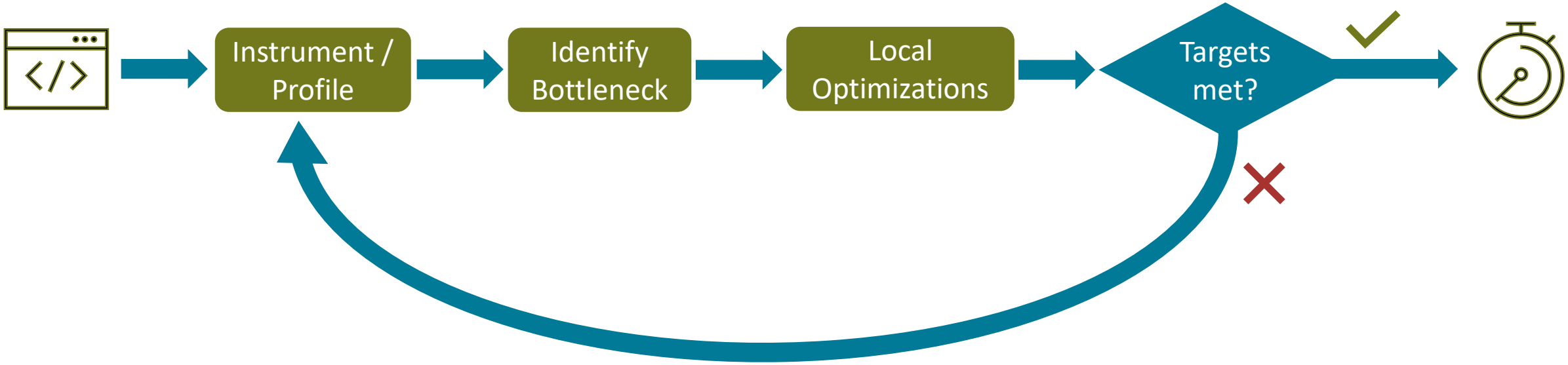
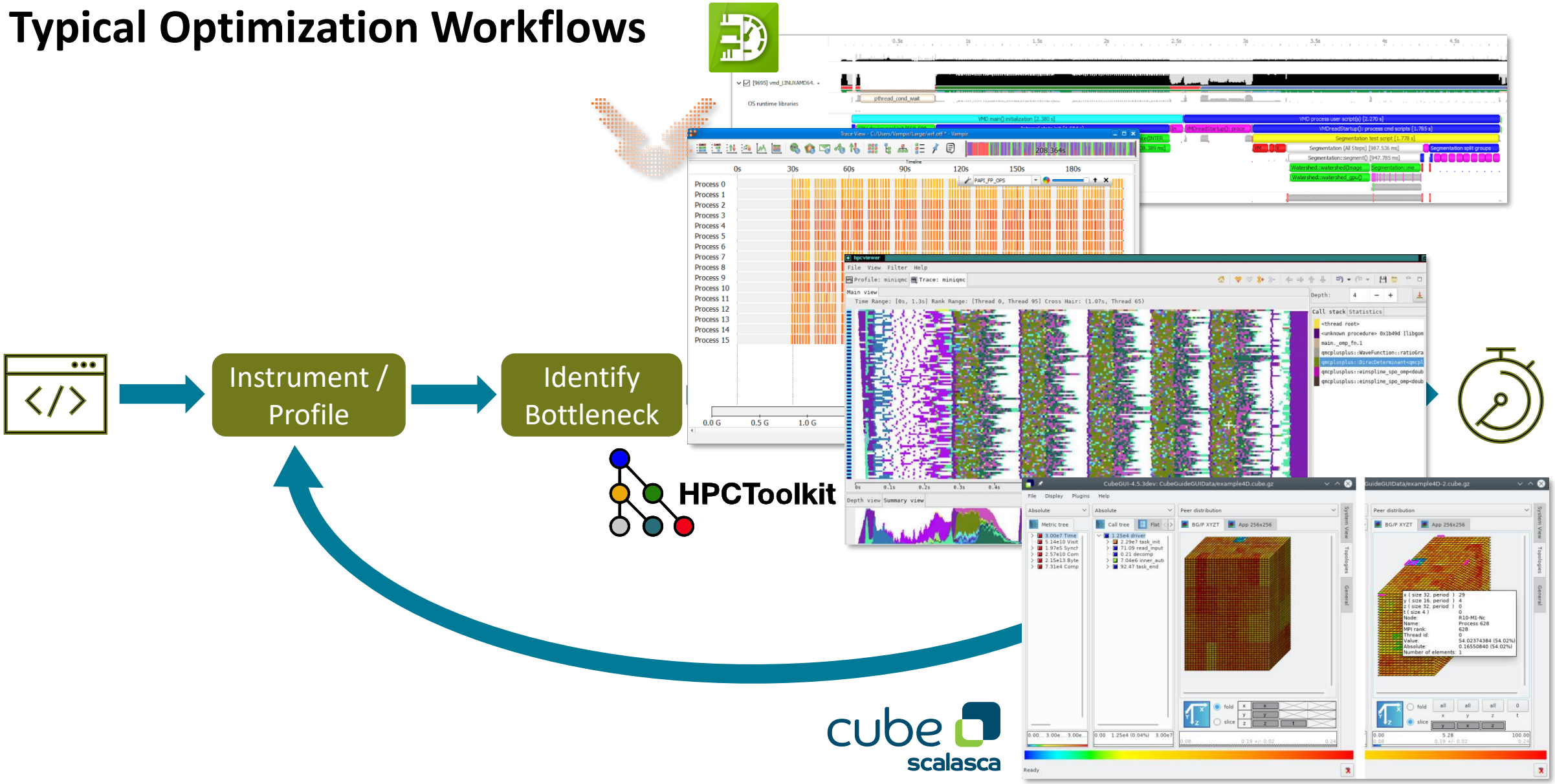
PHILIPP SCHAAD, TAL BEN-NUN, TORSTEN HOEFLER

