Evolving “convergent”: Lessons from Control Flow in AMDGPU

Nicolai Hähnle

Context

- IR: Add convergence control operand bundle and intrinsics
  - https://reviews.llvm.org/D85603
- New control flow implementation in the AMDGPU backend
  - https://github.com/nhaehnle/llvm-project/tree/controlflow-wip-v7

AMD

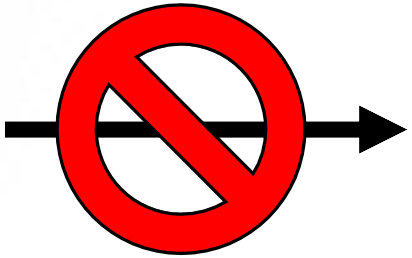
History of 'convergent'

```
barrier()
```

"Wait for all other threads in the threadgroup to reach the same point in the program"

Barriers and loop unswitching

```
bool cond = …;
for (int i = 0; i < 4; ++i) {
    if (cond) {
        A(i)
    } else {
        B(i)
    }
    barrier();
}
```



```
bool cond = …;
if (cond) {
    for (int i = 0; i < 4; ++i) {
        A(i)
    }
    barrier();
} else {
    for (int i = 0; i < 4; ++i) {
        B(i)
    }
    barrier();
}
```

Barriers and loop unrolling

```
bool cond = …;
for (int i = 0; i < 4; ++i) {
    if (cond) {
        A(i)
    } else {
        B(i)
    }
    barrier();
}
```

```
bool cond = …;
if (cond) { A(0) } else { B(0) }
barrier();
if (cond) { A(1) } else { B(1) }
barrier();
if (cond) { A(2) } else { B(2) }
barrier();
if (cond) { A(3) } else { B(3) }
barrier();
```

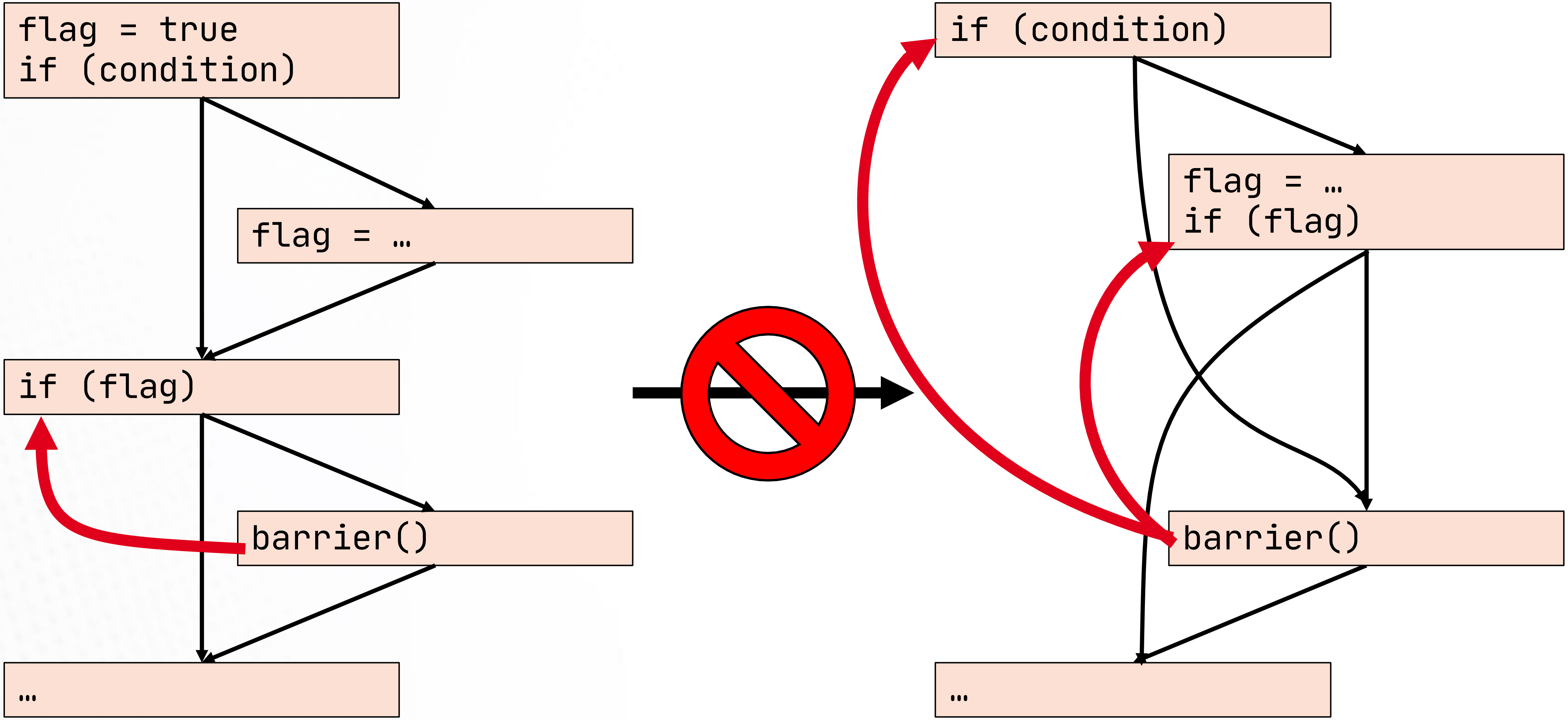
Current definition of ‘convergent’

From LangRef:

