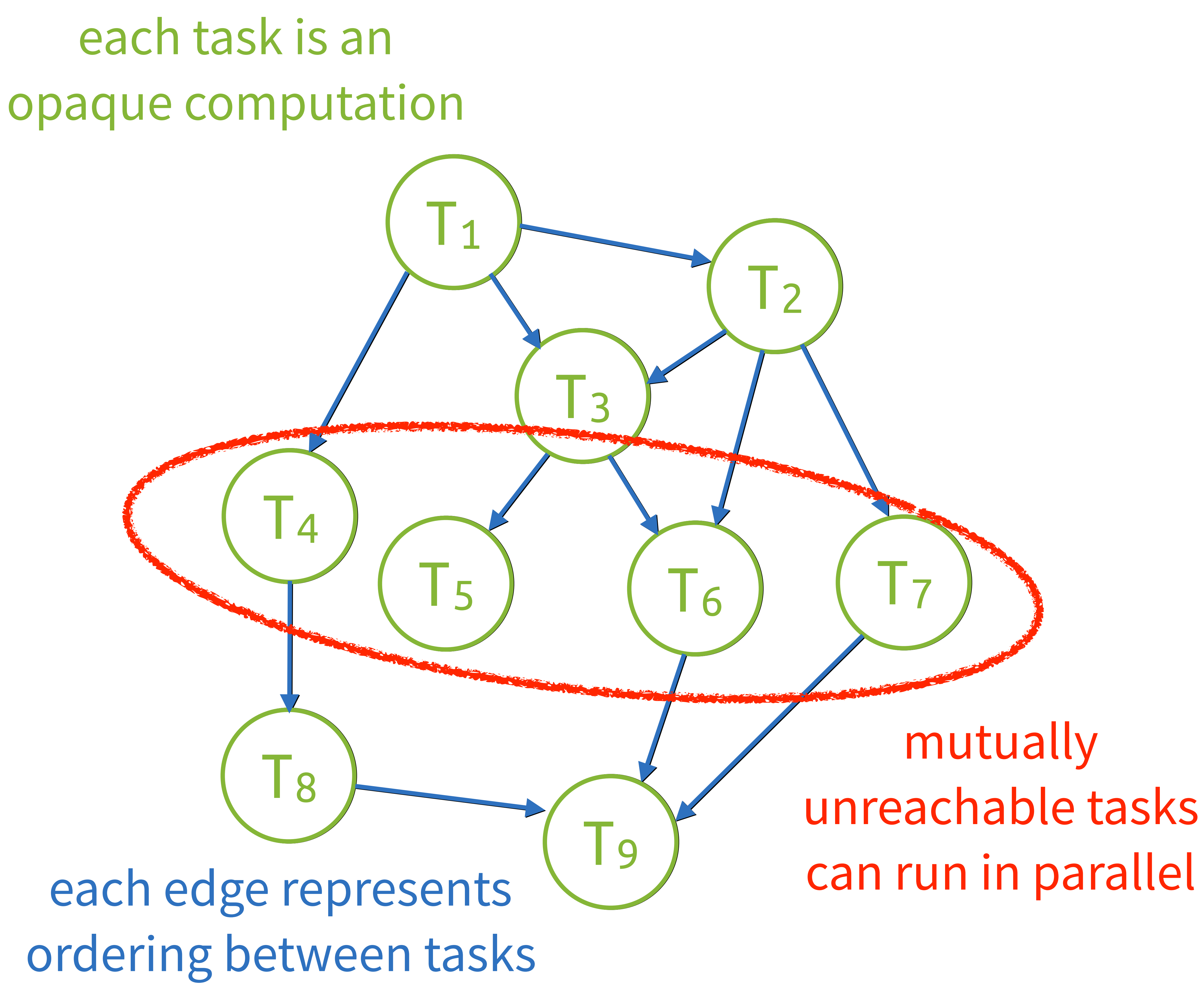
Dynamic Tracing: Memoization of Task Graphs for Dynamic Task-Based Runtimes

Wonchan Lee, Todd Warszawski, Alex Aiken	Stanford University
Elliott Slaughter	SLAC National Accelerator Laboratory
Michael Bauer, Sean Treichler, Michael Garland	NVIDIA

November 14, 2018 @ SC'18

Task Graphs Simplify Distributed Programming

Task graph is a DAG of tasks where



- Parallel execution is “straightforward” with task graphs
- Task graphs are most flexible when dynamically generated
  - Dynamic task graphs also facilitate fault recovery, load balancing, task (re-)mapping, resource allocation, etc.