C.A.T.S.: Memory and Control Flow Tracing for Whole-Program Performance Analysis

Typical Optimization Workflows



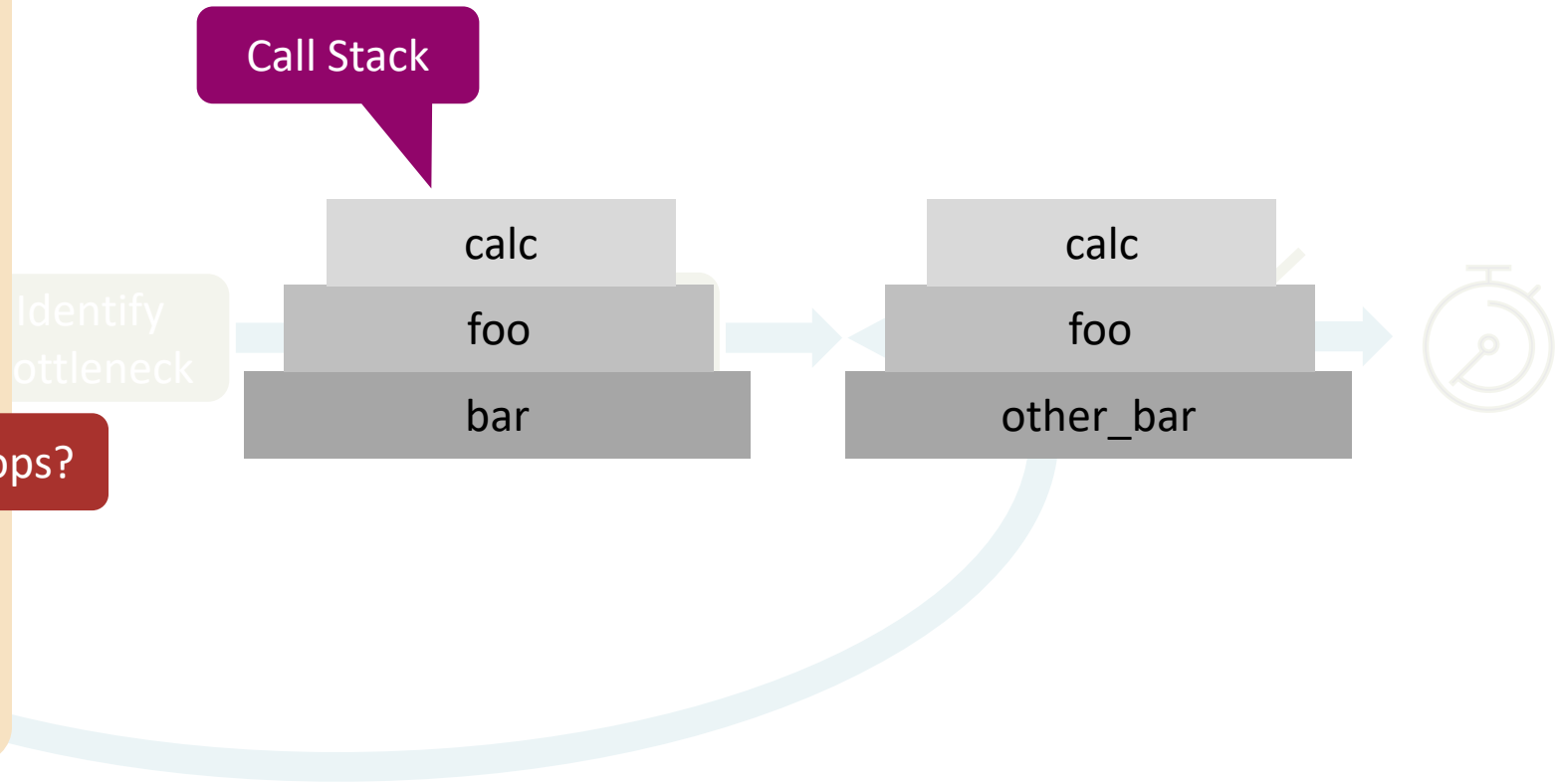
Typical Optimization Workflows



Typical Optimization Workflows

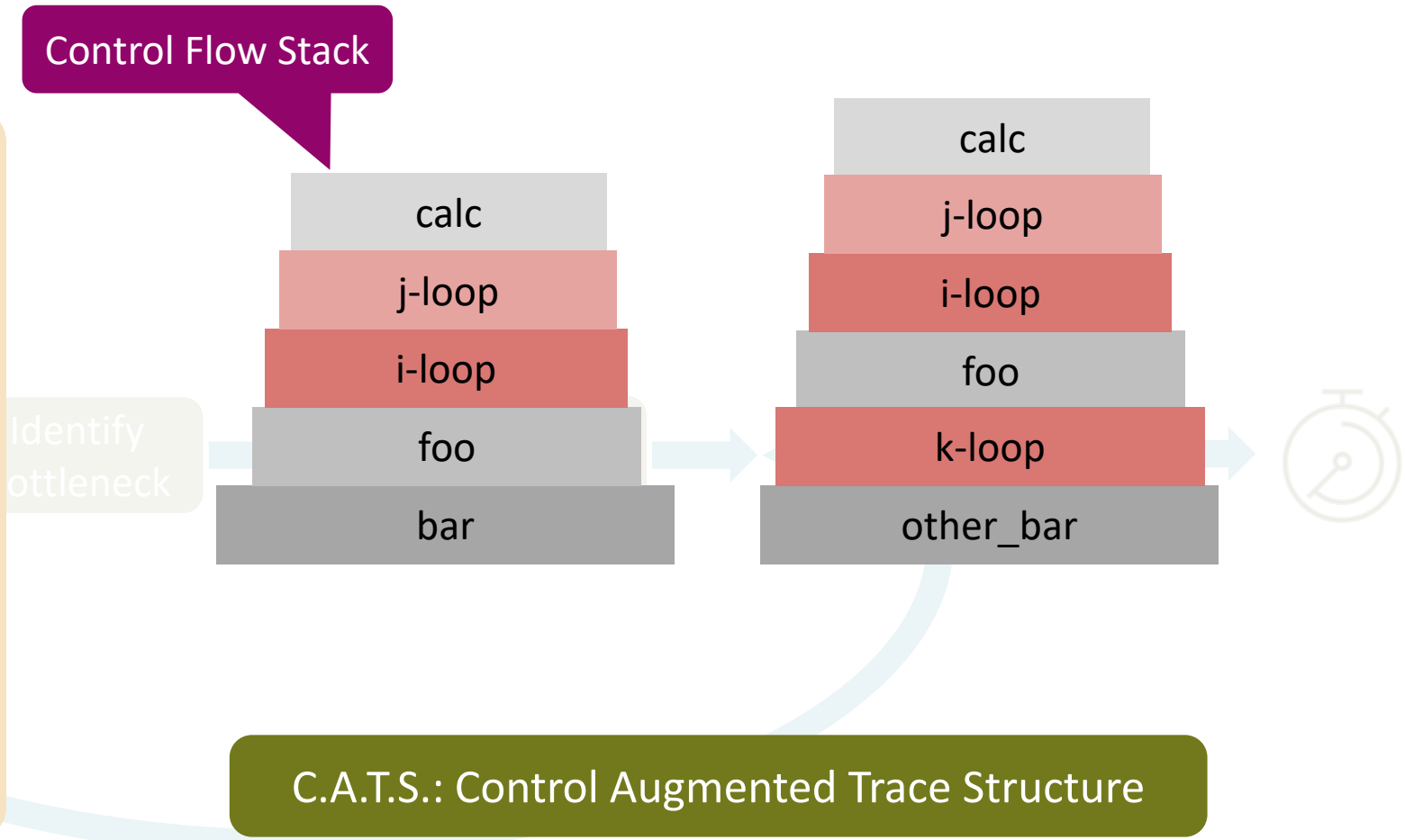
```
void foo() {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      A[i][j] = calc(i, j);
    }
  }
}

void bar() {
  foo();
}

void other_bar() {
  for (int k = 0; k < K; k++) {
    foo();
  }
}
```

But what about loops?

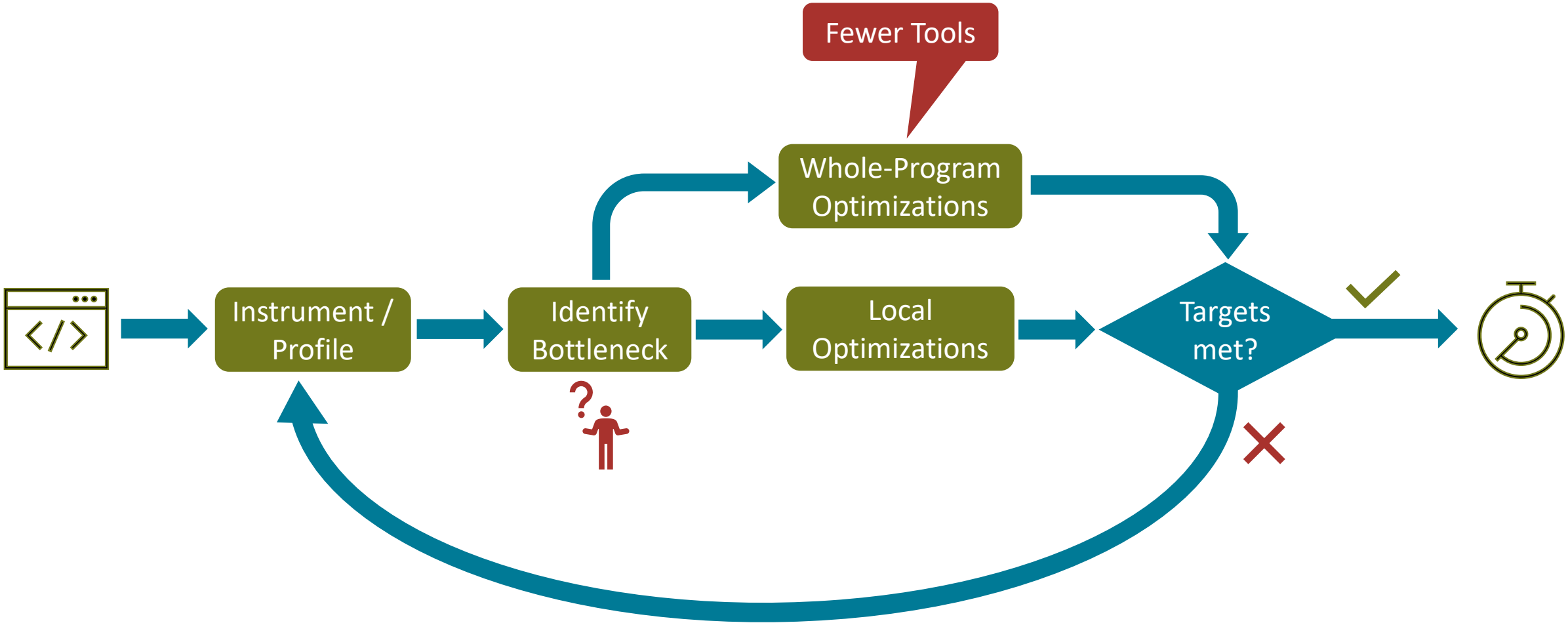


Typical Optimization Workflows

```
void foo() {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      A[i][j] = calc(i, j);
    }
  }
}

void bar() {
  foo();
}

void other_bar() {
  for (int k = 0; k < K; k++) {
    foo();
  }
}
```



Typical Optimization Workflows



Typical Optimization Workflows

3 Whole-Program Performance Visualizations

Fewer Tools

- Improve Data Movement
- Increase Coarse Parallelism
- Reduce Memory Footprint
- Mitigate Anti-Patterns

Whole-Program Optimizations

Instrument / Profile → Identify Bottleneck → Local Optimizations → Targets met?

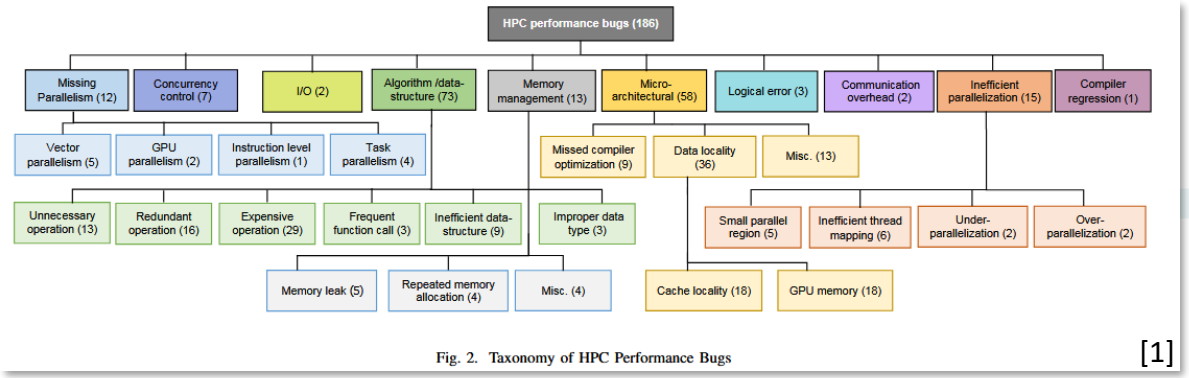


Fig. 2. Taxonomy of HPC Performance Bugs

[1]

