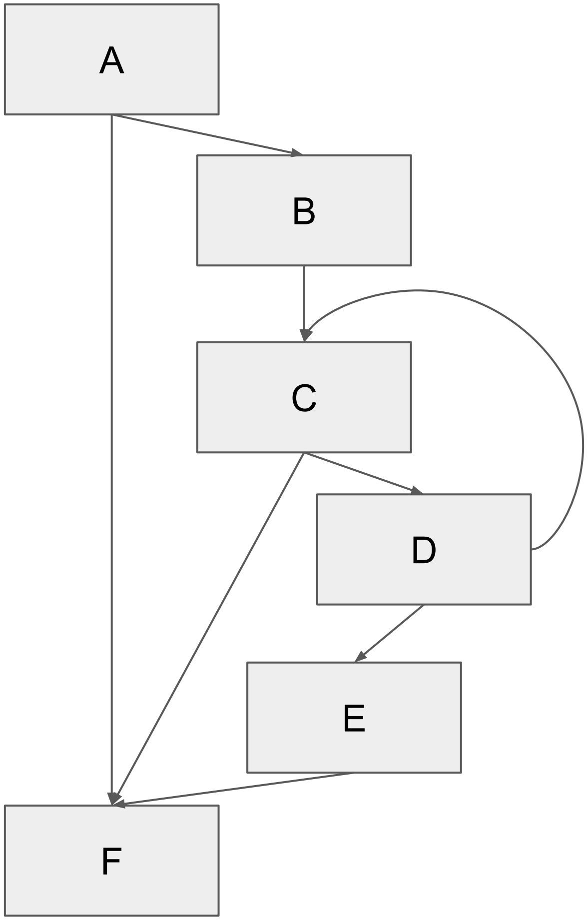
A1

A1 - LLVM regions vs custom region

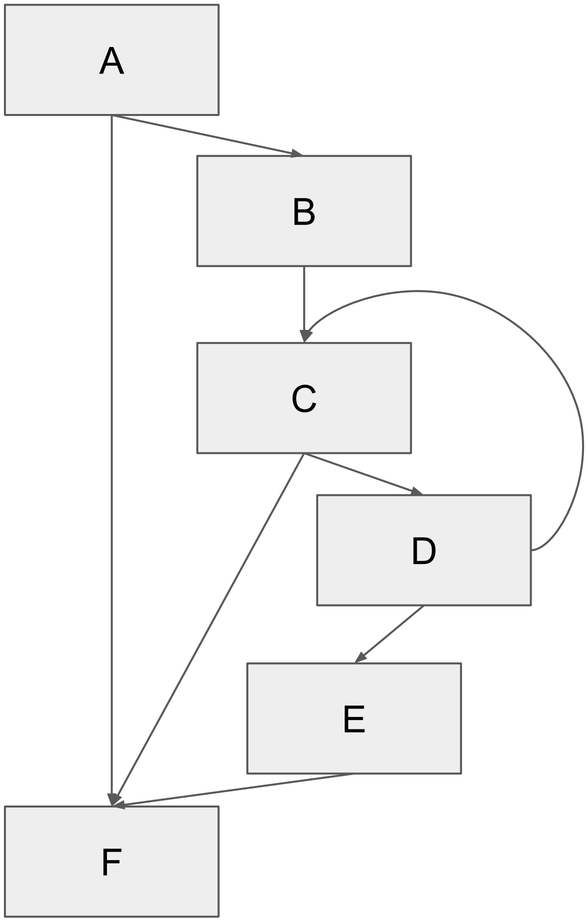
Example from the gcc testsuite: `gcc.c-torture/execute/pr54985.c`

```
int foo(ST *s, int c) {
 int first = 1;
 int count = c;
 ST *item = s;
 int a = s->a;
 int x;

 while (count--)
   {
     x = item->a;
     if (first)
       first = 0;
     else if (x >= a) {
       return 1;
     }
     a = x;
     item++;
   }
 return 0;
}
```