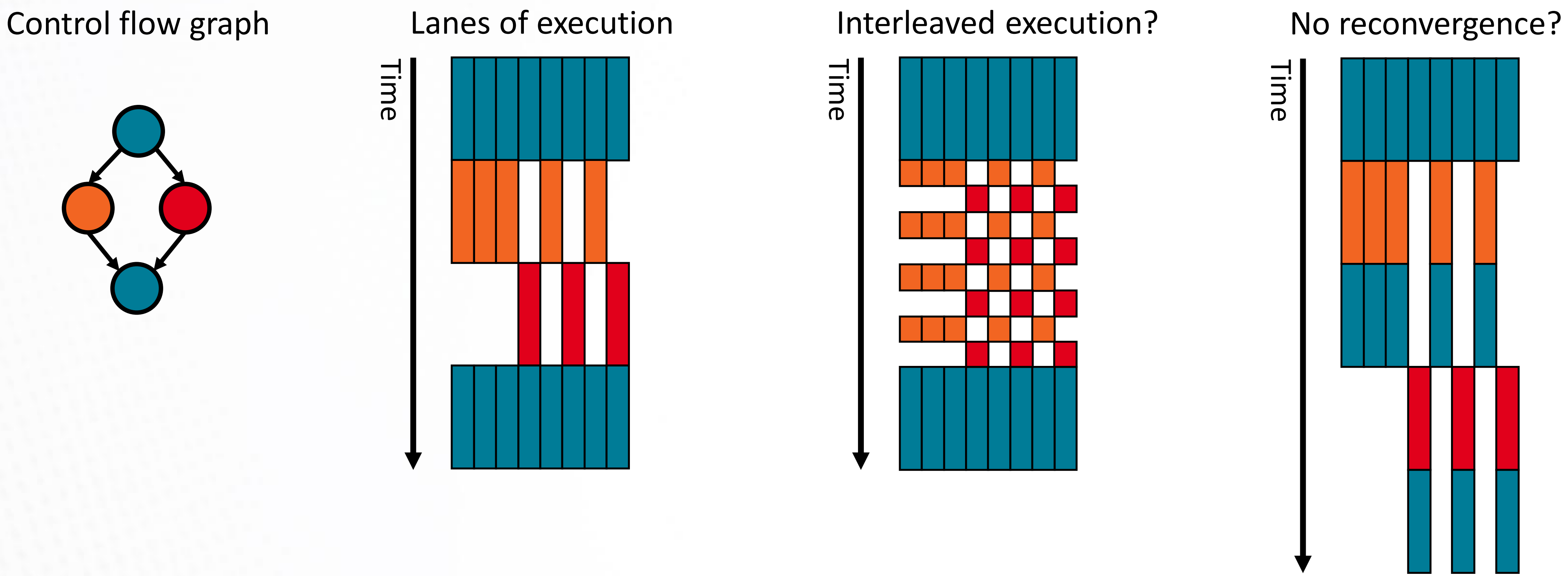
When [convergent] appears on a function, it indicates that calls to this function should not be made control-dependent on additional values.

AMD

~~JumpThreading~~ Convergent has issues



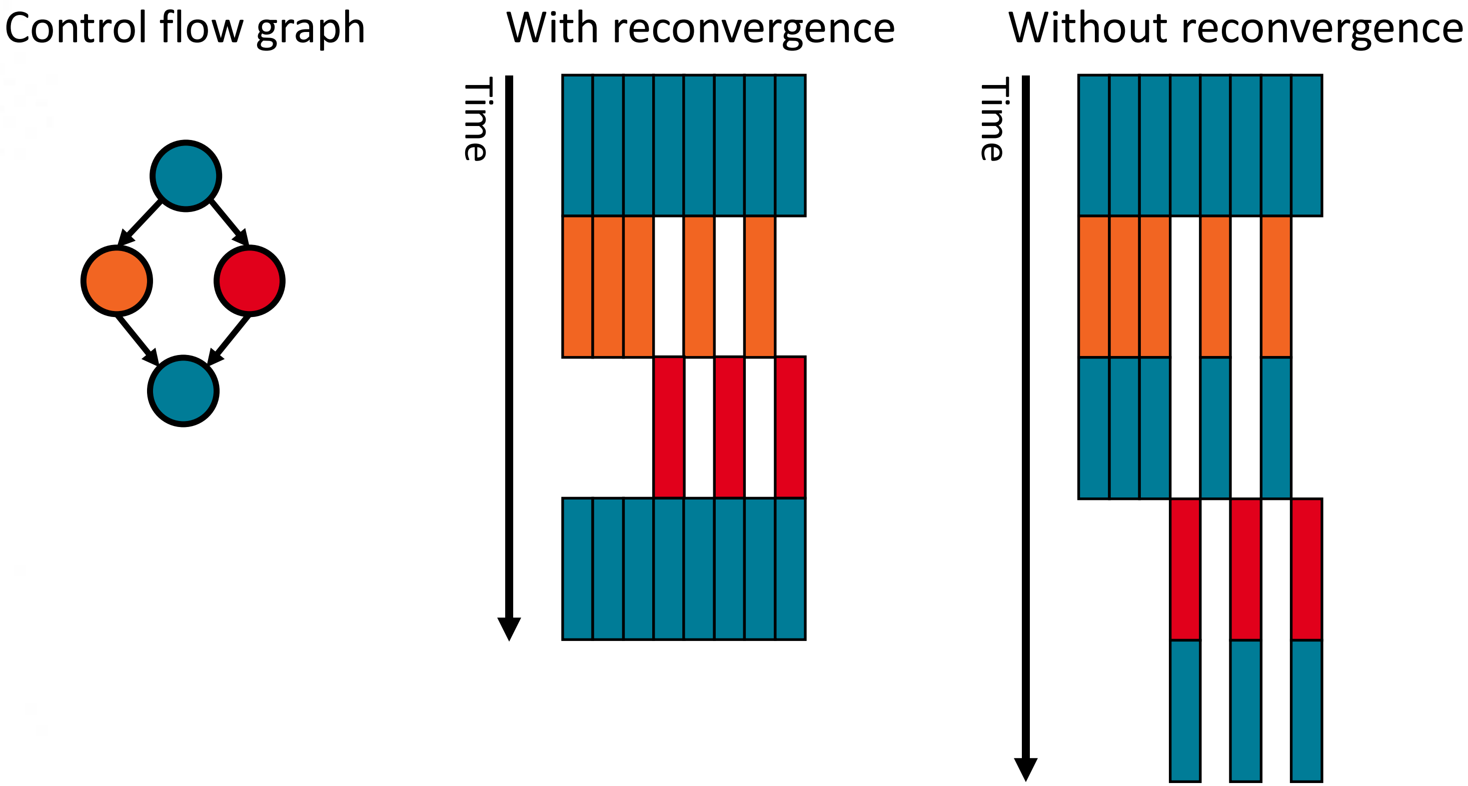
SIMT execution: threads mapped onto lanes of SIMD hardware



- LLVM IR **should not** care about linear temporal orderings
- LLVM IR **must** care about whether threads are converged or not