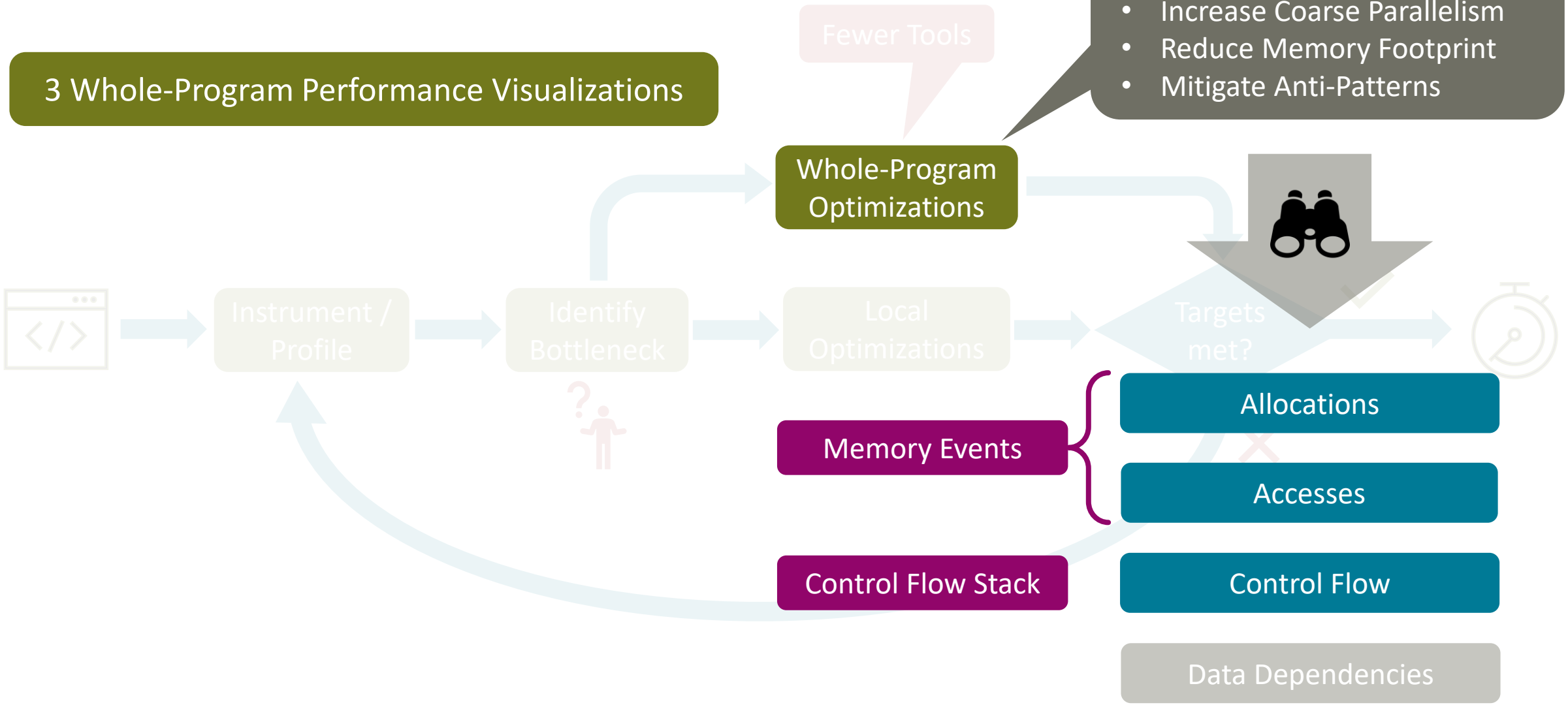
- Allocations
- Accesses
- Control Flow
- Data Dependencies

[1] Azad et al., An Empirical Study of High Performance Computing (HPC) Performance Bugs

Typical Optimization Workflows

3 Whole-Program Performance Visualizations

- Improve Data Movement
- Increase Coarse Parallelism
- Reduce Memory Footprint
- Mitigate Anti-Patterns



C.A.T.S.: Control Augmented Trace Structure

Allocation
- Identifier
- #Bytes
- Memory Type

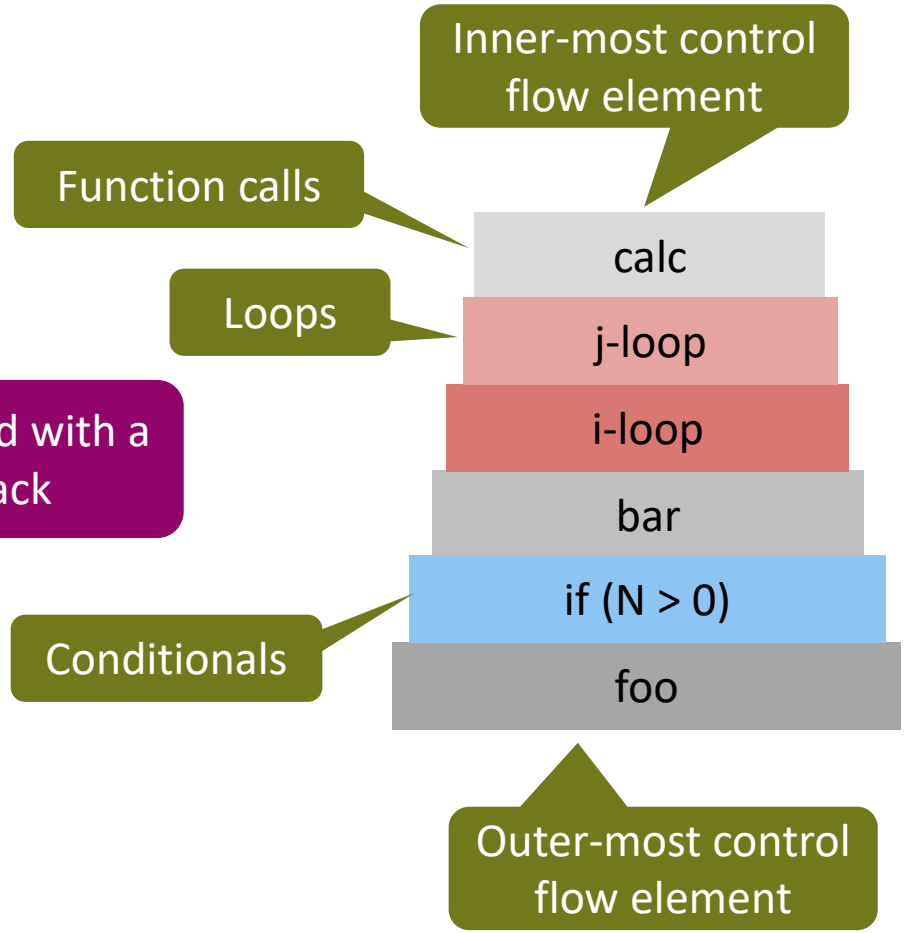
my_arr

Deallocation
- Identifier

Access
- Read / Write
- Identifier
- Byte Offset
- #Bytes

my_arr[5]

Each event associated with a Control Flow Stack



C.A.T.S.: Trace Construction

Runtime Tracing

- Observes Reality
- Data-Dependent Execution
- High Capture Time
- Memory Overhead

Static Tracing

- Low Capture Time
- Captures Symbolic Information
- Infeasible for Data-Dependent Execution

All LLVM compatible source languages

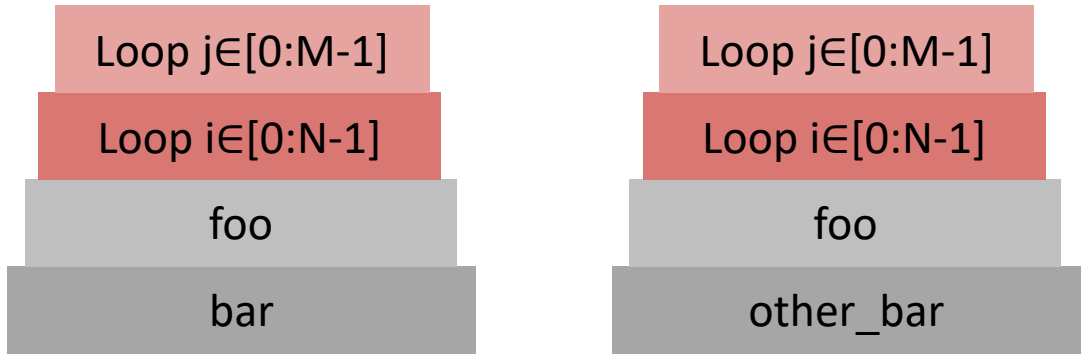
C.A.T.S.: Trace Compression

```
void foo() {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      A[i][j] = calc(i, j);
    }
  }
}

void bar() {
  foo();
}

void other_bar() {
  foo();
}
```

Control flow stack captures fact that this is in a loop nest

Only really care about the event once per loop nest

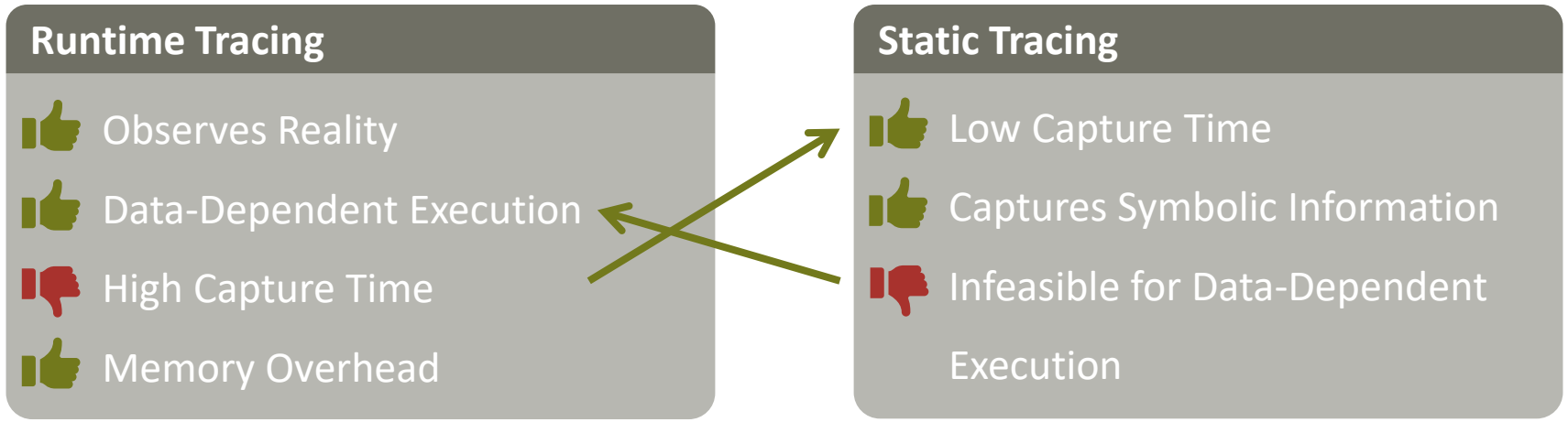
Instrumenting this access causes 2 * N * M events