A1 - LLVM regions vs custom region



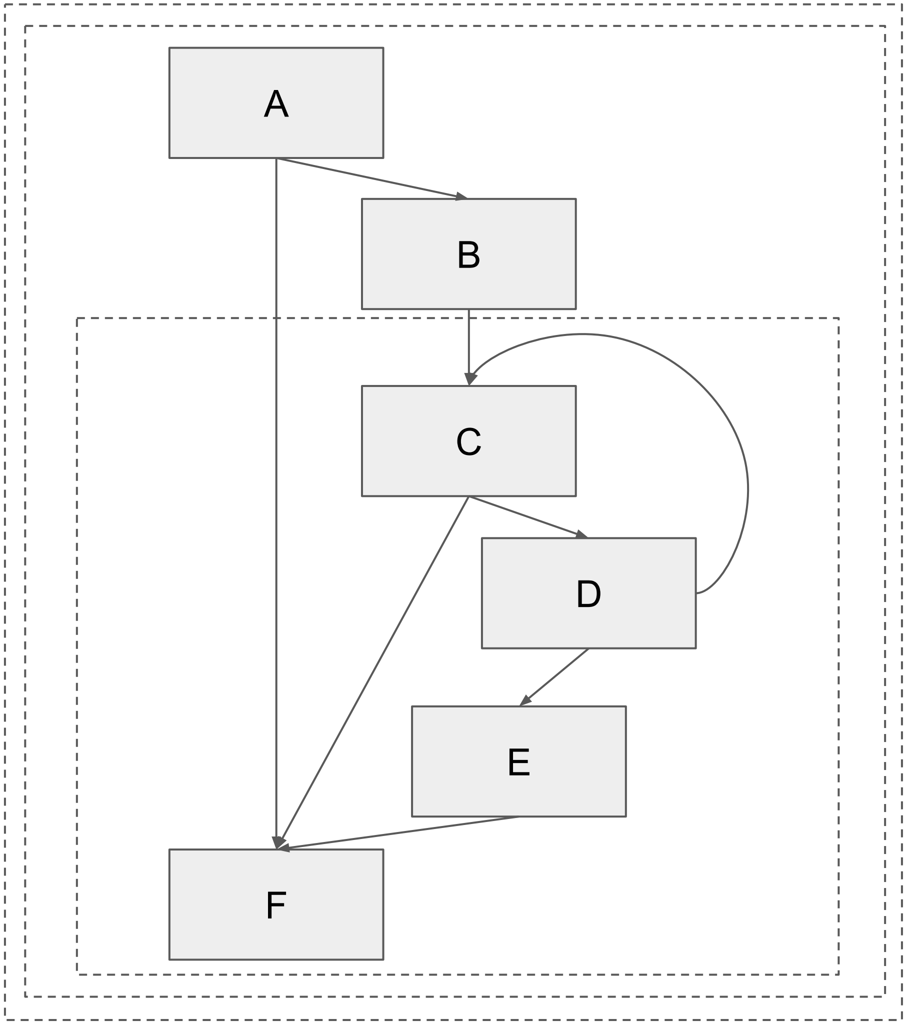
A1 - LLVM regions vs custom region



LLVM regions

```
Region tree:
[0] A => <Function Return>
  [1] A => F
    [2] C => F
```