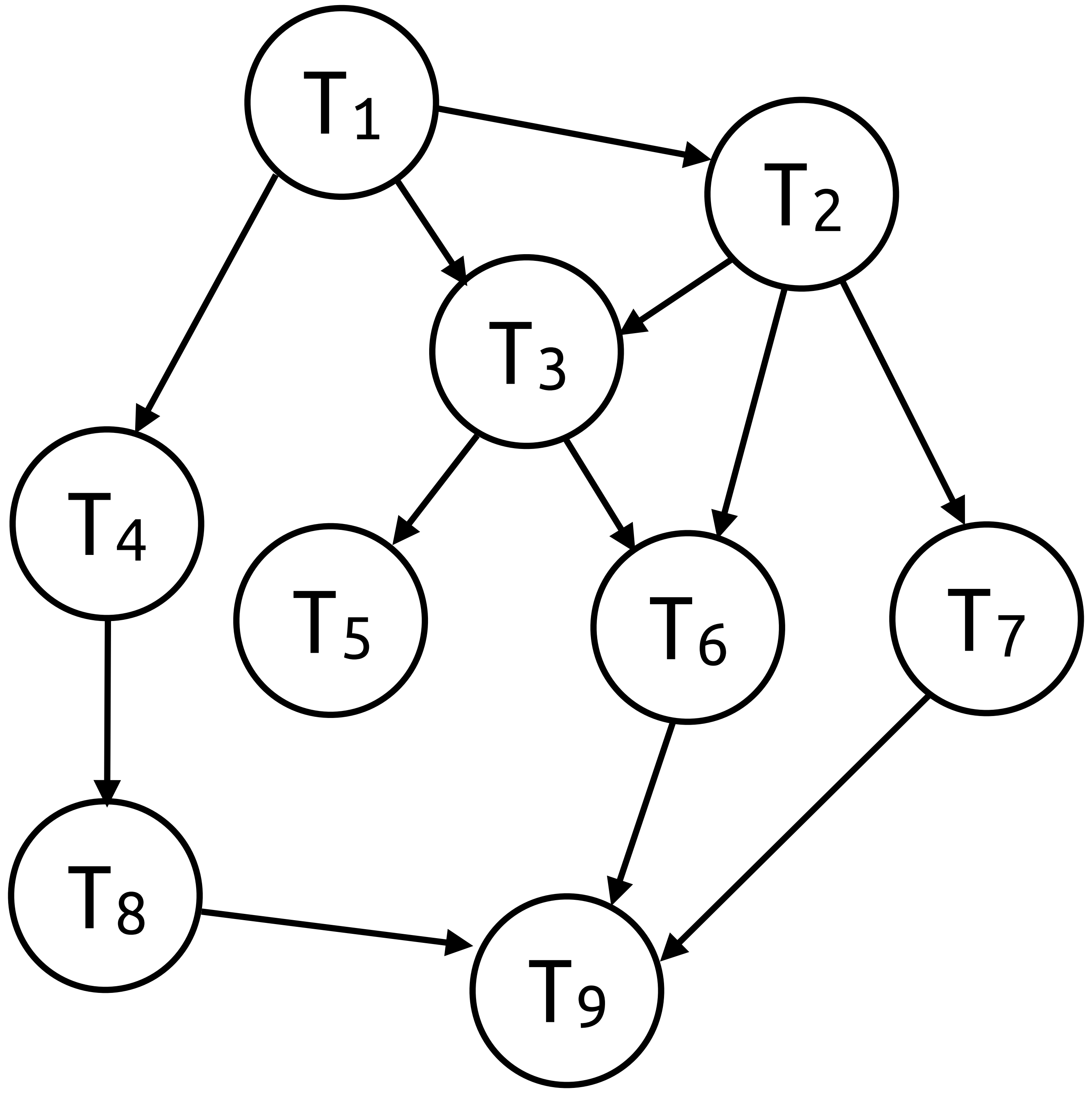
Approaches to Dynamic Task Graph Construction

Explicit construction

Program = Graph generator

```
g = new TaskGraph()
g.add(T1)
g.add(T2 ← T1)
g.add(T3 ← {T1, T2})
g.add(T4 ← T1)
…
```

Exec.



✔ Efficient
✘ Error-prone
✘ Not composable

e.g., Realm, CUDA

Implicit construction

Program = Task generator

Dep. Analysis

```
T1(A,B) T2(A) T3(A) T4(B) ...
```

Exec.

```
T1(A,B) // writes(A),reads(B)
T2(A)   // reads(A)
T3(A)   // writes(A)
T4(B)   // writes(B)
```

✔ Correct by construction
✔ Composable
✘ Runtime overhead

Dynamic task-based runtimes (Legion, StarPU, PaRSEC, PyTorch, etc.)

Is there a hybrid approach that enjoys benefits of both?

Dynamic Tracing: Memoizing Task Graphs

- Bring efficiency of explicit construction to dynamic task-based runtimes
- Key observation: programs often have *traces* of repetitive tasks
  - The same traces produce the same subgraph

```
task T(x,y) writes(x),reads(y)
task U(x,y) reads(x), reads(y)

while (*):
  T(A,B); T(C,D)
  U(A,D); U(C,B)
```

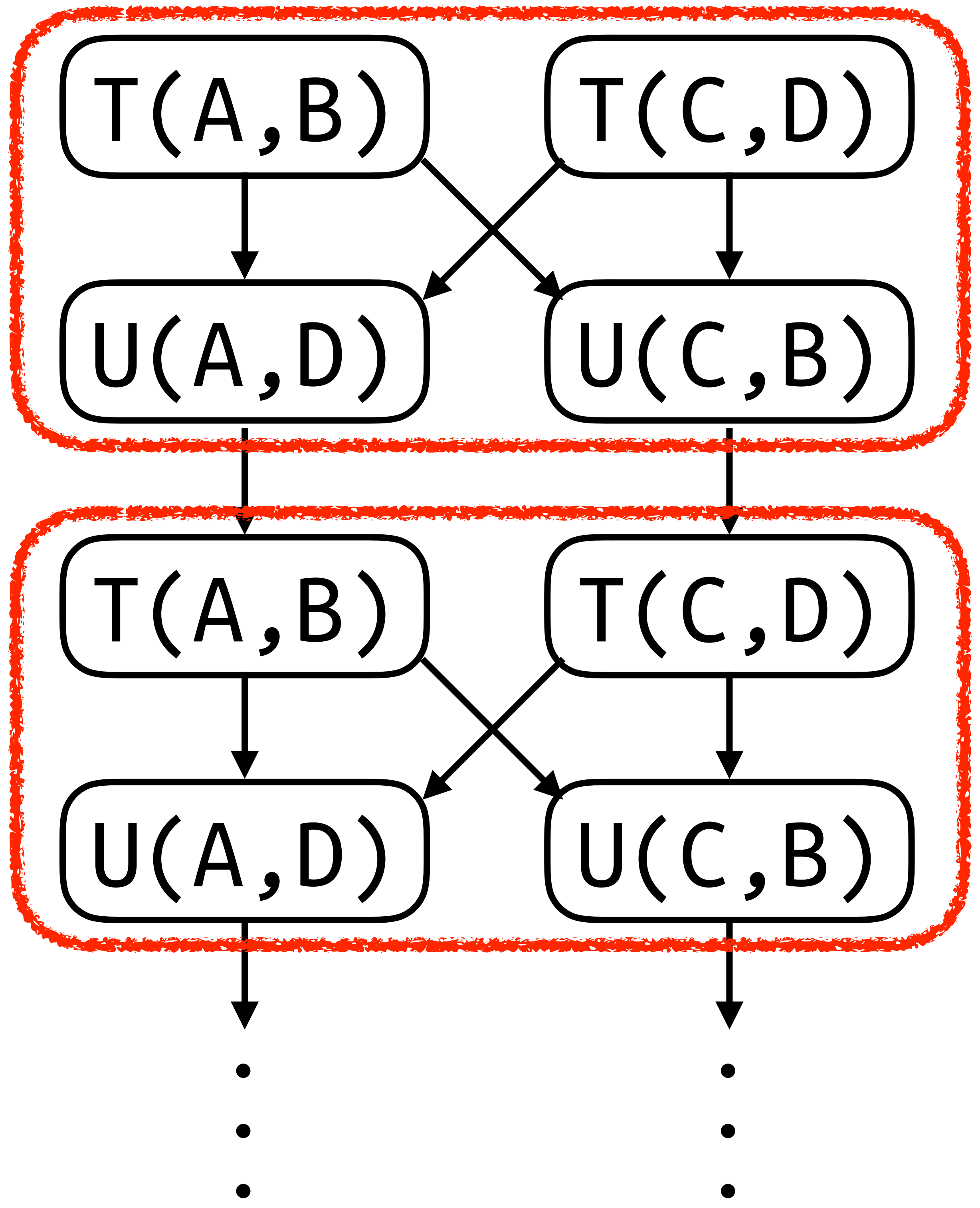
Exec.