→ Events only recorded once per ***unique*** control flow stack

Only 2 unique stacks → 2 events

C.A.T.S.: Implementations



C.A.T.S.: Static Tracing

Tracing via symbolic execution gives symbolic information

Allocation

- Identifier
- #Bytes
- Memory Type

Use symbolic expression (e.g., N*M)

Access

- Read / Write
- Identifier
- Byte Offset
- #Bytes

`i*80+j`

Establish relationship between control flow and events

C.A.T.S.: Novel Peformance Visualizations

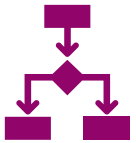
Reduce Data Movement

Increase Coarse Parallelism

Reduce Memory Footprint

Mitigate Anti-Patterns

Control Flow Data Dependency View



Memory Timeline View

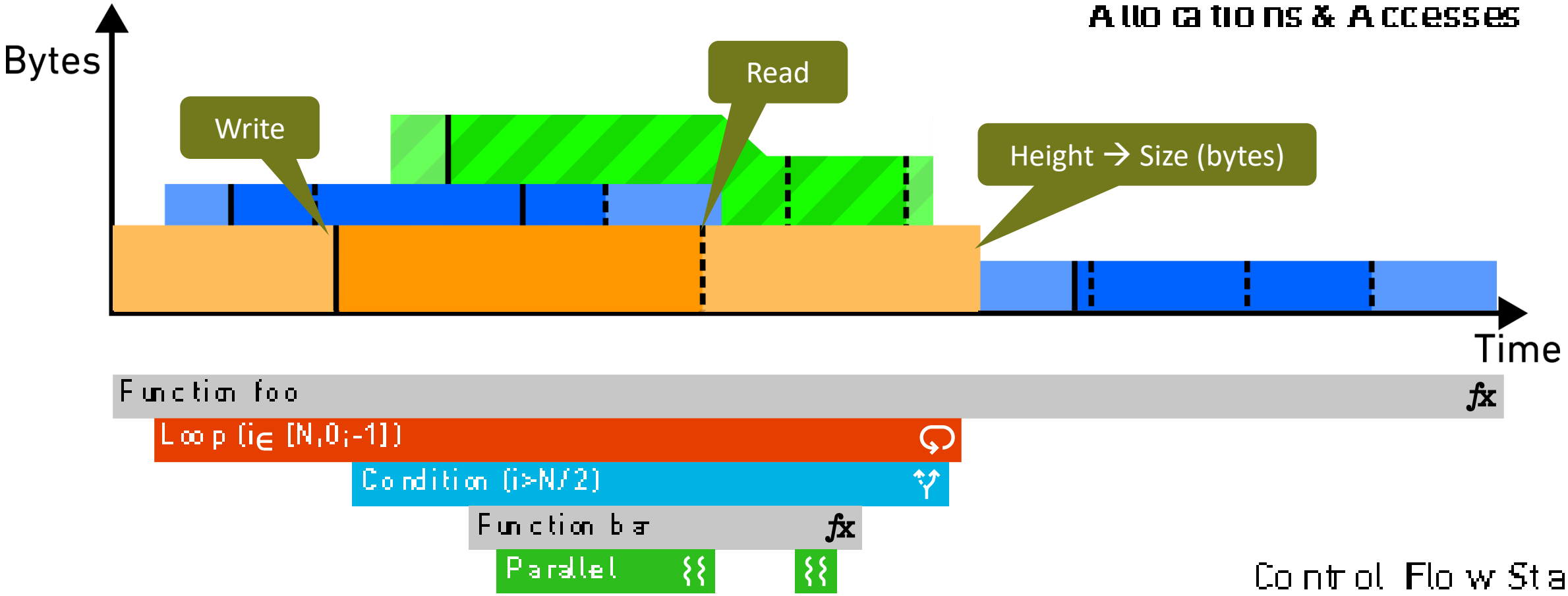
Data Access Statistics View

C.A.T.S.: Novel Peformance Visualizations

Reduce Data Movement | Increase Coarse Parallelism | Reduce Memory Footprint | Mitigate Anti-Patterns

Control Flow Data Dependency View | **Memory Timeline View** | Data Access Statistics View

Memory Timeline View

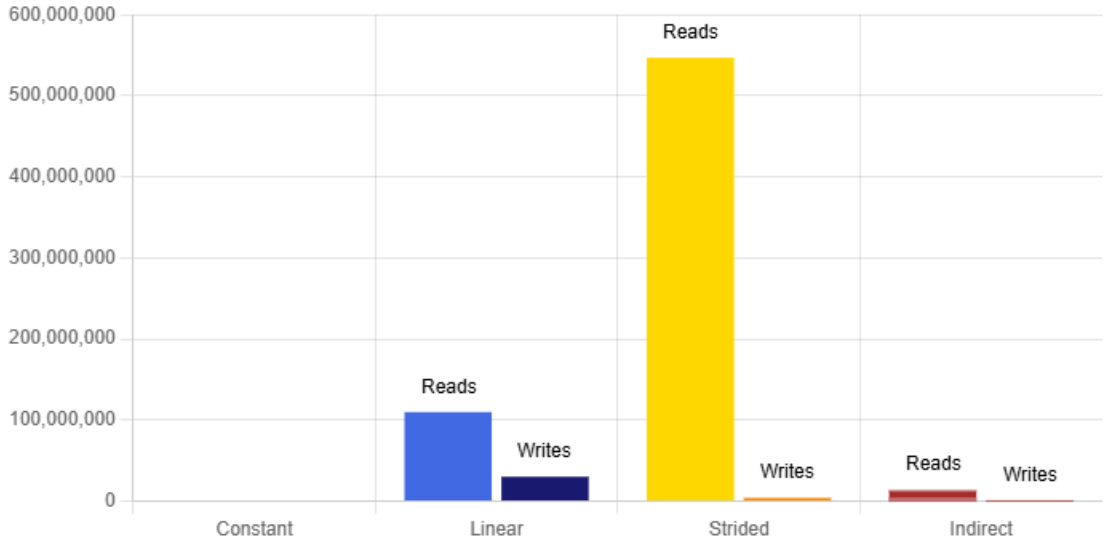


Access Statistics View

Histogram of data accesses by type

Choose File full_step_07_...ss_stats.json | Sort by Indirect Accesses

Sorting order Descending | Group by Data Container



domain.zd (737373)

lulesh_dace_lagrange_nodal_y1 (1512000)

lulesh_dace_lagrange_nodal_z1 (1512000)

lulesh_dace_lagrange_nodal_x1 (1512000)

domain.z (923582)

Breakdown based on data container / control flow element

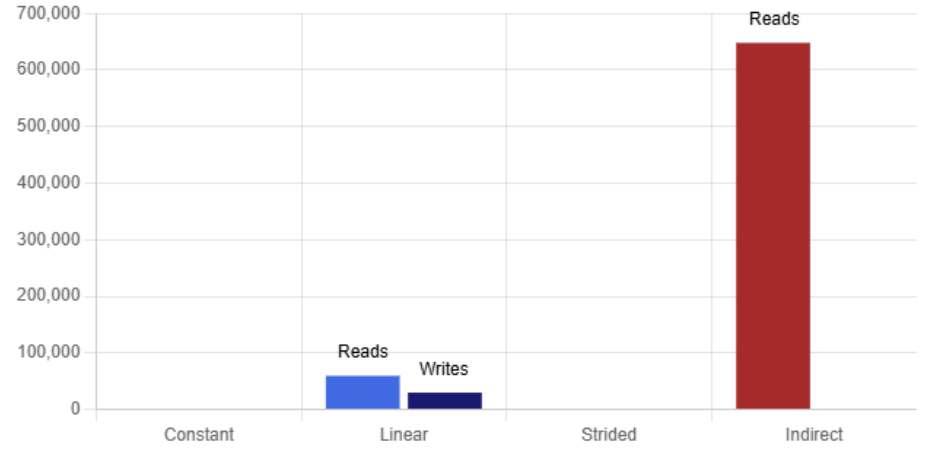
domain.x (923582)

domain.y (923582)

domain.yd (737373)