Why LLVM cares: Cross-lane operations

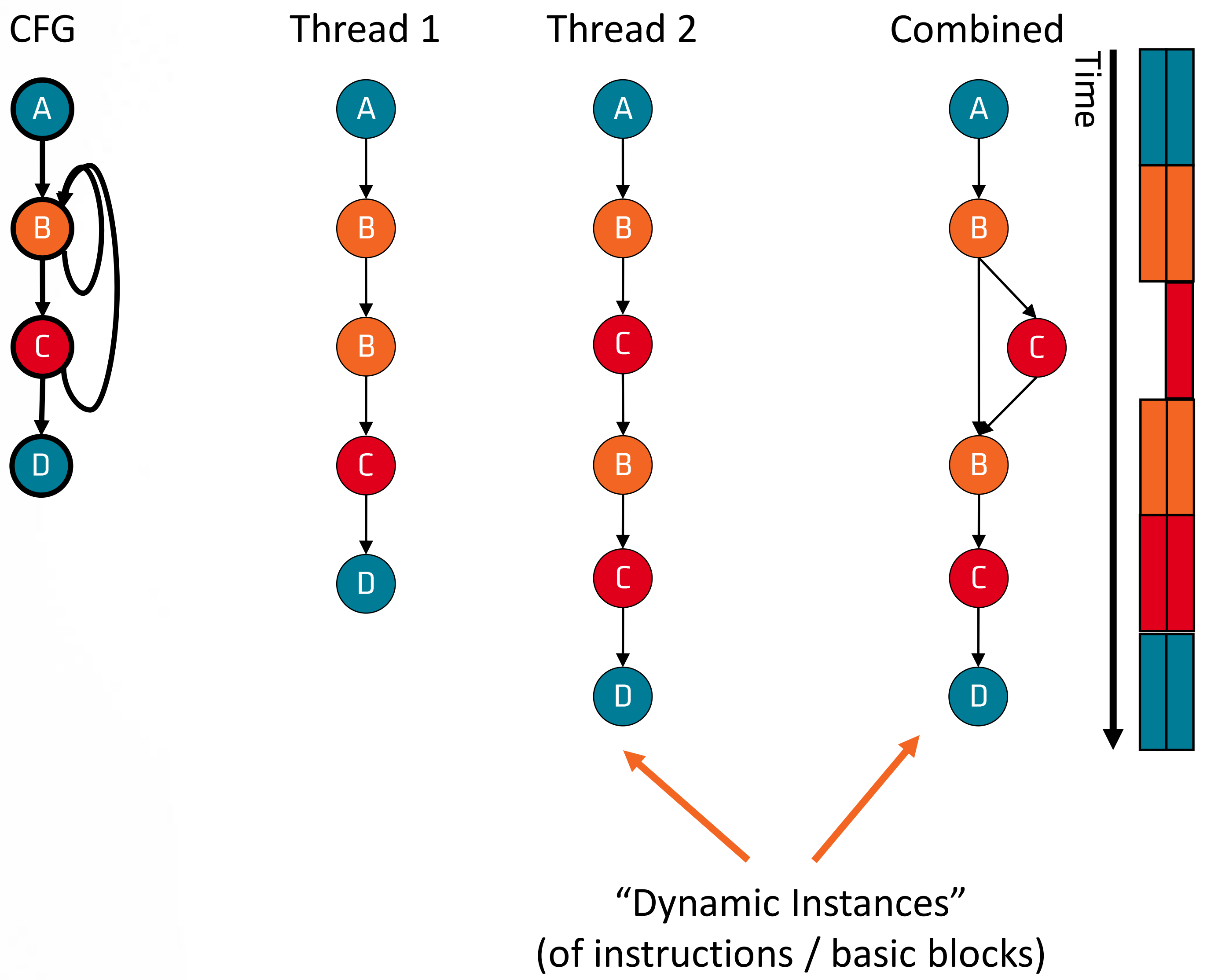
```
bool cond = …;
int value = …;
if (cond) {
    value = foo();
} else {
    value = bar();
}
int sum = subgroupAdd(value);
```



- subgroupAdd computes sum over all "active" threads that are mapped to the same vector
  - Communication with other threads
  - Key question: How is the set of communicating threads defined?

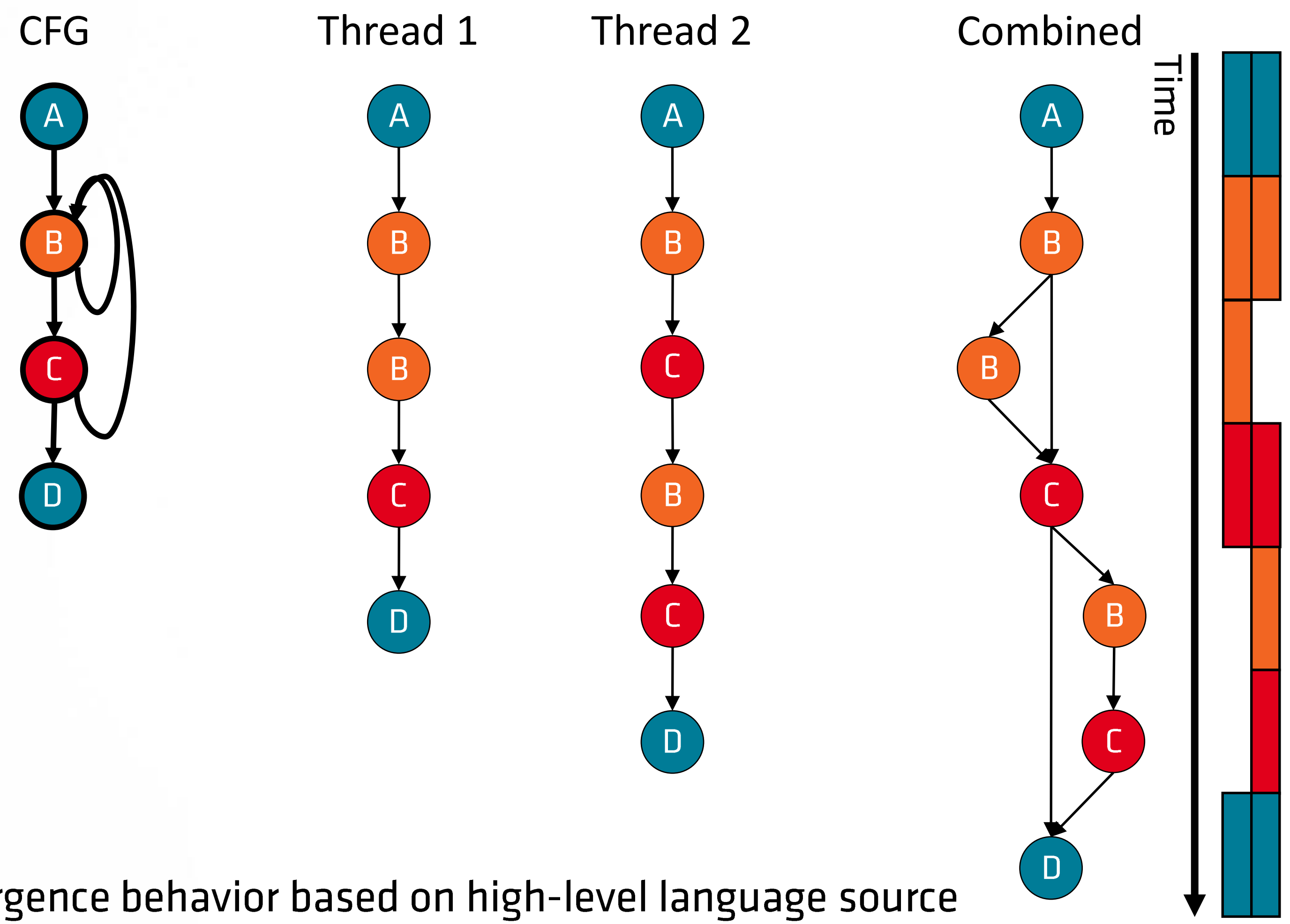
Unstructured loops allow many convergence behaviors

```
void fn_continue() {
  // (A)
  do {

    // (B)
    if (…)
      continue;
    // (C)
  } while (…);
  // (D)
}
```