A1 - LLVM regions vs custom region

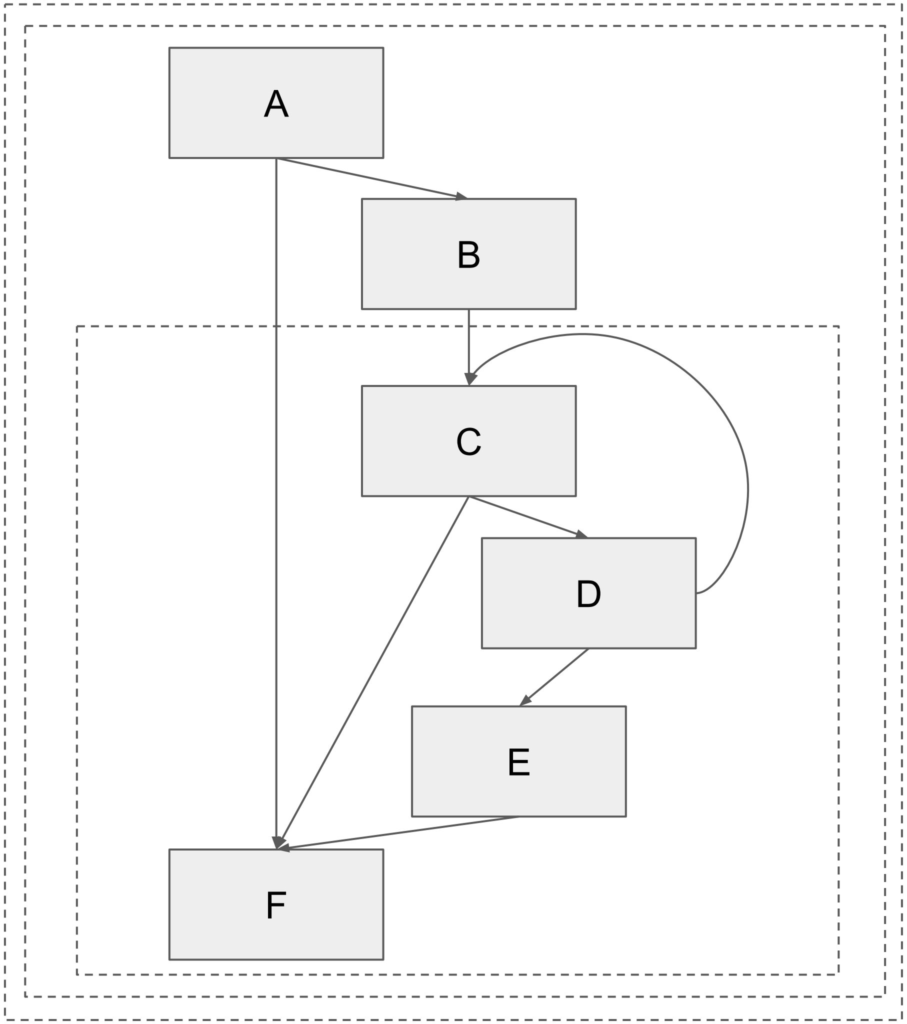


LLVM regions