Sub-historgram per category

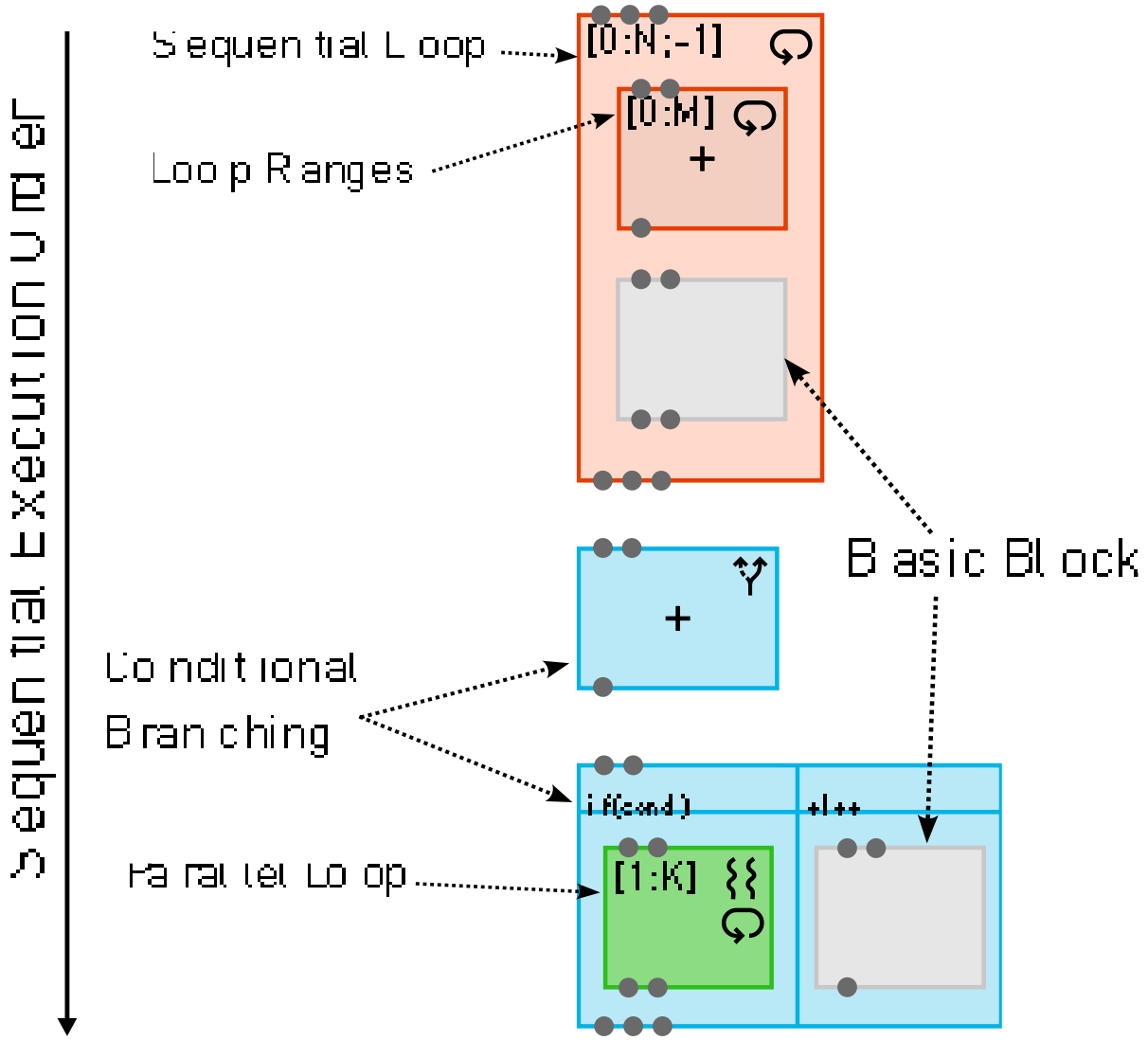
domain.xd (737373)



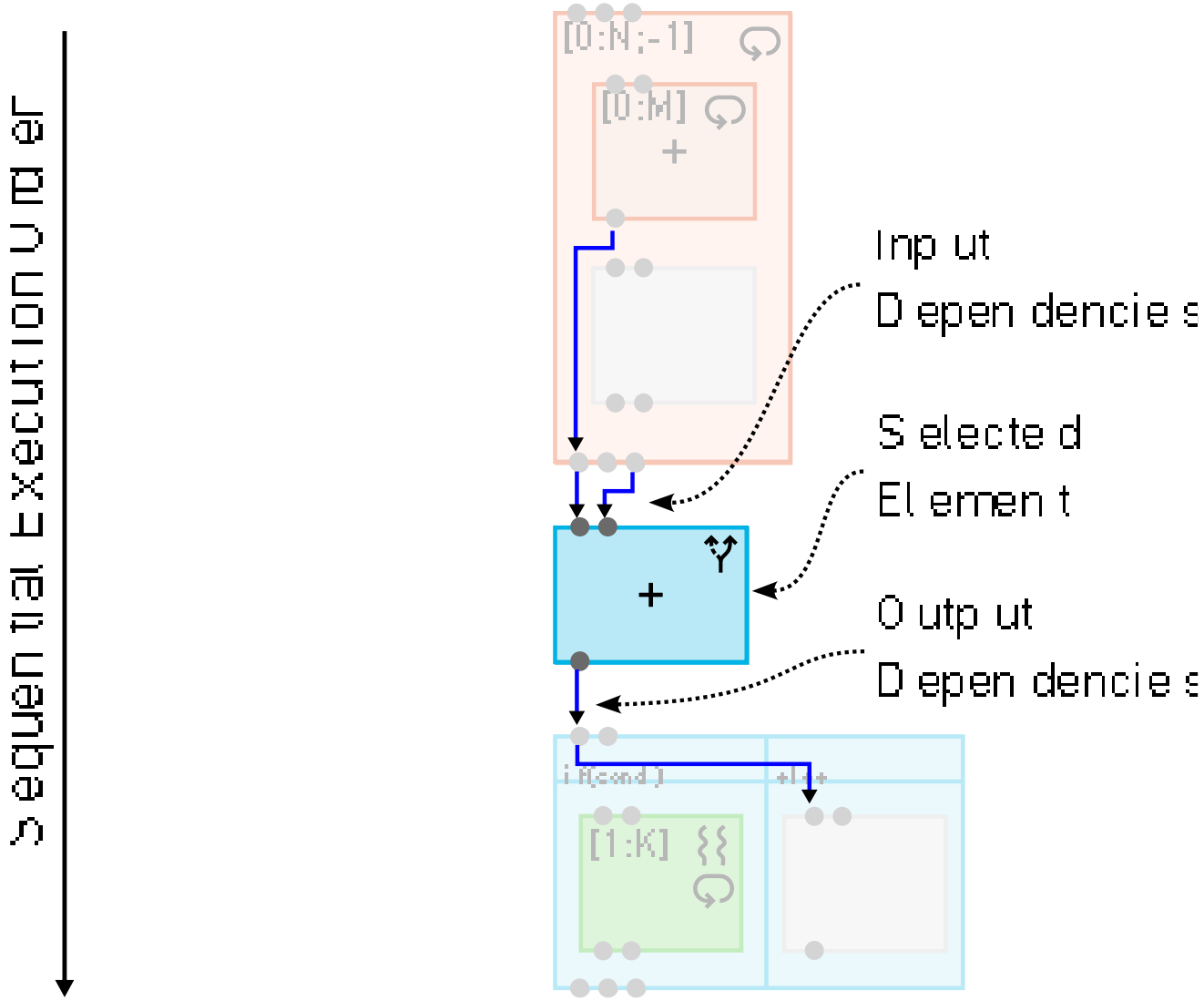
domain.delv_xi (54000)