(T(A,B) T(C,D) U(A,D) U(C,B))*

Repetitive tasks

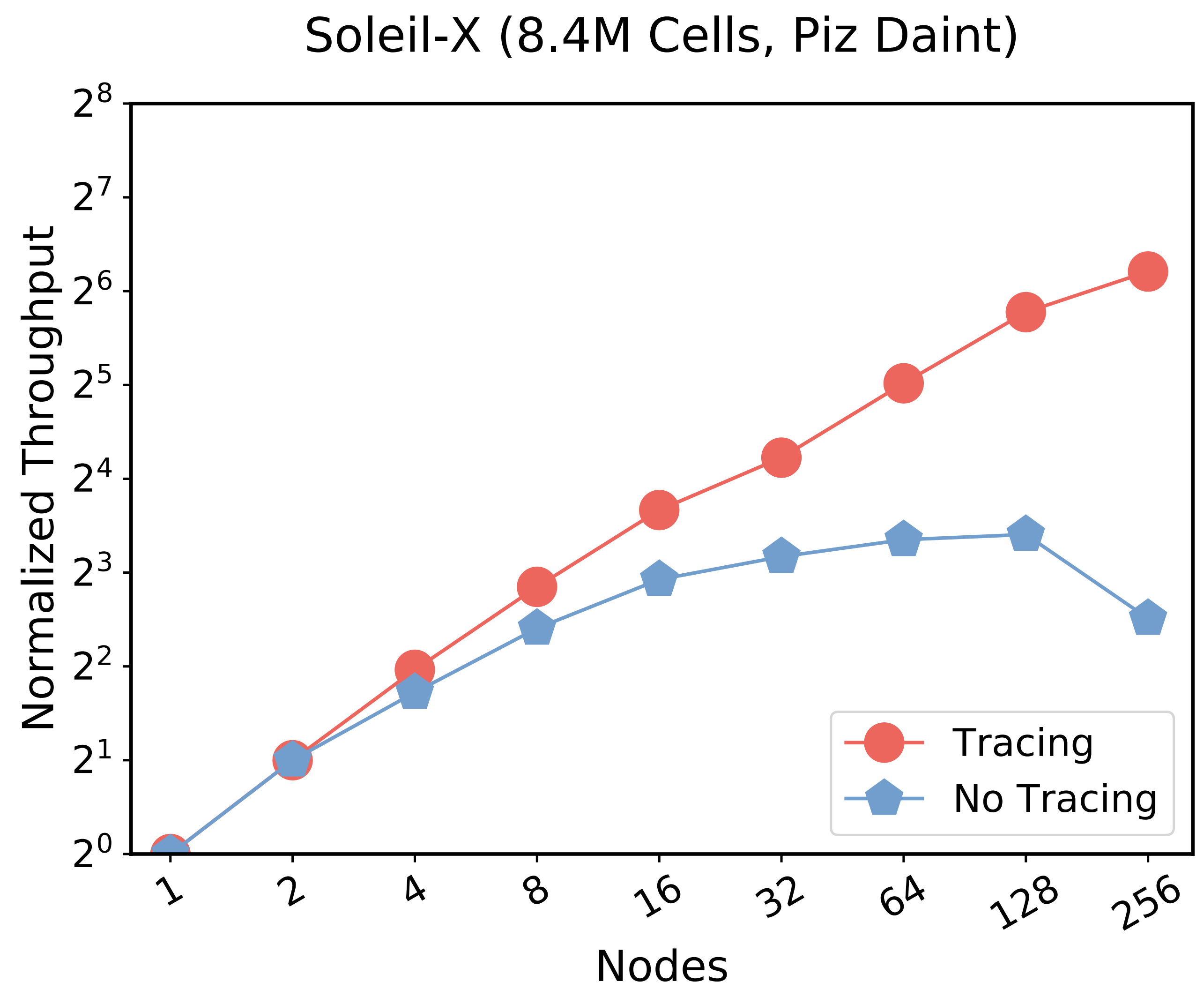
Dep. Analysis

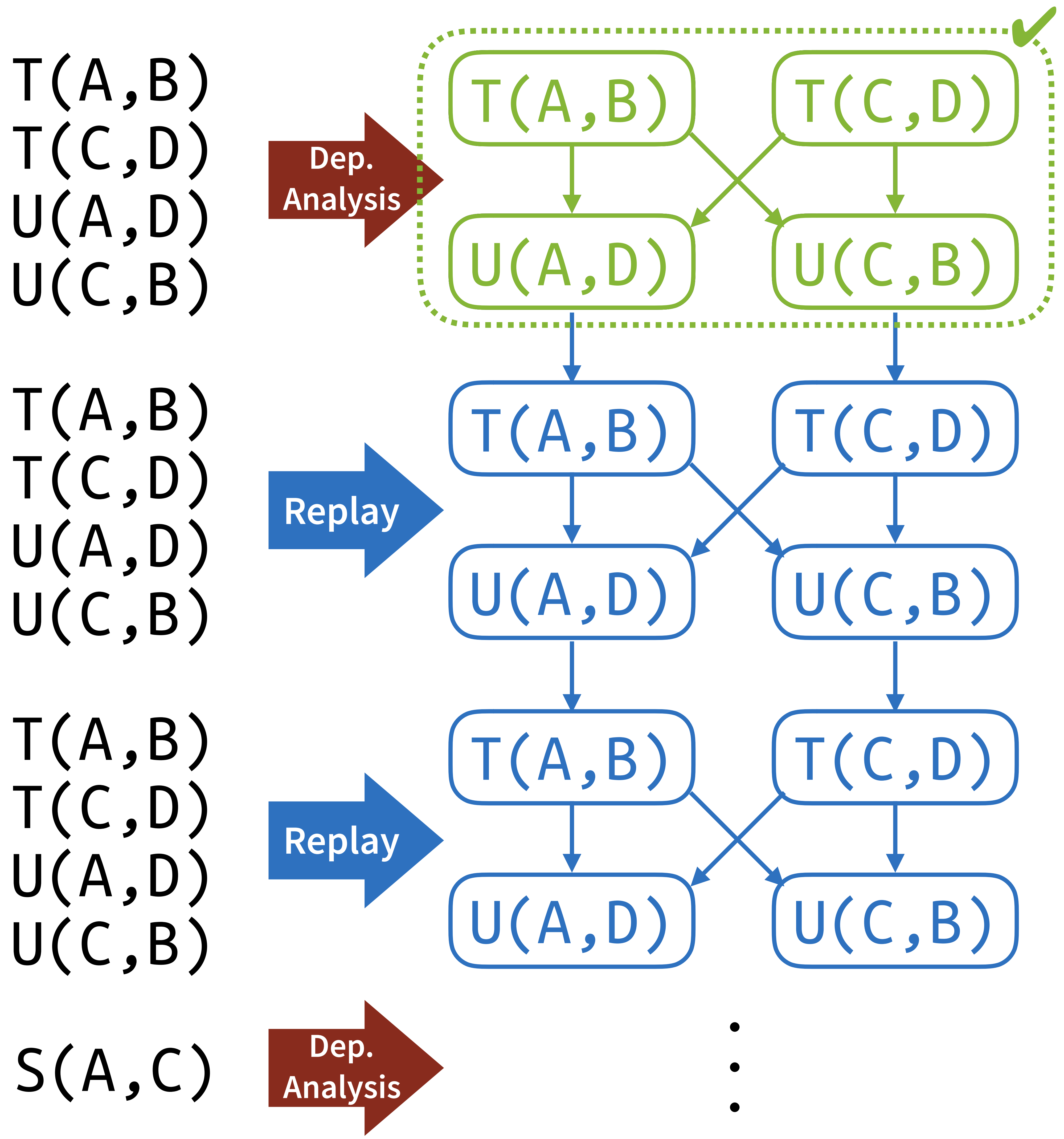
Subgraphs are isomorphic!



Dynamic Tracing: Memoizing Task Graphs

- Idea: record-and-replay
  - Record the subgraph once for a trace
  - Replay the recording whenever applicable