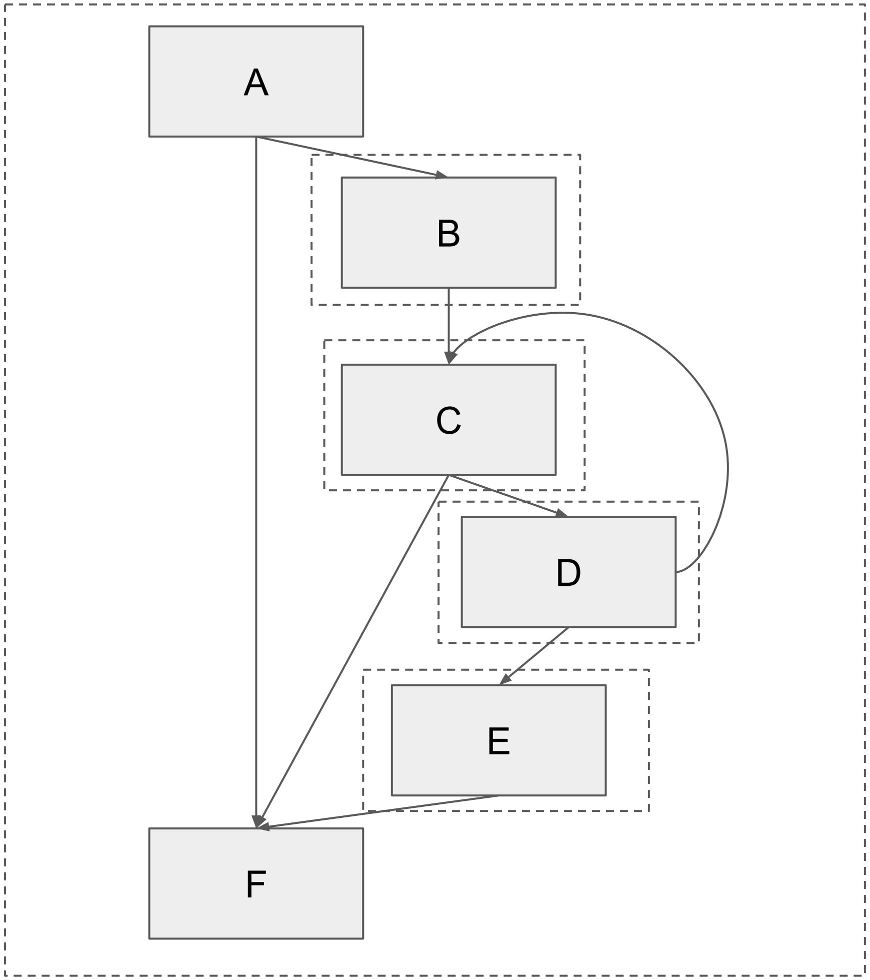
```
Region tree:
[0] A => <Function Return>
  [1] A => F
    [2] C => F
```