__tmp18.ss (27000)

Control Flow Data Dependency View



Control Flow Data Dependency View



Shock Hydrodynamics – LULESH

Valgrind: 41 minutes / 11.8 MB trace

→ Static Tracing →

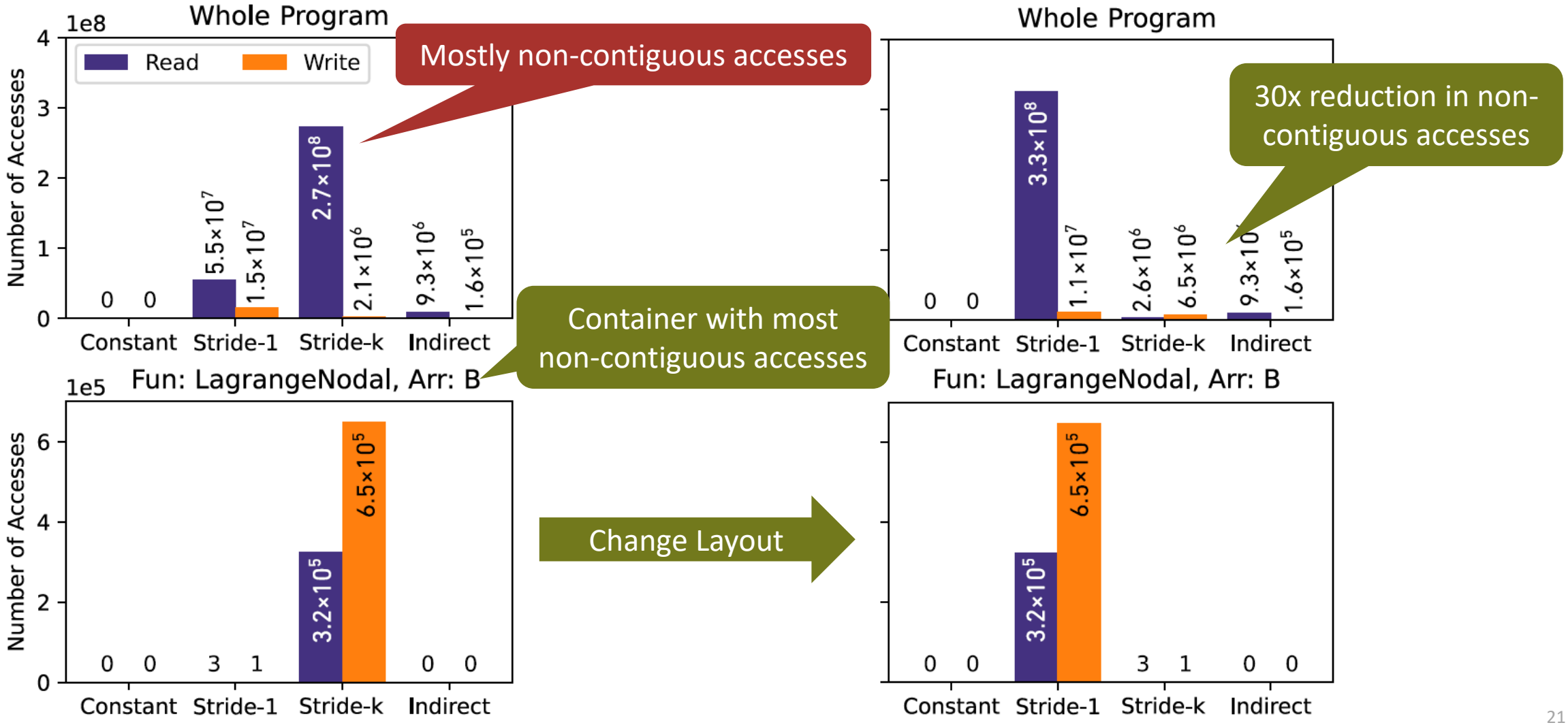
24 seconds / 395 KB trace

30x smaller trace

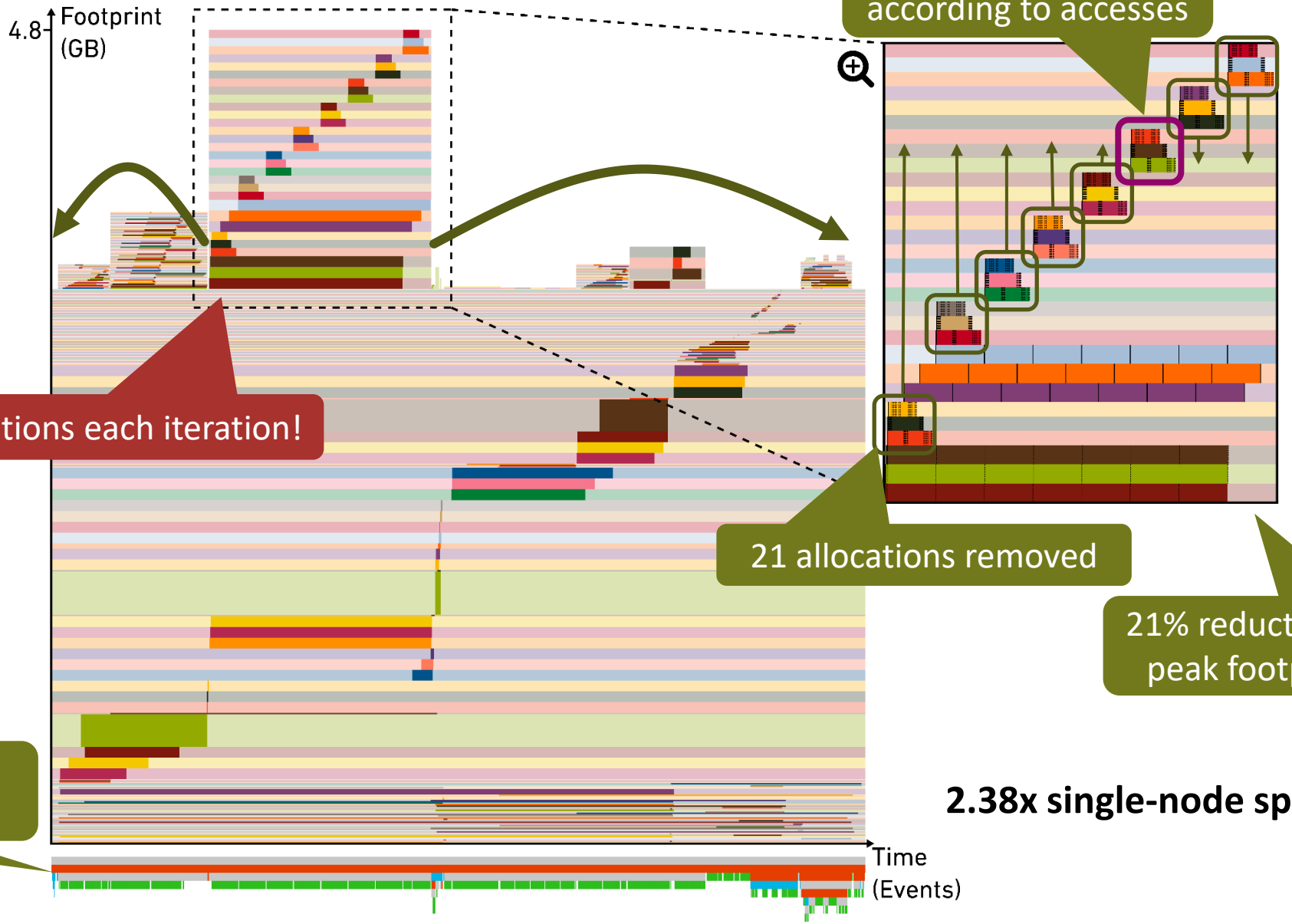
Shock Hydrodynamics – LULESH

Static Tracing → 24 seconds / 395 KB trace

1.18x single-node speedup



Shock Hydrodynamics – LULESH



Reuse is possible according to accesses

Large allocations each iteration!

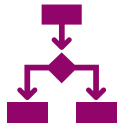
21 allocations removed

21% reduction of peak footprint

Program dominated by one large time-step loop

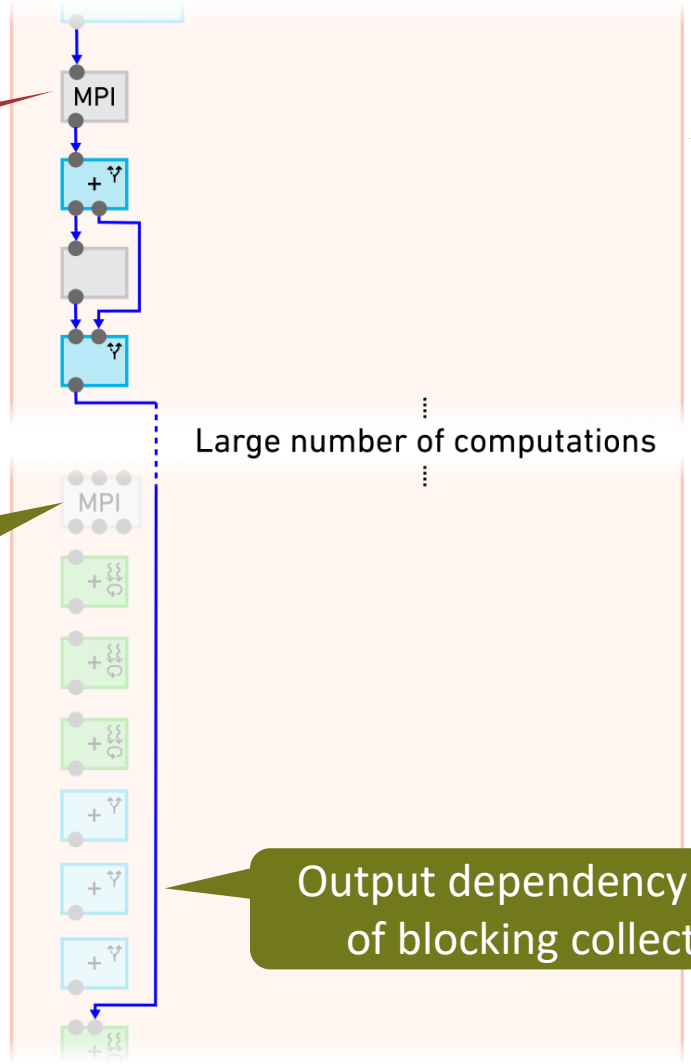
2.38x single-node speedup

Shock Hydrodynamics – LULESH



Blocking **MPI_Allreduce**

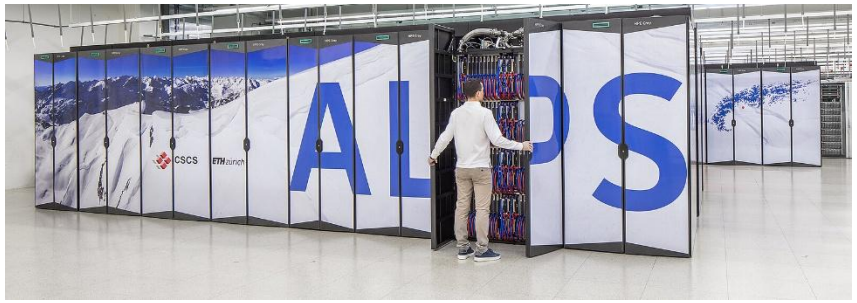
Use non-blocking version!