- **Improves strong scaling performance by 4.9X**





Contents

- Programming model
- Baseline dependence analysis
- Challenges in dynamic tracing
- Optimizations
- Experiment results

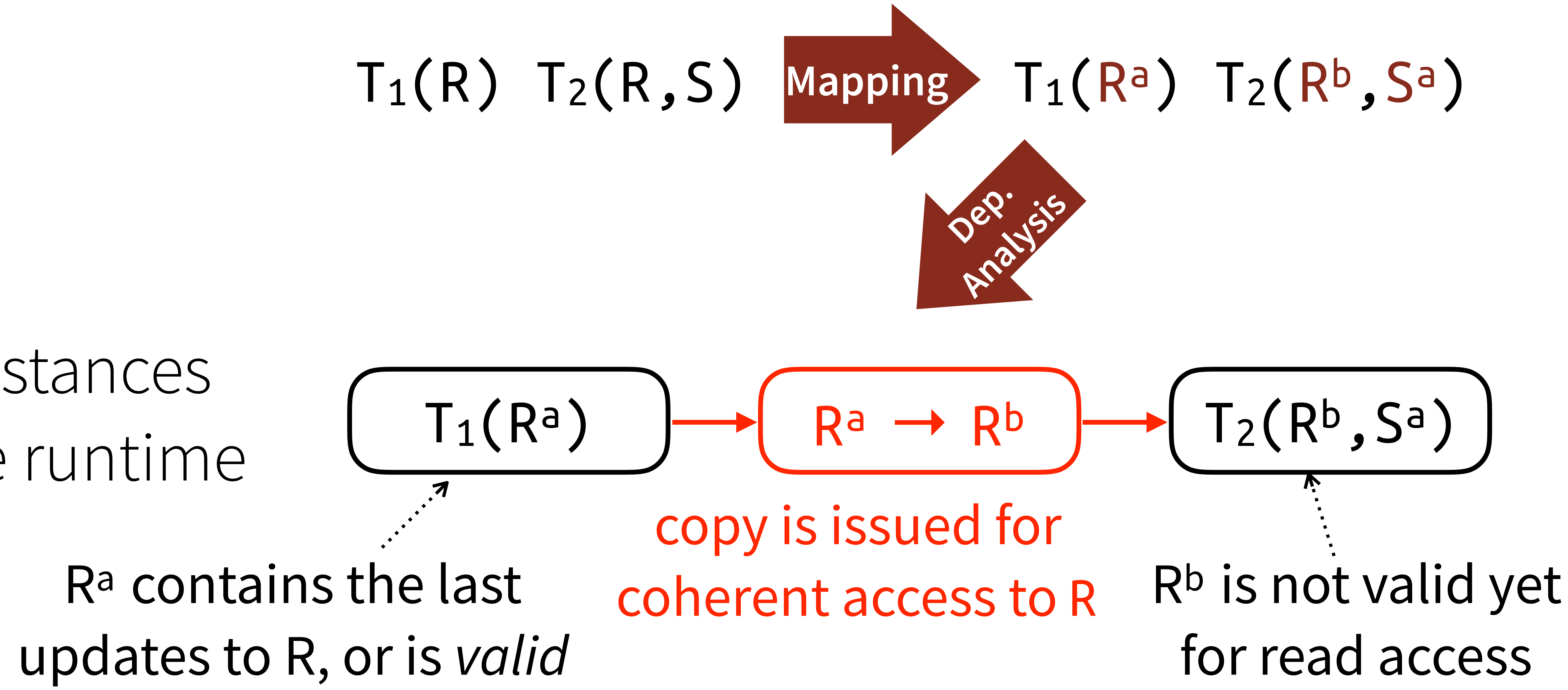
Programming Model

- Task-based
  - Programs consist of tasks
  - Tasks use regions and declare permissions