A1 - LLVM regions vs custom region

LLVM regions



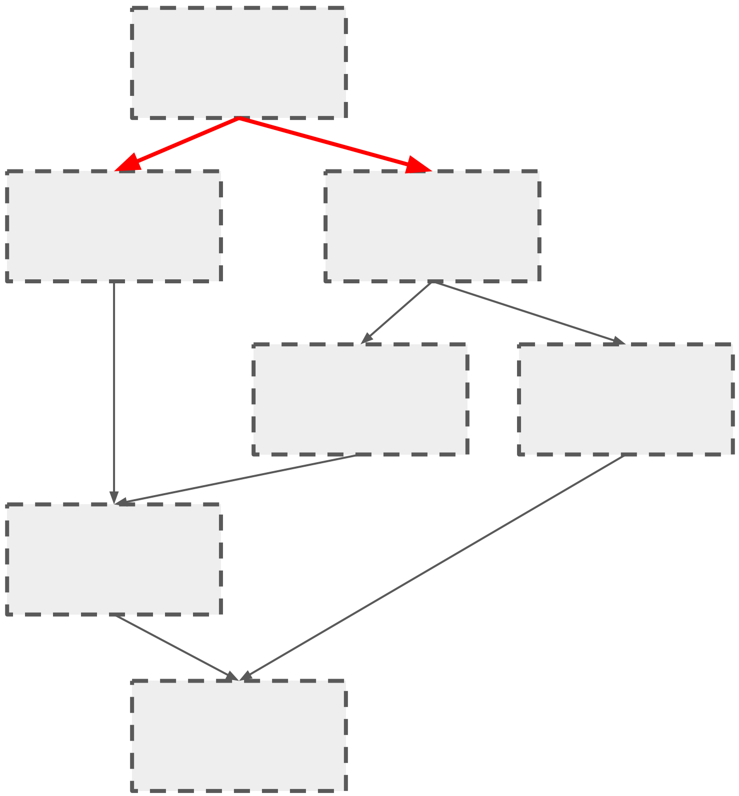
Custom regions



A2

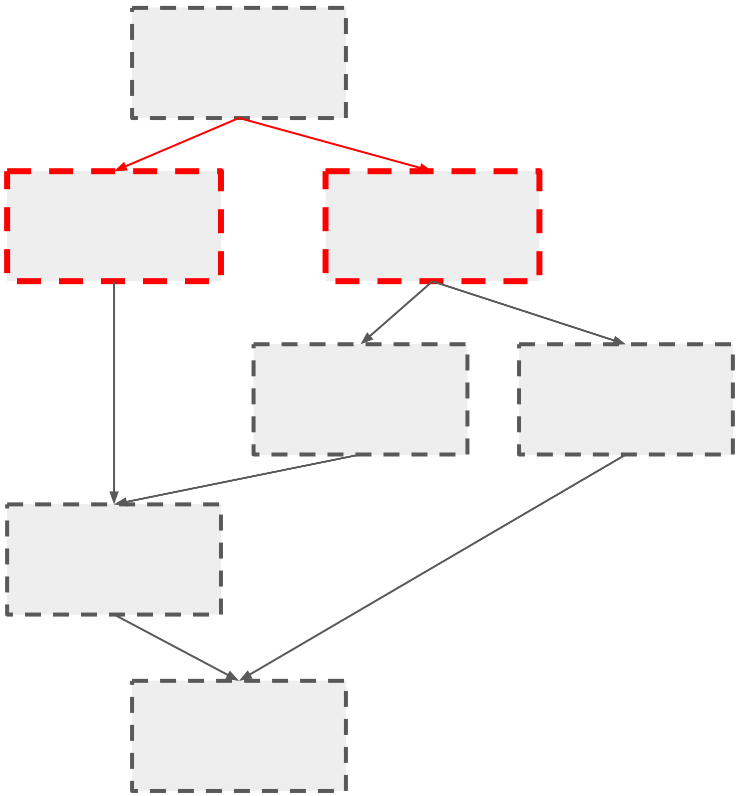
A2 - Propagating divergence to SESE regions