Unstructured loops allow many convergence behaviors

```
void fn_loopnest() {
  // (A)
  do {
    do {
      // (B)
    } while (…);

    // (C)
  } while (…);
  // (D)
}
```

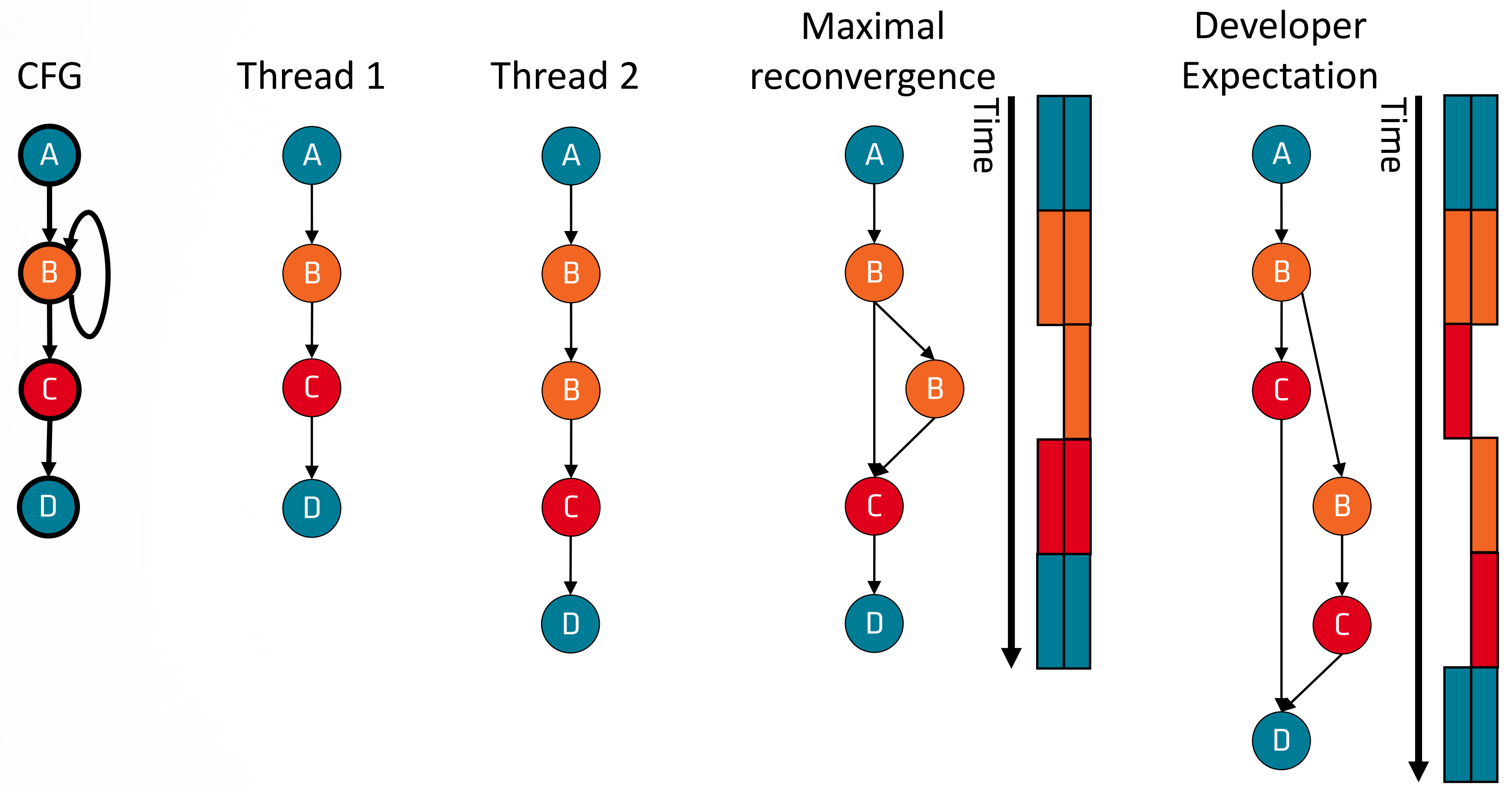


- Same CFG, different expected convergence behavior based on high-level language source
- Loss of information: CFG by itself doesn't bound convergence behavior at all

AMD

Break blocks

```
void fn_break() {
  // (A)
  for (;;) {
    // (B)
    if (…) {
      // (C)
      break;
    }
  }
  // (D)
}
```



- With convergent operations in (C), maximal reconvergence may not be desired

Composition

- Functions that internally use convergent operations may or may not “care about” the “active set of threads” with which they are called
  - For subgroupAverage, the set of communicating threads is part of the contract with the caller
  - unorderedAppend only requires that all convergent operations communicate among the same set of threads
- Want a way to express this distinction in IR

```
float subgroupAverage(float x) {
  return subgroupAdd(x) /
         subgroupAdd(1);
}
```

```
void unorderedAppend(T data) {
  uint popcount = subgroupAdd(1);
  uint base;
  if (subgroupElect())
    base = atomicAdd(bufferTail, popcount);
  uint idx = subgroupBroadcastFirst(base) +
             subgroupExclusiveAdd(1);
  buffer[idx] = data;
}
```

Convergent: a new definition

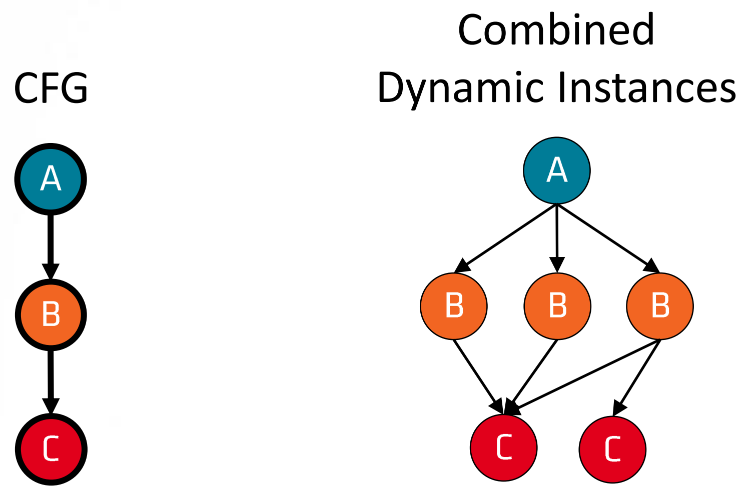
- Convergent operations communicate with other threads
- The set of communicating threads is the set of threads that executes the same dynamic instance

- Basic rules:
  - Different static instructions → different dynamic instances
  - Different executions of the same static instruction by the same thread (e.g. different loop iterations) → different dynamic instances
  - Different threads executing the same static instruction → may be the same dynamic instance