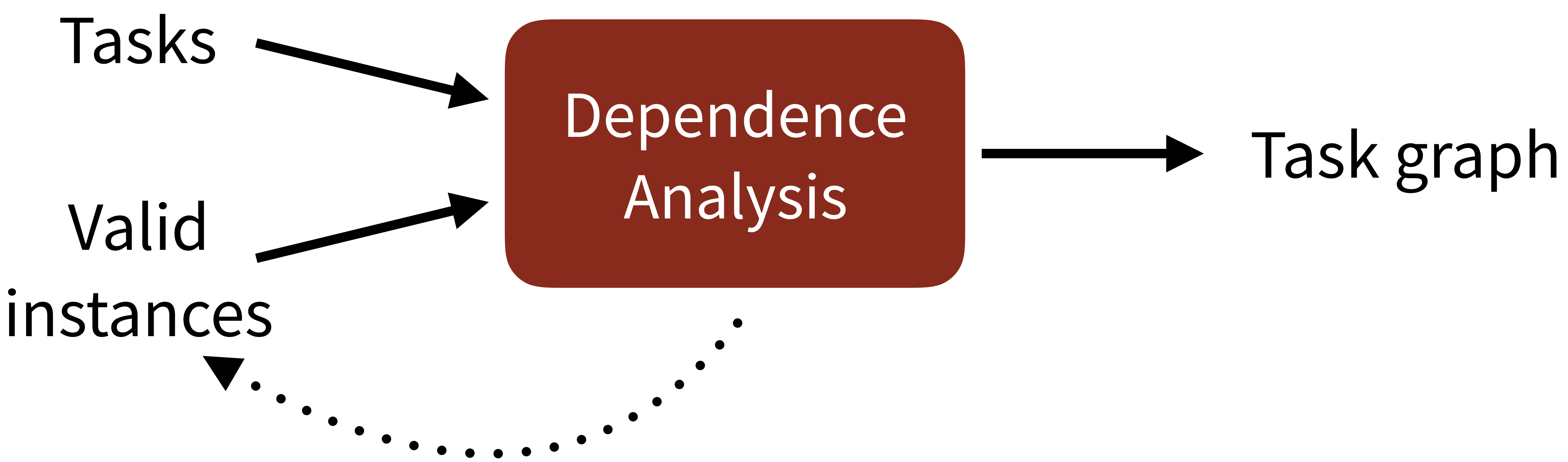
```
task T1(x)   reads(x),writes(x)
task T2(x,y) reads(x),writes(y)

T1(R)
T2(R,S)
```

- Distributed
  - Regions must be mapped to instances
  - One region can be mapped to multiple instances
    → *Coherence* must be maintained by the runtime

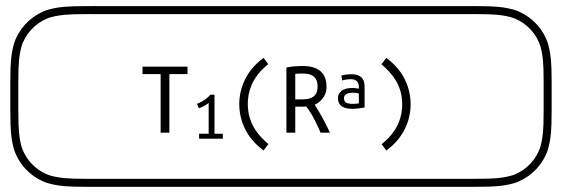
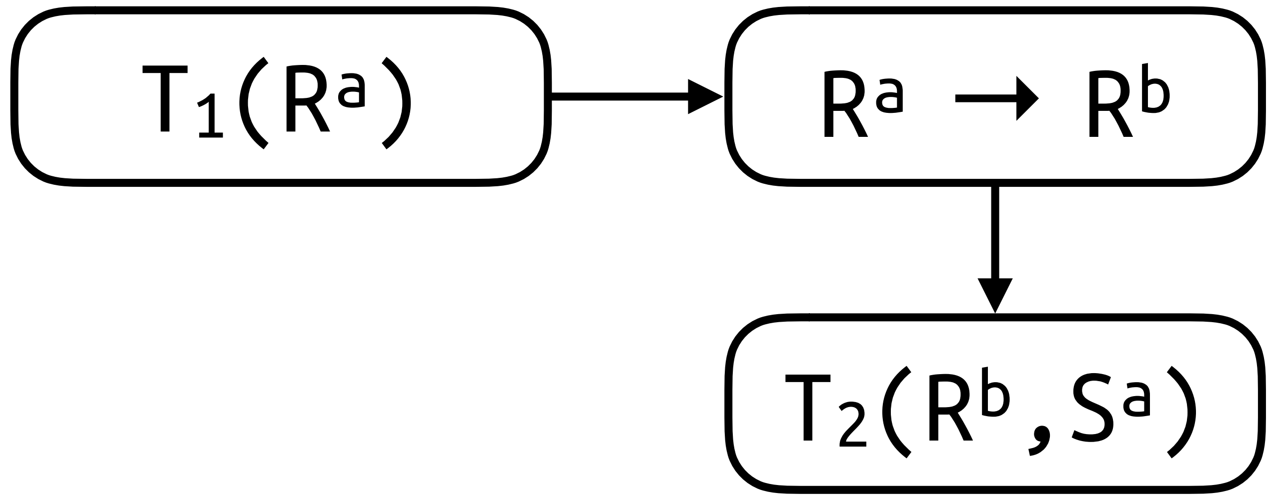
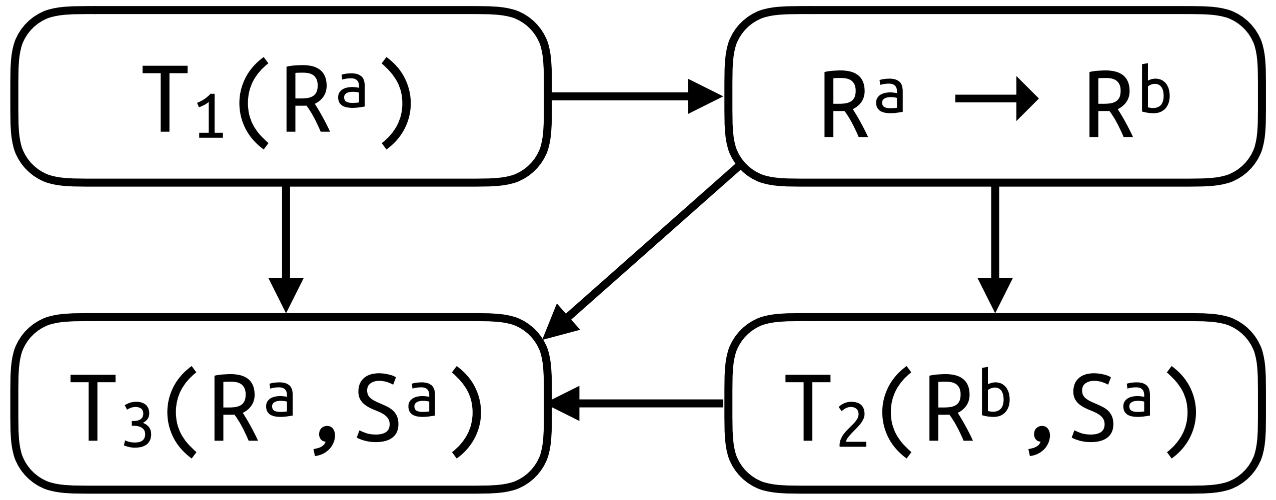