Time-step loop

Next blocking MPI synchronization point

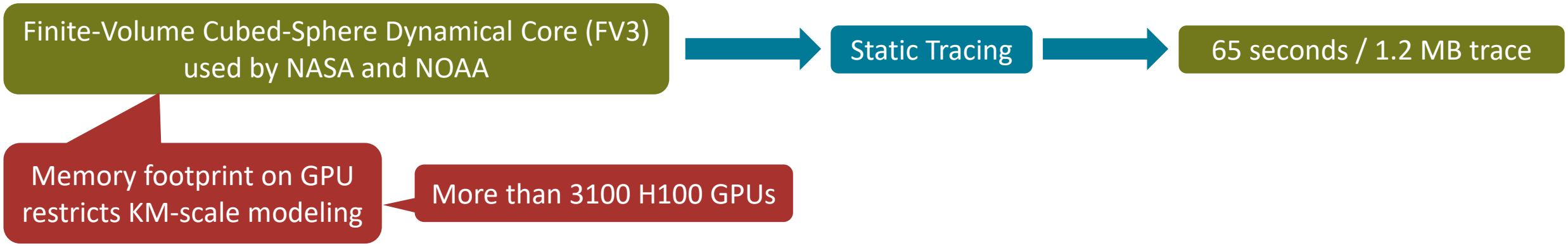
Output dependency chain of blocking collective