- Only the dynamic instances of convergent operations are relevant for program behavior

AMD

Spontaneous divergence and reconvergence is generally allowed



- Additional tools are required to usefully constrain dynamic instances

Convergence control bundles and intrinsics

- Intrinsics producing convergence control token values

```
token @llvm.experimental.convergence.entry() convergent readnone
token @llvm.experimental.convergence.loop() [ "convergencectrl"(token) ] convergent
                                                                         readnone
token @llvm.experimental.convergence.anchor() convergent readnone
```

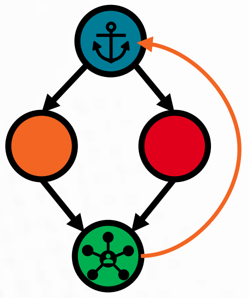
- Convergent operations are controlled

```
call void @myConvergentOperation() [ "convergencectrl"(token %tok) ]
```

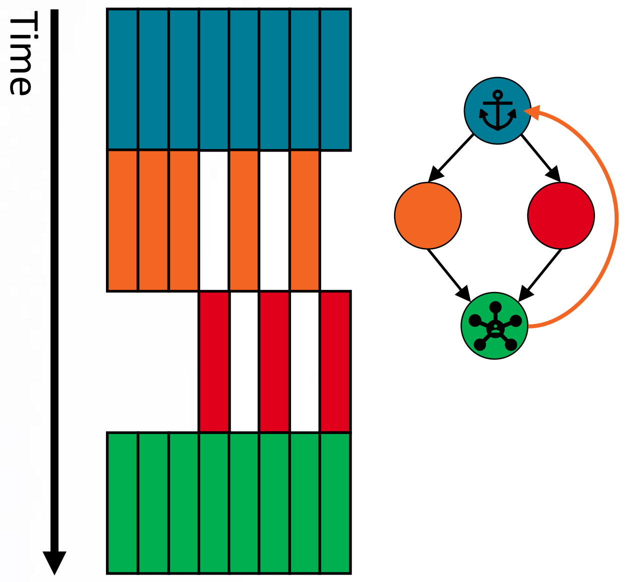
- Fundamental rule:
  - Let U be a controlled convergent operation [...] whose convergence token is produced by an instruction D
  - Two threads executing U execute the same dynamic instance of U if and only if they obtained the token value from the same dynamic instance of D