Example 1



A2 - Propagating divergence to SESE regions

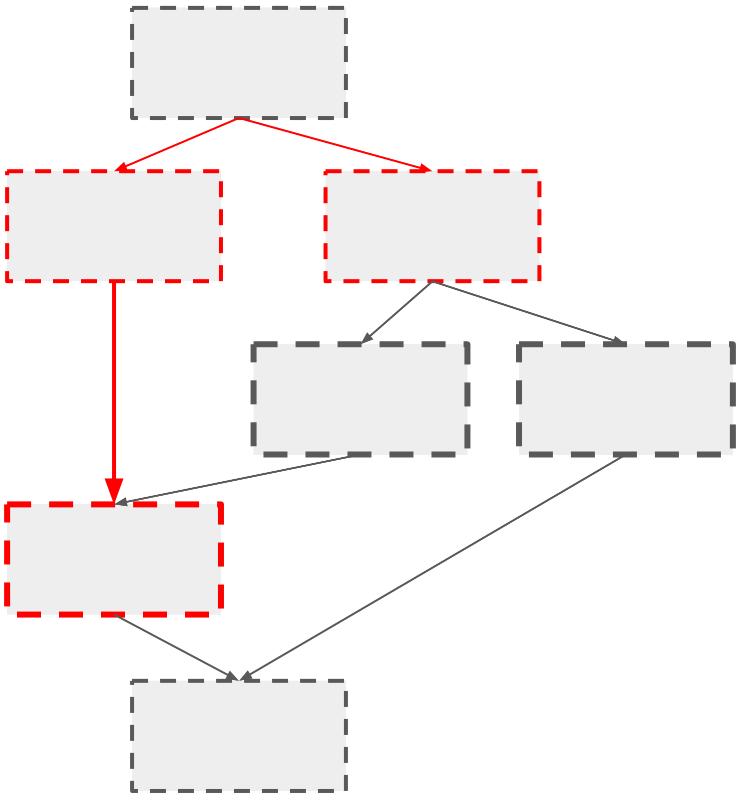
Example 1



Regions with incoming divergent edge(s) -> divergent

A2 - Propagating divergence to SESE regions

Example 1



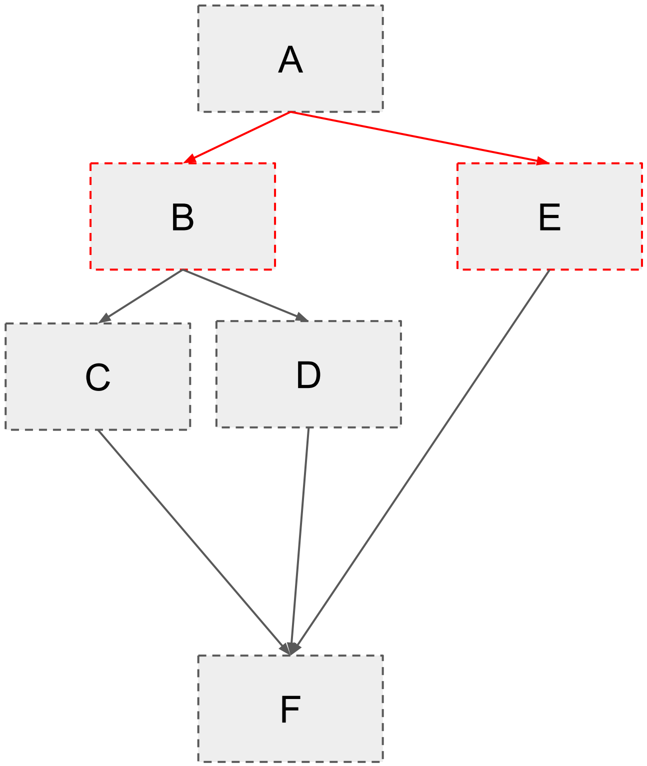
Divergence propagates from regions to child regions, unless

- child dominates divergent source(s)
- parent dominates child

A3

A3 - Generating SIMT code

Restricting branch aheads

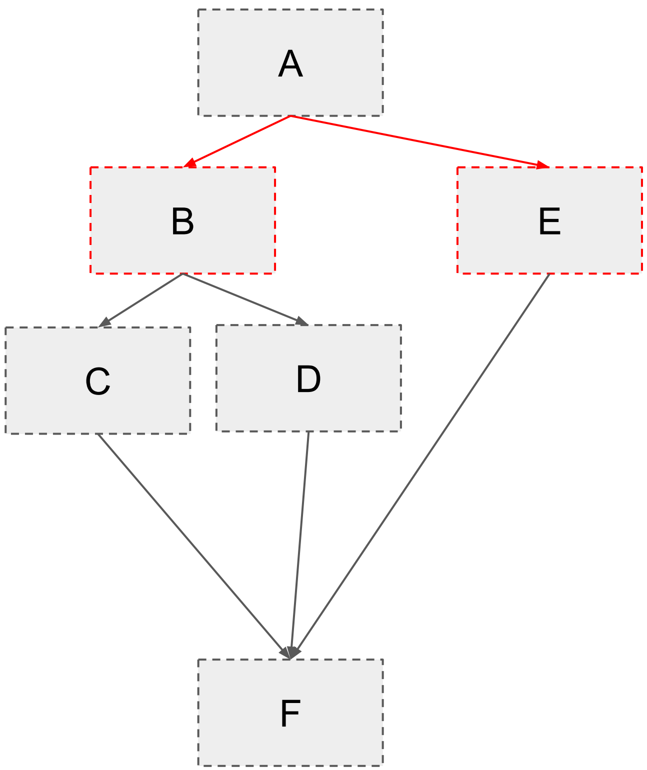


Uniform branches can be left in

```
A:
  ...
B:
  enable = do_b
  ...
  if do_d, br D
C:
  ...
  br F
D:
  ...
  br F
E:
  enable = do_e
  ...
F:
  ...
```