Baseline Dependence Analysis



Baseline Dependence Analysis

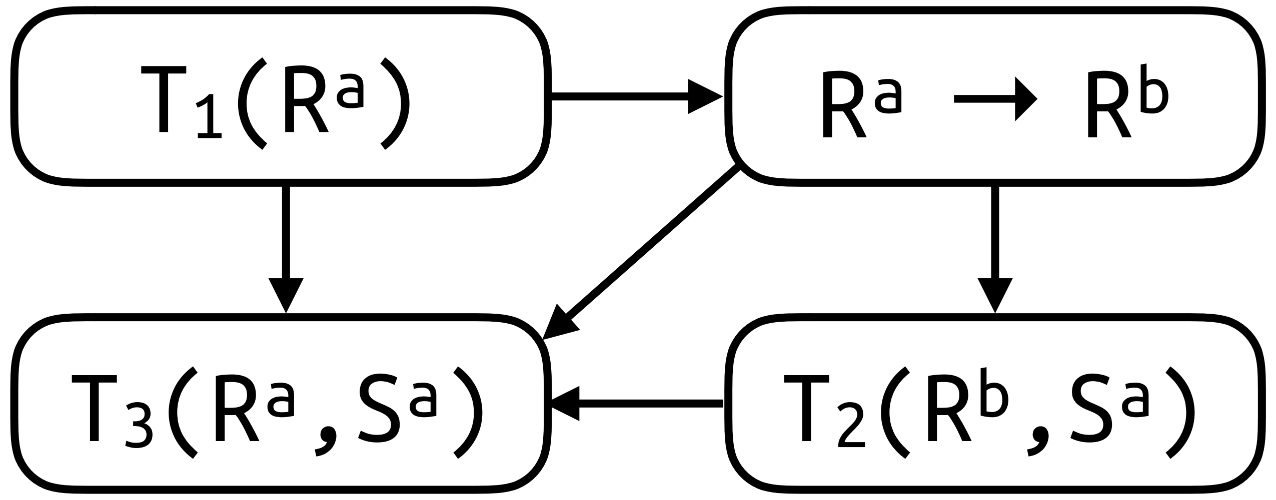
```
T₁(Rᵃ)        reads (Rᵃ),writes(Rᵃ)
T₂(Rᵇ,Sᵃ)     reads (Rᵇ),writes(Sᵃ)
T₃(Rᵃ,Sᵃ)     writes(Rᵃ),reads (Sᵃ)
```

Task	Valid Instances	Task graph
$T_1(R^a)$	R: R^a	
$T_2(R^b,S^a)$	R: R^b,R^a S: S^a	
$T_3(R^a,S^a)$	R: R^a S: S^a	

Challenges in Dynamic Tracing

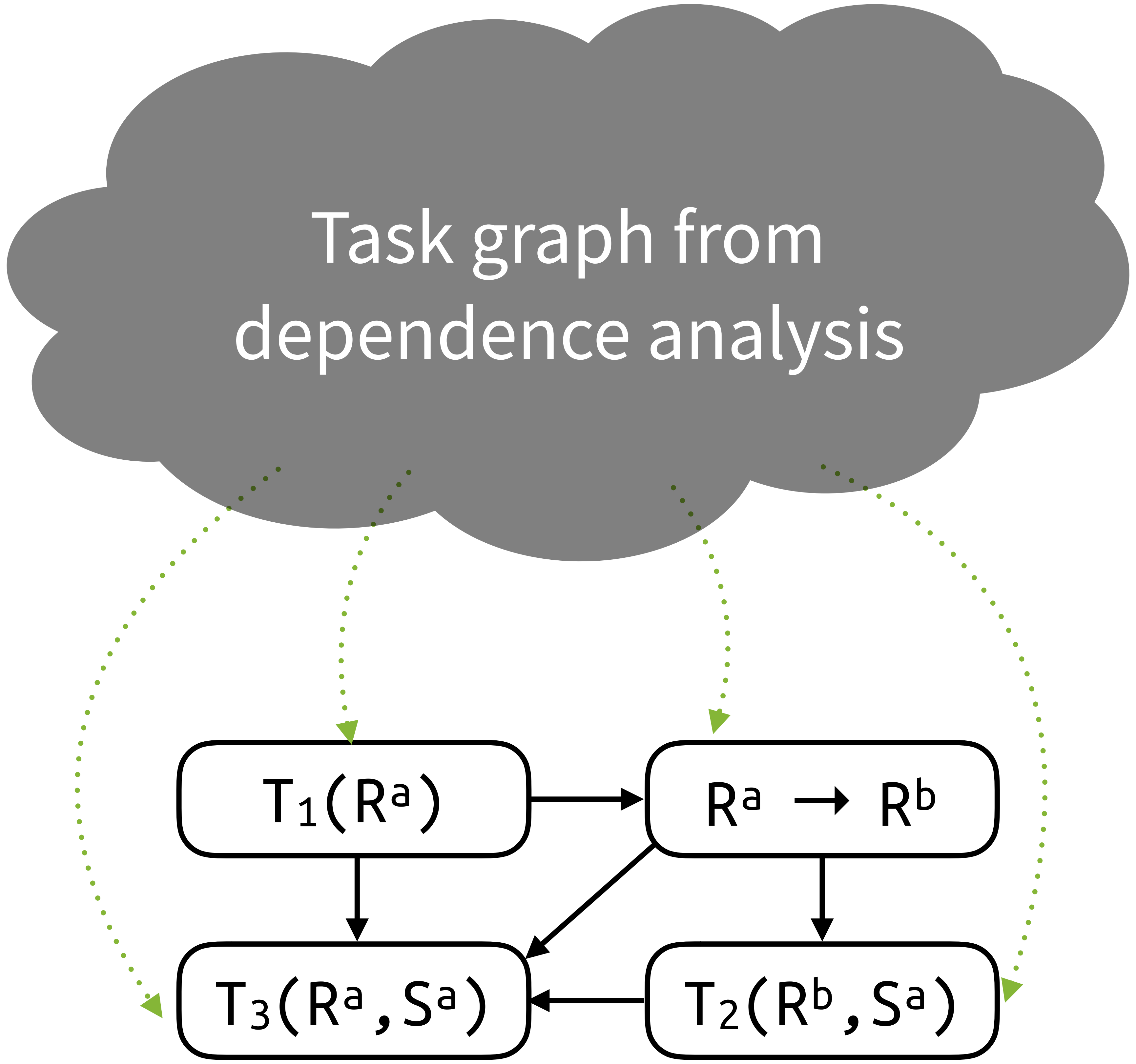
- Challenge 1: transition from dep. analysis to subgraph replay