Enforcing reconvergence: the simplest case

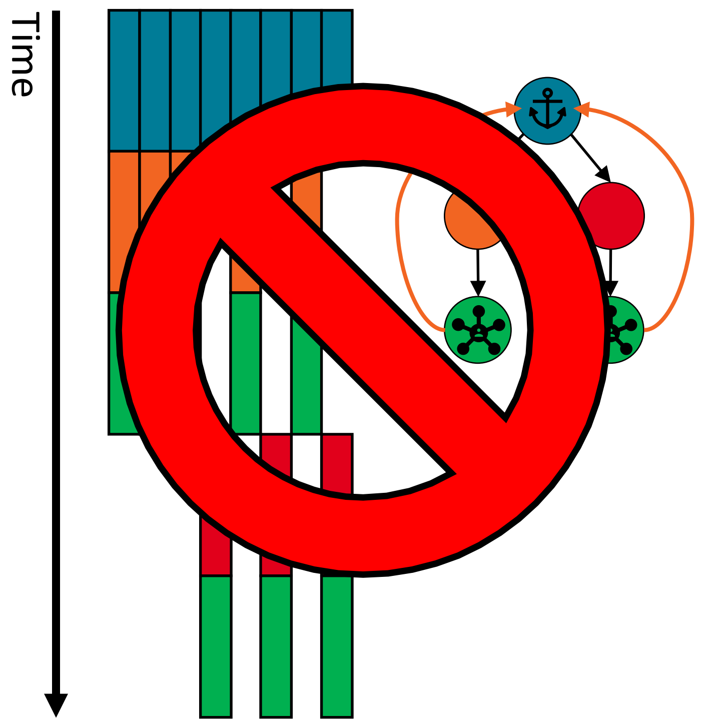
Control-flow graph



Reconvergence

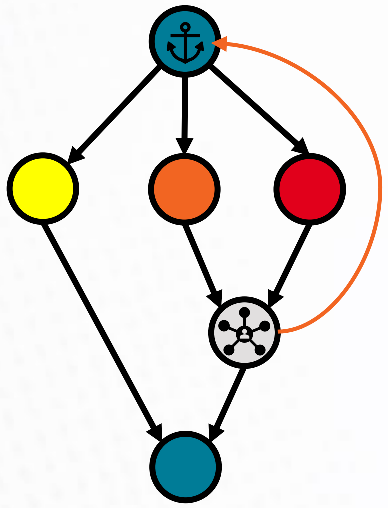


No reconvergence

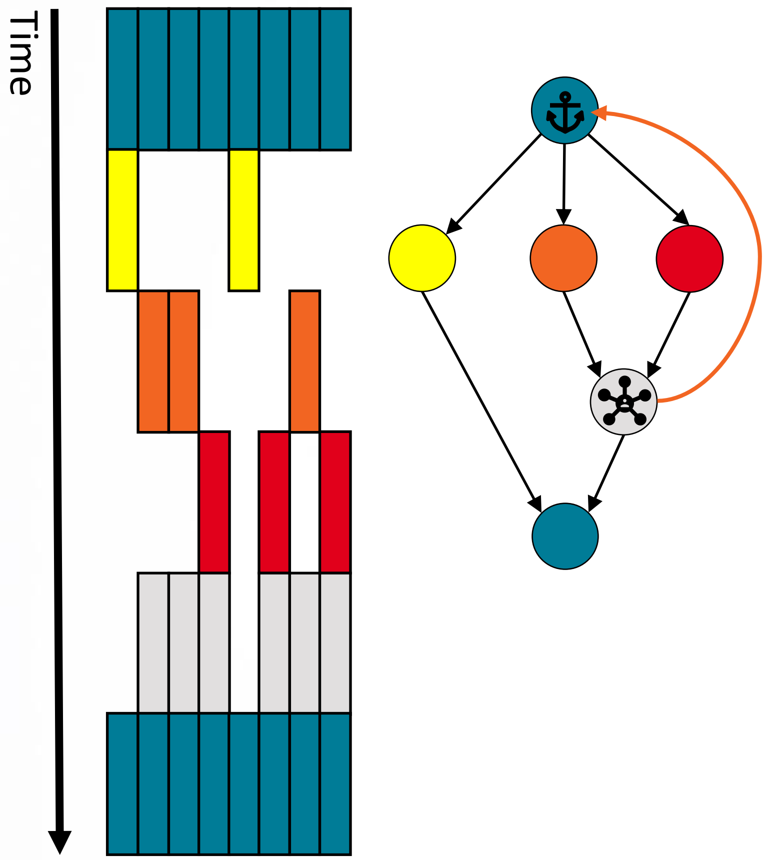


Enforcing reconvergence: a partial case

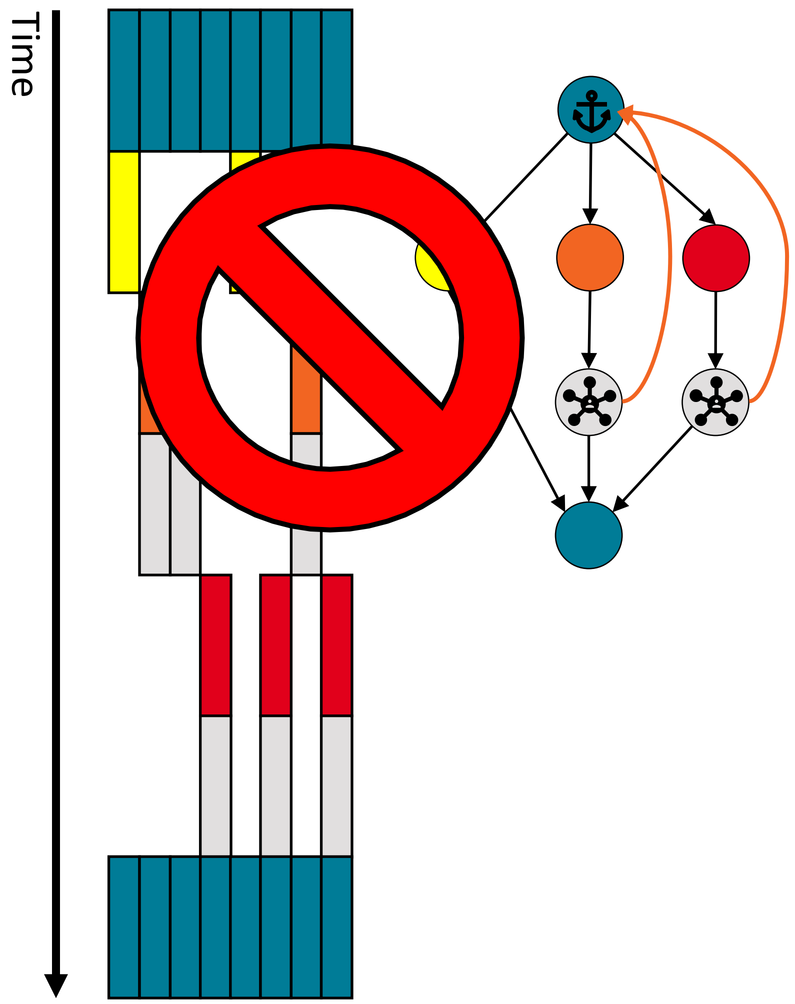
Control-flow graph



Tight reconvergence

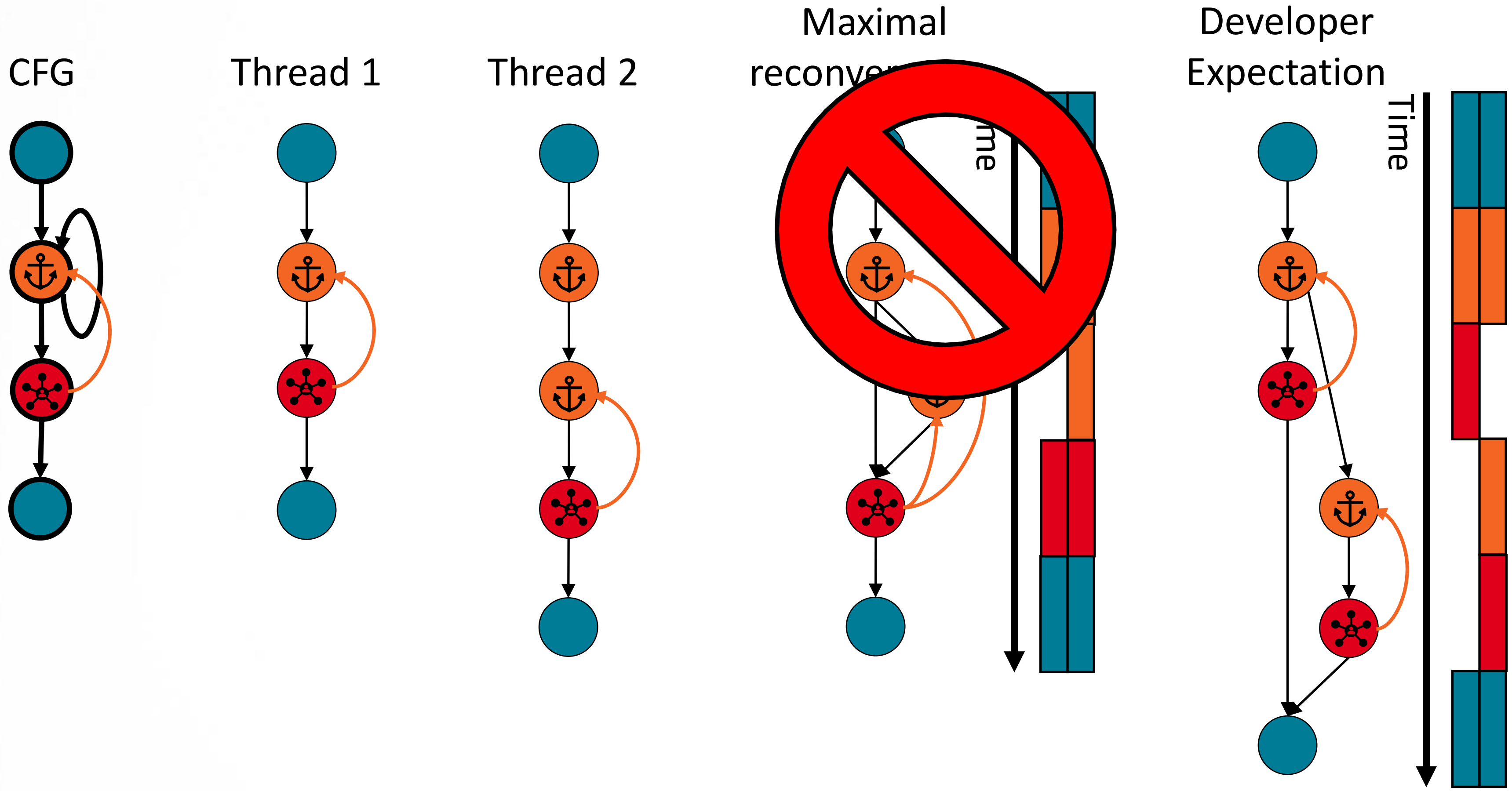


Late reconvergence



Enforcing non-reconvergence: break blocks

```
void fn_break() {
  // (A)
  for (;;) {
    // (B)
    if (…) {
      // (C)
      break;
    }
  }
  // (D)
}
```