Will skip over E for threads that need to execute it

A3 - Generating SIMT code
Restricting branch aheads



Uniform branches can be left in

- but can only branch as far ahead as they dominate

```
A:
  ...
B:
  enable = do_b
  ...
  if do_d, br D
C:
  ...
  br E
D:
  ...
E:
  enable = do_e
  ...
F:
  ...
```

A4

A4 - Custom pipeline

IR pass

Tag uniform branches

```
DA = &getAnalysis<DivergenceAnalysis>();

for (BasicBlock &B : F) {
 Instruction *Term = B.getTerminator();

 if ((isa<BranchInst>(Term) || isa<SwitchInst>(Term))
     && DA->isUniform(Term)) {

     Term->setMetadata("my-target-divergent-branch",
                       MDNode::get(Term->getContext(), {}));
 }
}
```

A4 - Custom pipeline

Add pseudo instructions

Before selection

```
run_count_loop:
 ...
 br i1 %cond, label %consume, label %produce_entry, !my.target.uniform !2
```

After selection

```
bb.0.run_count_loop:
 ...
 P_UNIFORM_MBB
 I_BR %2:cond, %bb.1,consume, %bb.3.produce_entry
```

A5

A5 - An extension

Thread unique property

Only one thread enabled

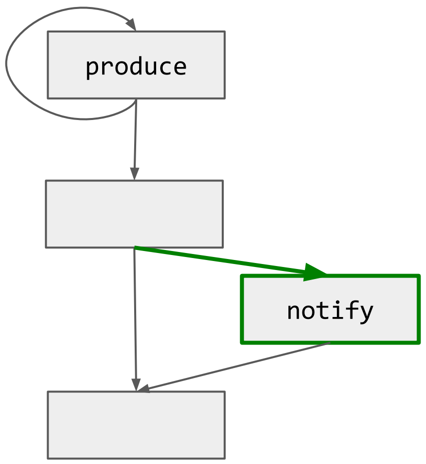
```
while(more_to_produce())
 produce();

if (lead_produce(warp_id, thread_id))
 notify_completion();
```

A5 - An extension

Thread unique property

Only one thread enabled

```
while(more_to_produce())
 produce();

if (lead_produce(warp_id, thread_id))
 notify_completion();
```

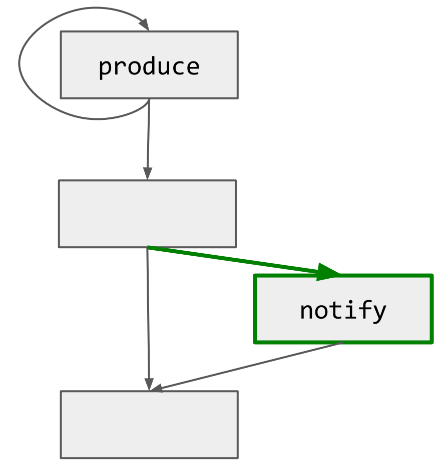


A5 - An extension

Thread unique property

Only one thread enabled

```
while(more_to_produce())
 produce();

if (lead_produce(warp_id, thread_id))
 notify_completion();
```



Value propagation:

- Thread unique sources
- Operations that persevere uniqueness

Region propagation:

- One thread unique entry edge-> thread unique
- IDom unique -> child unique

A5 - An extension

Thread unique property

Use case: predication

Regions dominated by thread unique regions

- don’t need a predicate
- all branches can be left in*

A6

A6 - Other applications

Uniform regions

Use case: hardware loops

Some hardware loops may require uniform loop conditions

Loops in uniform regions are candidates

A6 - Other applications

Function uniform regions

Use case: tail call optimizations

Standard tail call optimizations can be applied when return is in a function uniform region

References

References

[1] Johnson, R., Pearson, D., and Pingali, K. "The program structure tree: computing control regions in linear time." Programming Language Design and Implementation, 1994

[2] Vanhatalo, J., Völzer, H., Koehler, J. "The refined process structure tree." Data Knowl. Eng. 2009, 68, 793–818