Captured subgraph **G**



Found the same trace

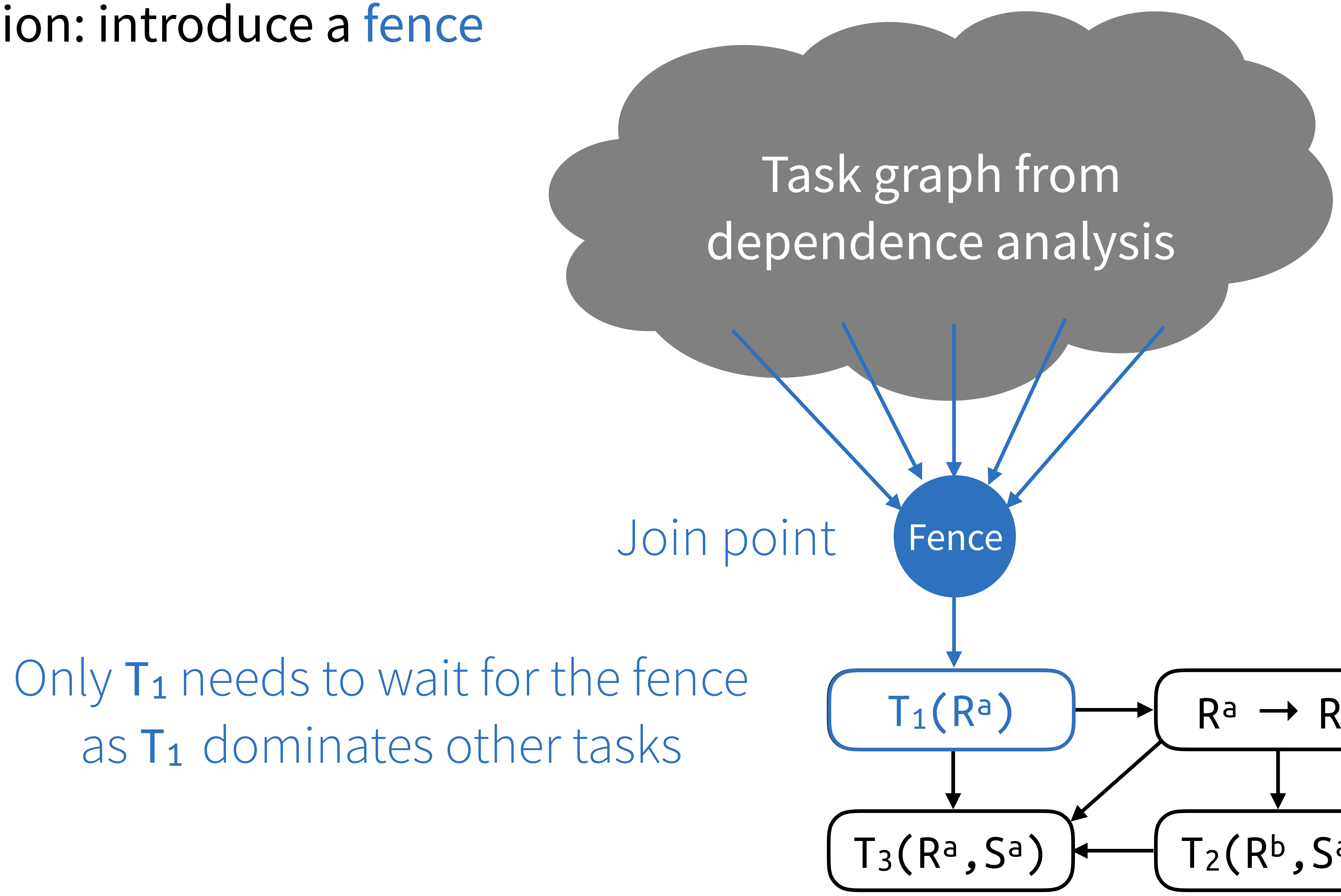
... $T_1(R^a)$ $T_2(R^b, S^a)$ $T_3(R^a, S^a)$ ...



How can we connect G to the graph from dep. analysis?