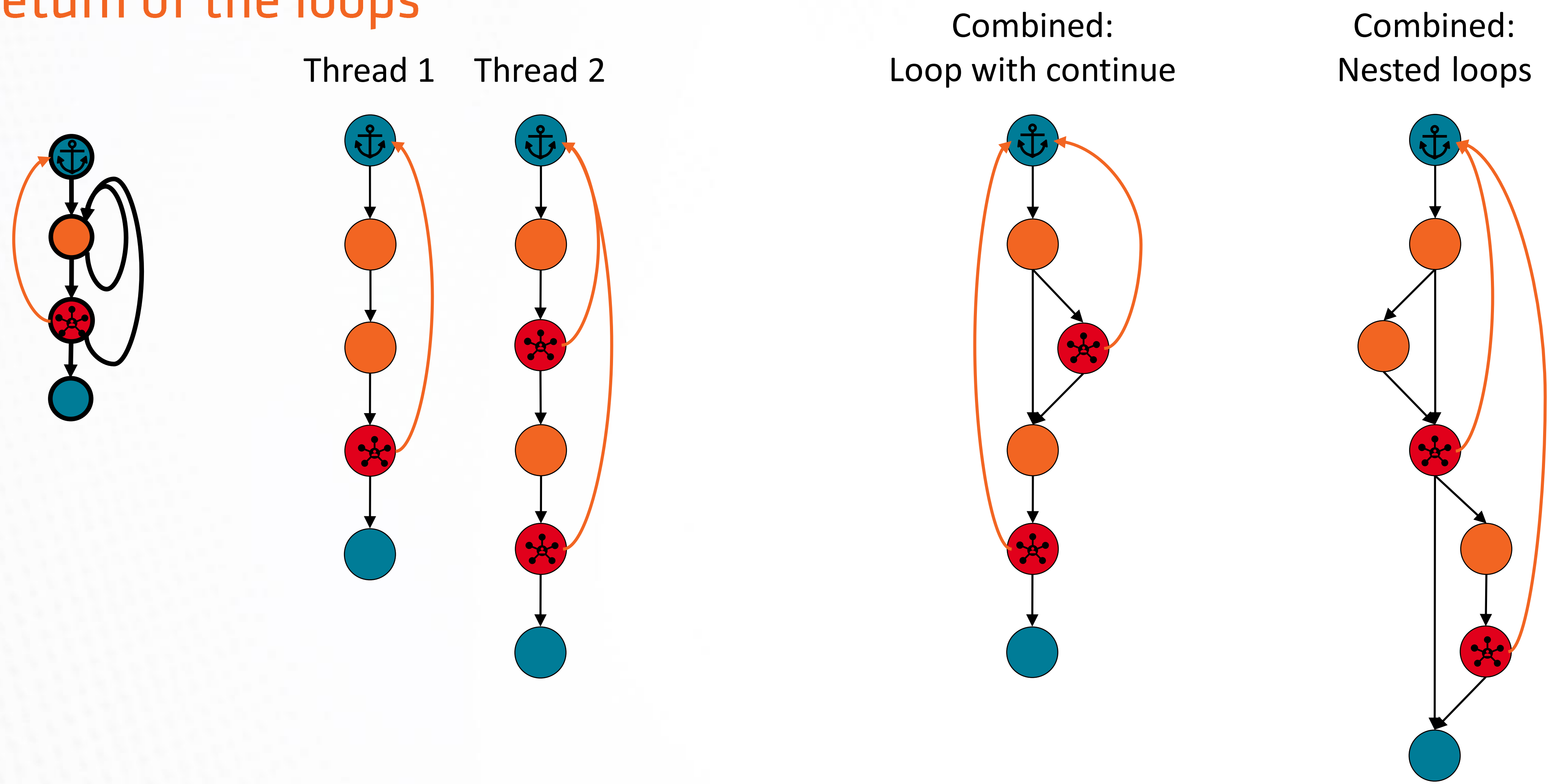
AMD

Difference between “entry” and “anchor”

```
token @llvm.experimental.convergence.entry() convergent readnone
token @llvm.experimental.convergence.anchor() convergent readnone
```

- Entry links to the set of threads in the caller
  - Dynamic instances of “entry” congruent to dynamic instances of “call” instruction
  - Can only appear in a function’s entry block
  - Use in subgroupAverage
- Anchor can appear anywhere, provides no guarantees
  - Dynamic instances are implementation-defined
  - Intention is to capture as many threads as possible while allowing maximum freedom for optimizations
  - Use in unorderedAppend

Return of the loops



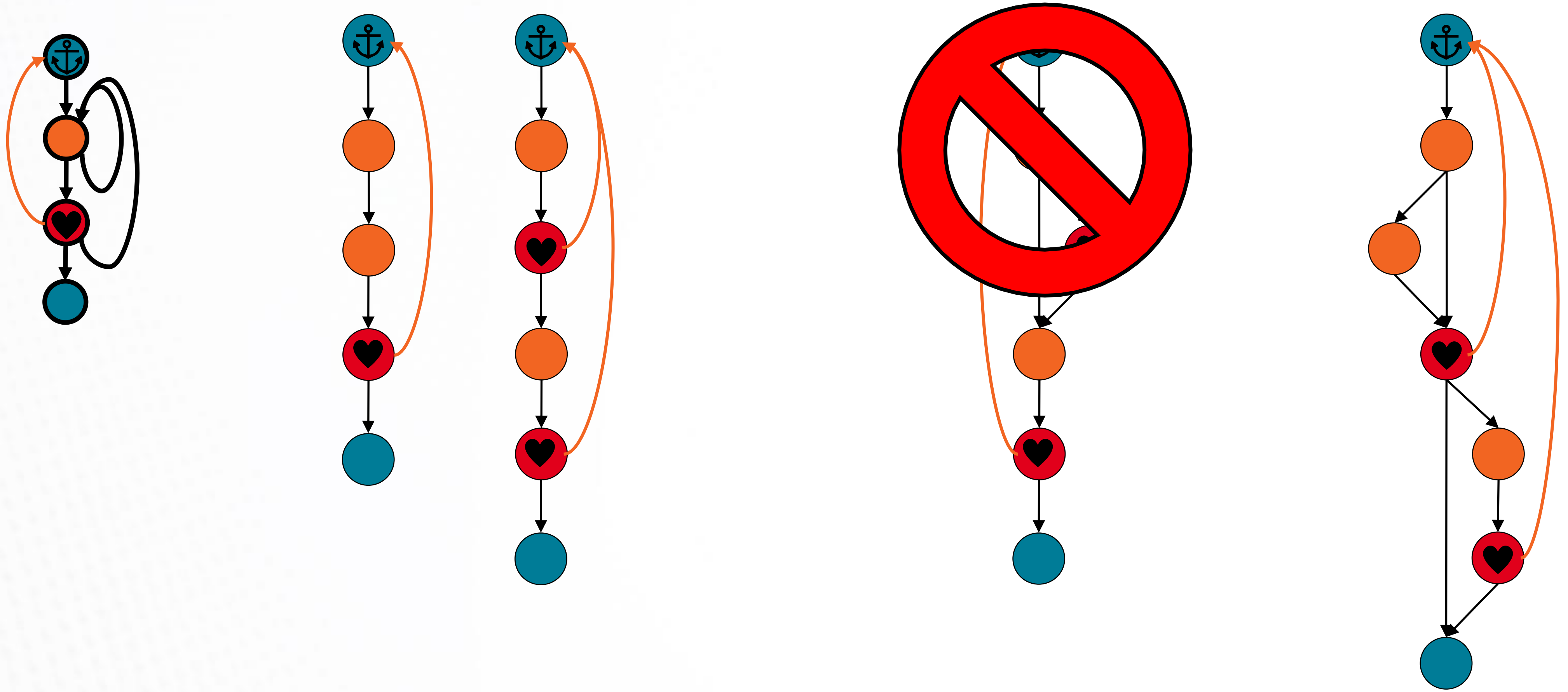
- Contradicts the fundamental rule of controlled convergent operations!
- This is defined to be invalid IR (addition to the IR verifier will flag this)