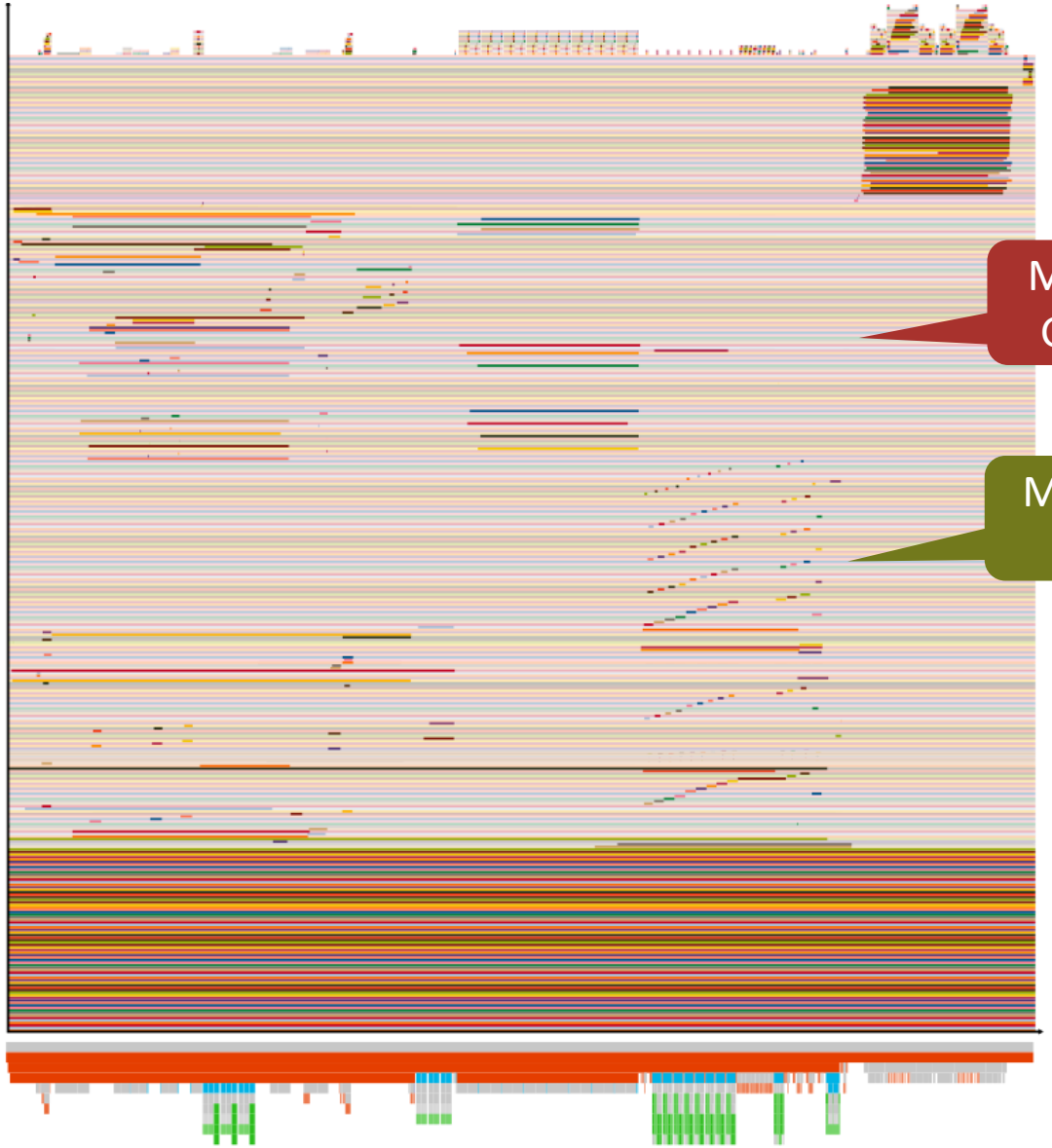
256 nodes with 4 GH200 each, 2197 ranks

5% speedup / 10% communication overhead reduction

Kilometer-Scale Weather Modeling



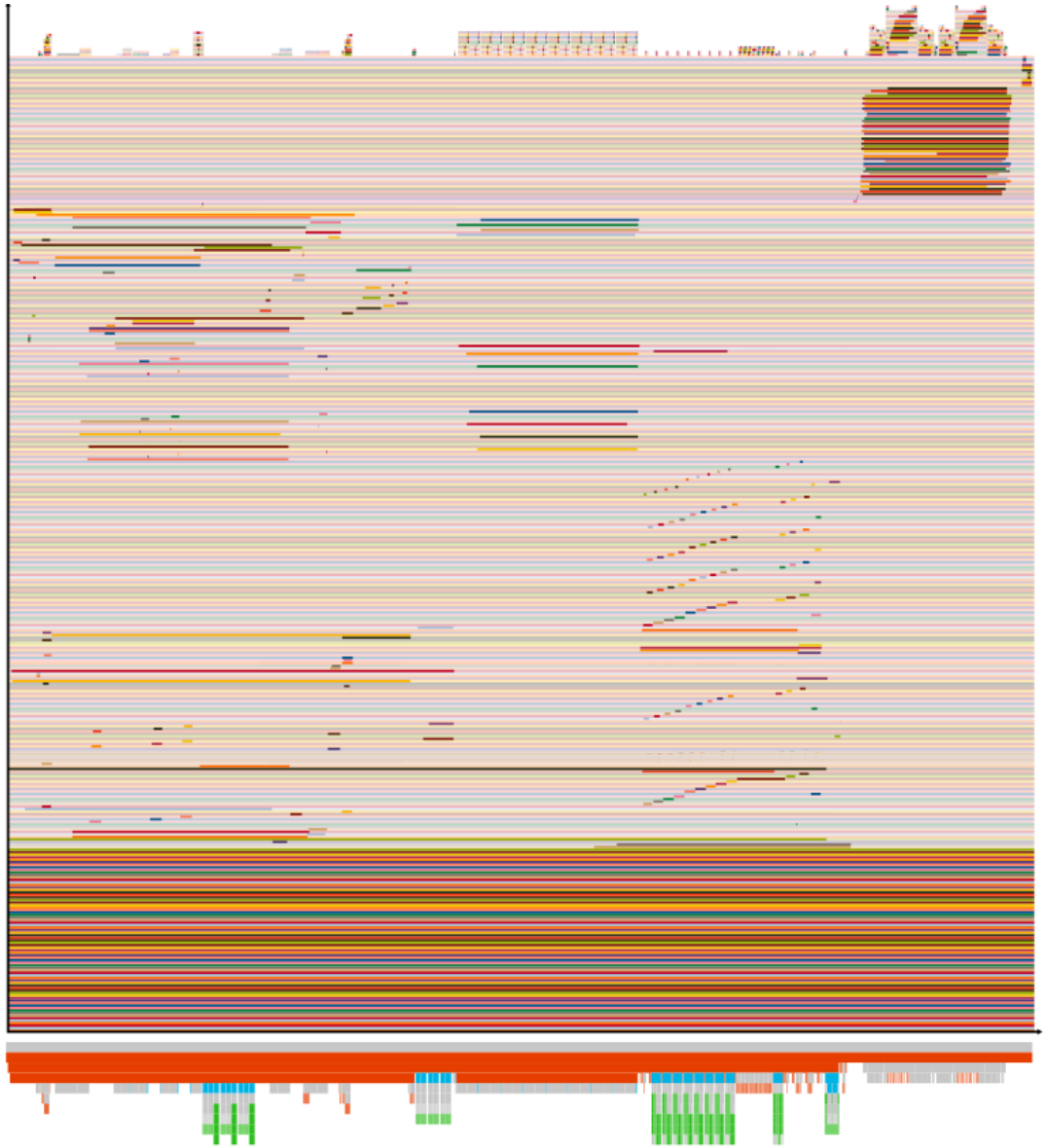
Kilometer-Scale Weather Modeling

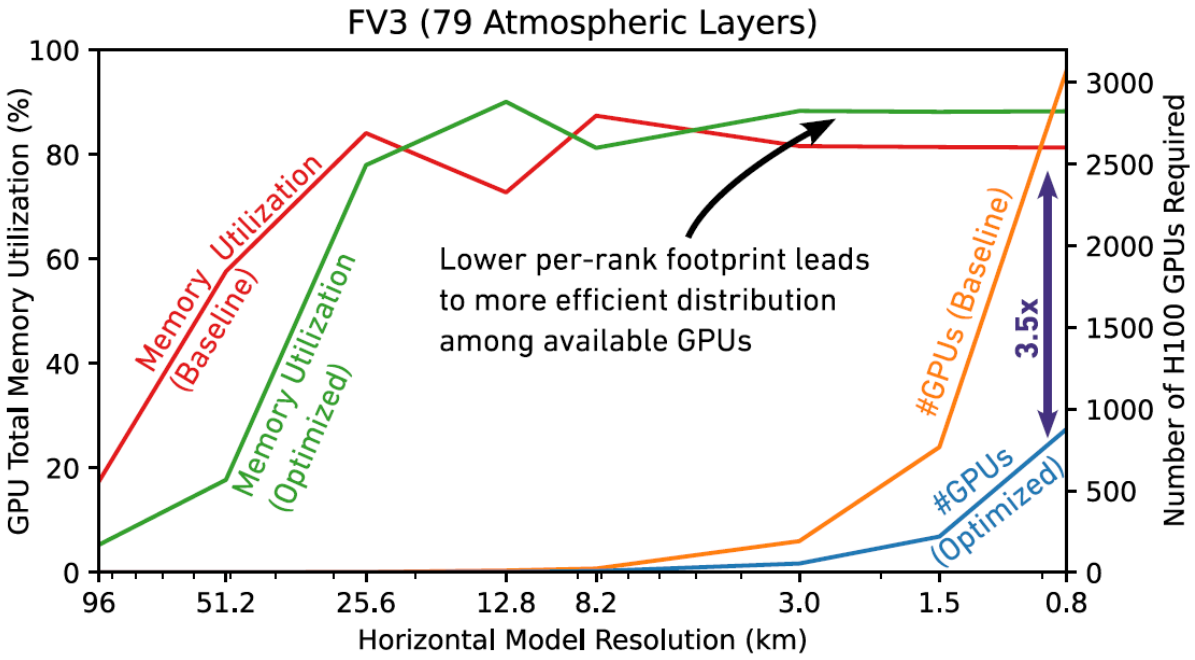


Mostly unused GPU memory

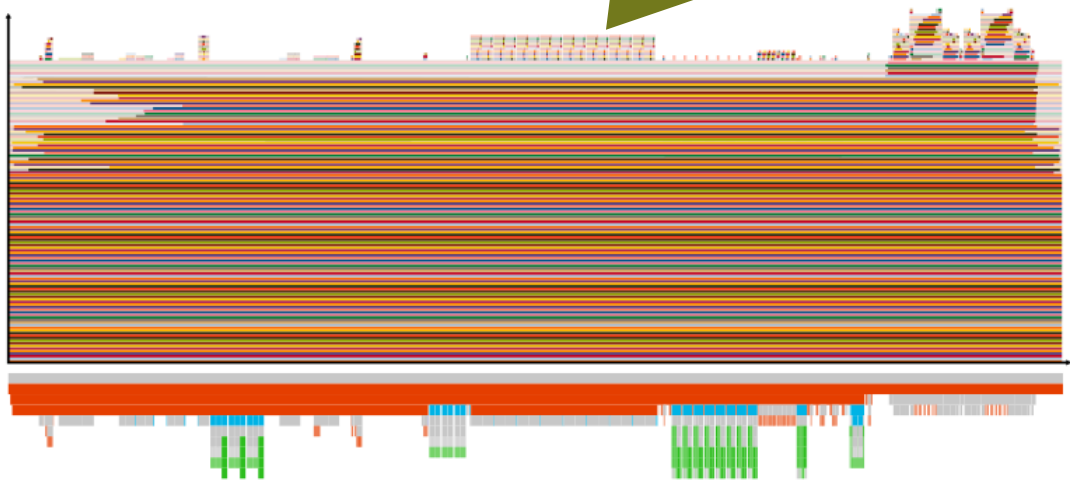
Most temporaries can be reused!

Kilometer-Scale Weather Modeling

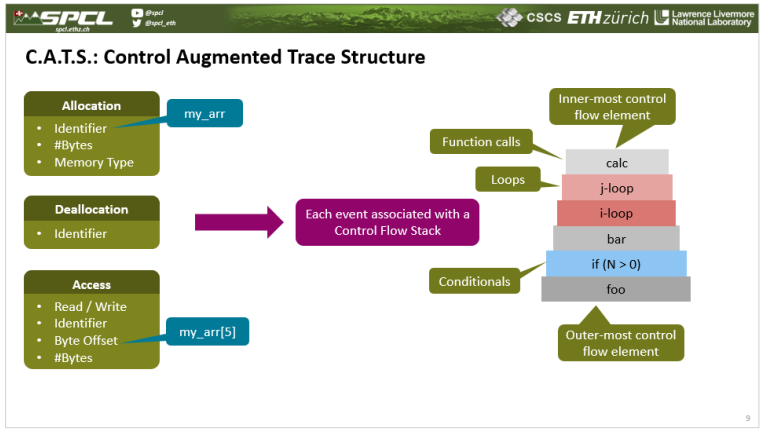


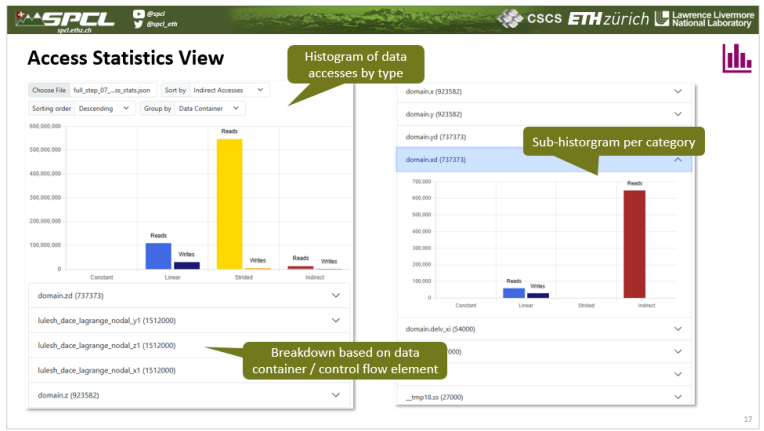


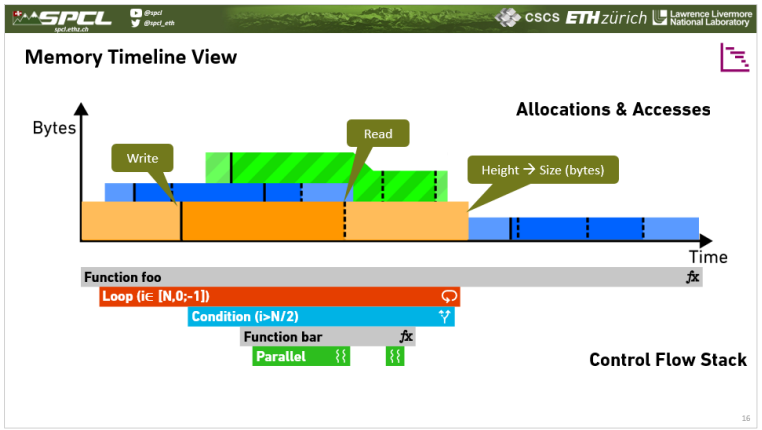
More than 3x footprint reduction

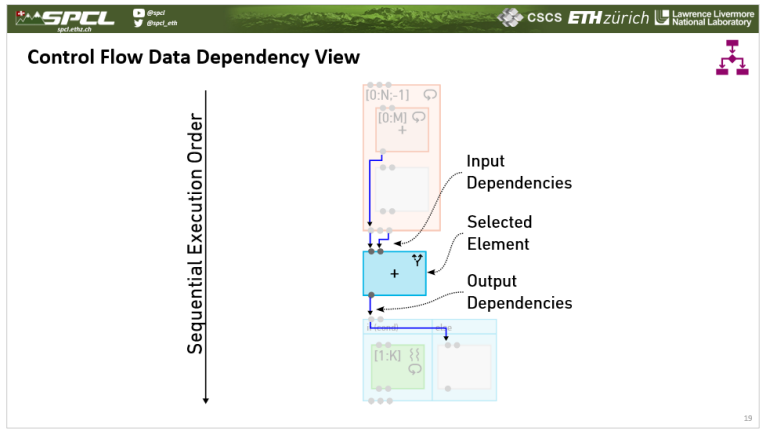


Conclusions









More of SPCL's research:

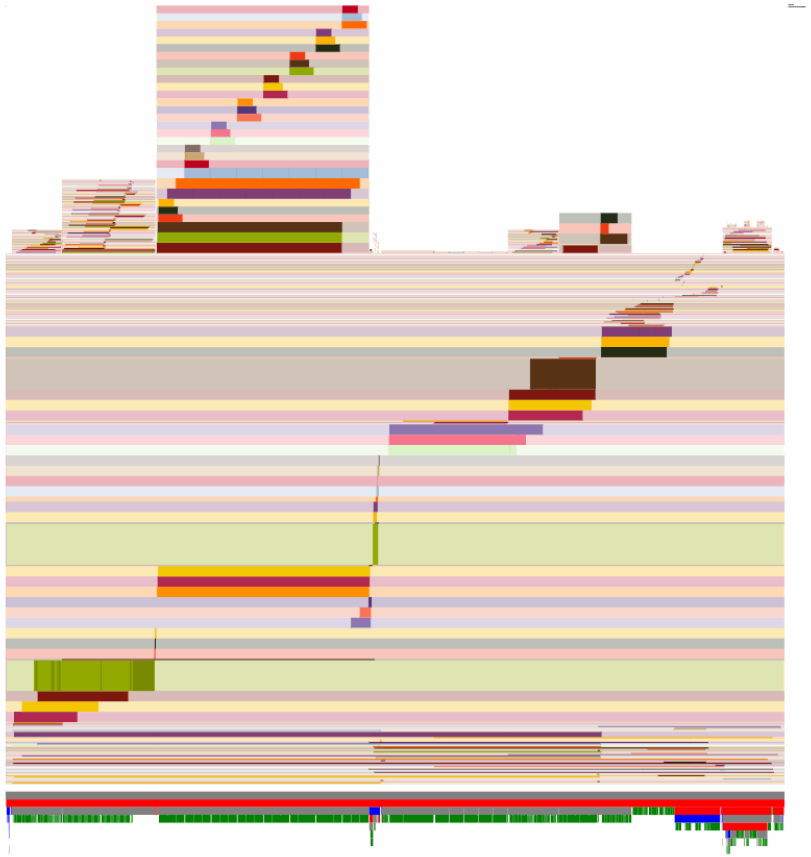
youtube.com/@spcl — 240+ Talks

x.com/spcl_eth — 1.7K+ Followers

github.com/spcl — 7.2K+ Stars

… or spcl.ethz.ch

Conclusions



More of SPCL's research:

youtube.com/@spcl — 240+ Talks

x.com/spcl_eth — 1.7K+ Followers

github.com/spcl — 7.2K+ Stars

… or spcl.ethz.ch

Data-Dependent Program: HPCG