Loop hearts

Thread 1 Thread 2

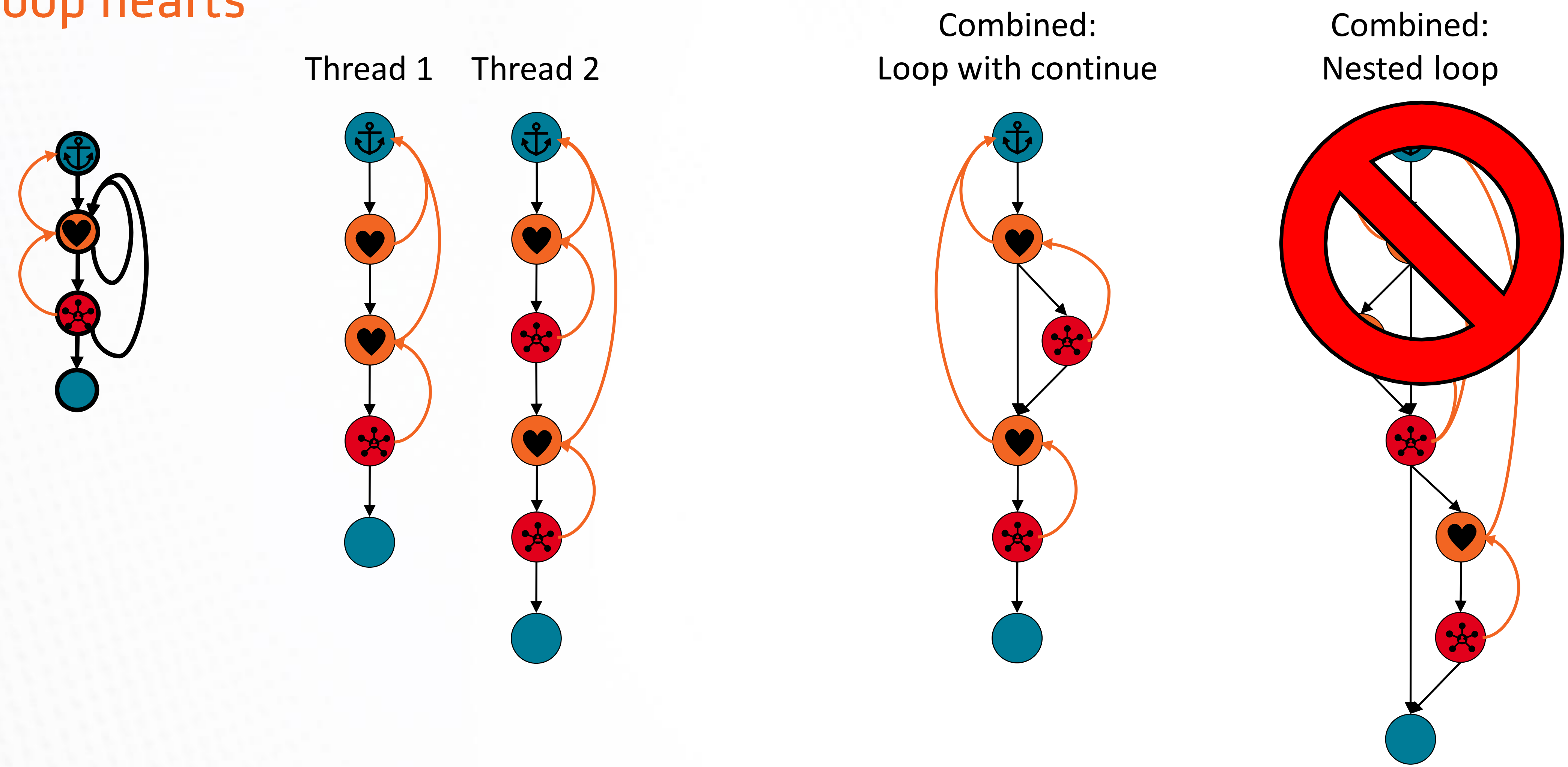
Combined: Loop with continue

Combined: Nested loop



- Loop heart rule: two threads execute the same dynamic instance of a loop heart if and only if the convergence token was produced by the same dynamic instance and both threads execute the heart the n’th time with that value (same n)

Loop hearts



- Loop heart rule: two threads execute the same dynamic instance of a loop heart if and only if the convergence token was produced by the same dynamic instance and both threads execute the heart the n’th time with that value (same n)

AMD

Impact of the new "convergent" on the compiler flow

- Frontend
  - Clang? Insert "convergencectrl" bundles and instructions for languages with convergent operations
  - Done The ConvergenceControlHeuristic pass provides best-effort insertion heuristics
- Transforms
  - Generic transforms are conservatively correct if they "don't move convergent operations across control flow"
  - Done No general theorem, but that's what experience suggests so far
  - No known cases of spooky action at a distance
- Backend
  - WIP Ensure convergence as required by convergence control intrinsics
- Uniform / Divergence analysis
  - Bug fixing Uniformity of values can be affected by convergence control intrinsics
  - A value V is uniform at a program point P if an appropriately controlled convergent operation in P sees the same value of V in all communicating threads
  - To do Want an API where users of divergence analysis can query the correct convergence control intrinsics / token to be inserted

AMD

The end

- History of “convergent”
- Cross-lane operations and examples
- Composition
- Convergence control intrinsics and rules for dynamic instances

https://reviews.llvm.org/D85603

```
token @llvm.experimental.convergence.entry() convergent readnone
token @llvm.experimental.convergence.loop() [ "convergencectrl"(token) ] convergent
                                                                         readnone
token @llvm.experimental.convergence.anchor() convergent readnone
```

Thank you!

Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

AMD