Challenges in Dynamic Tracing

- Solution: introduce a fence

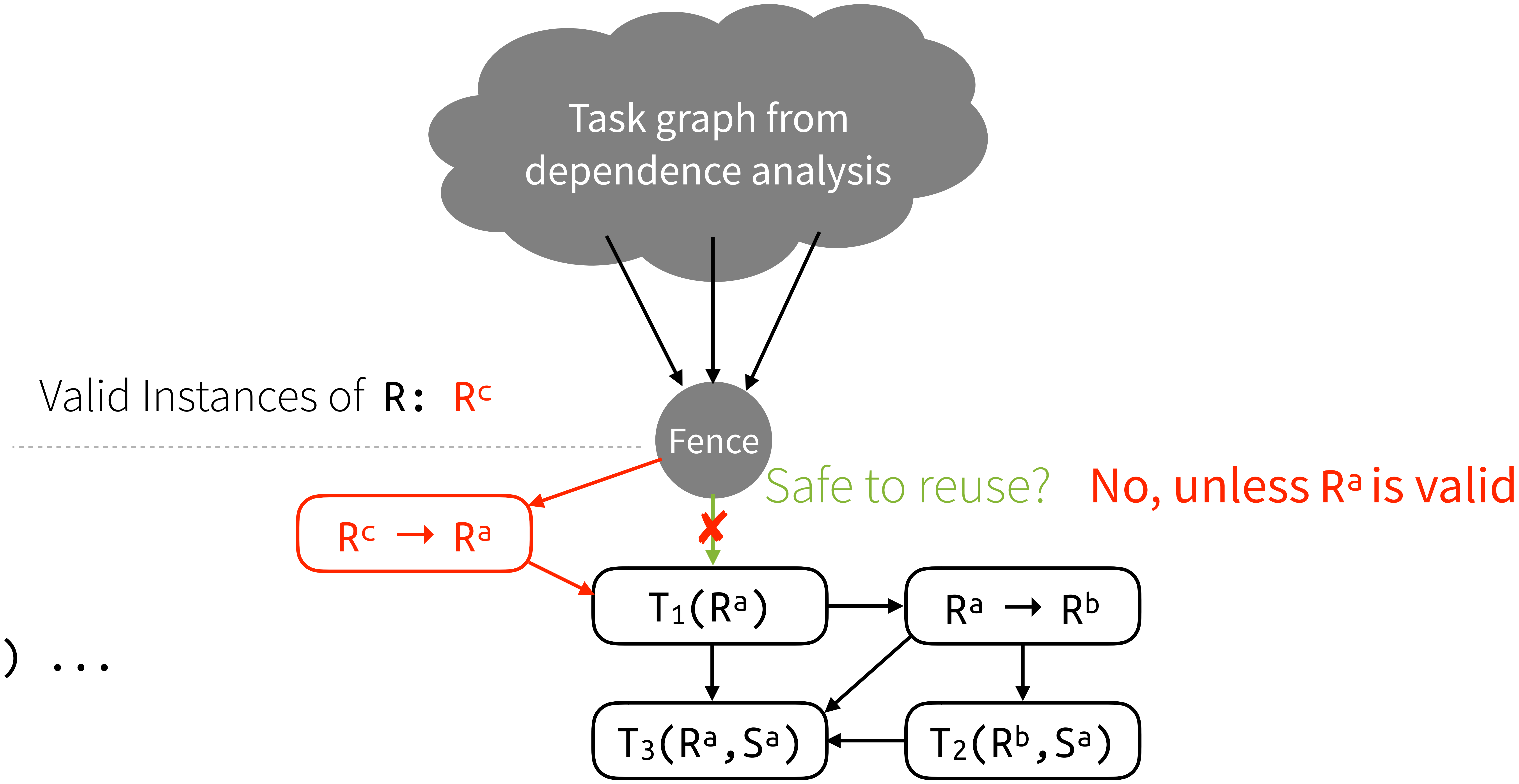


Challenges in Dynamic Tracing

- Challenge 2: coherence

Found the same trace

```
... T₁(Rᵃ) T₂(Rᵇ,Sᵃ) T₃(Rᵃ,Sᵃ) ...
```



Challenges in Dynamic Tracing

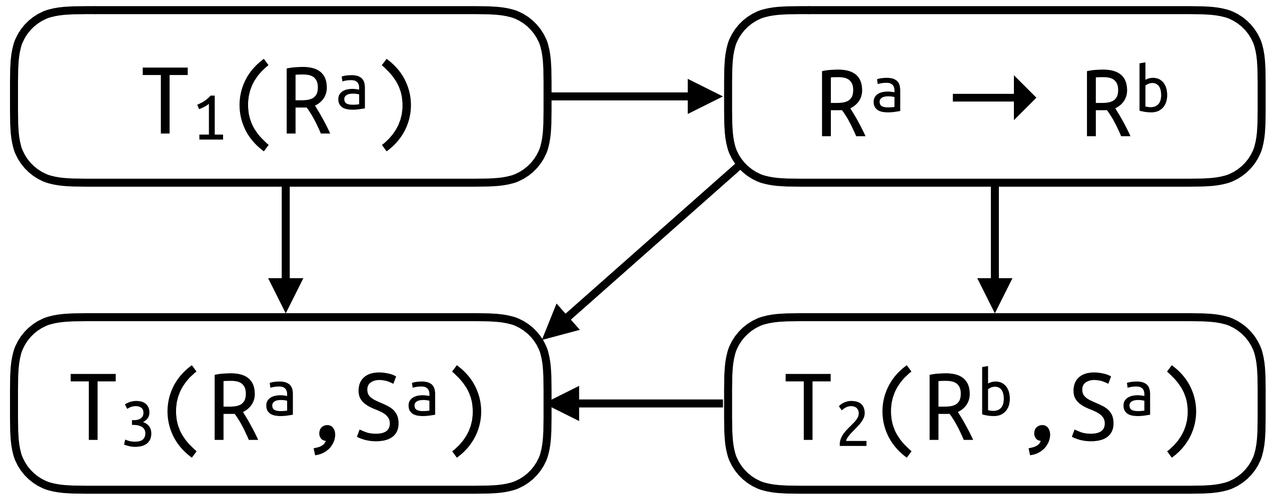
- Solution: remember precondition for a safe replay

Valid instances are input of dependence analysis

Tasks: $T_1(R^a)$ $T_2(R^b,S^a)$ $T_3(R^a,S^a)$

Precondition: R^a is valid

Task graph:

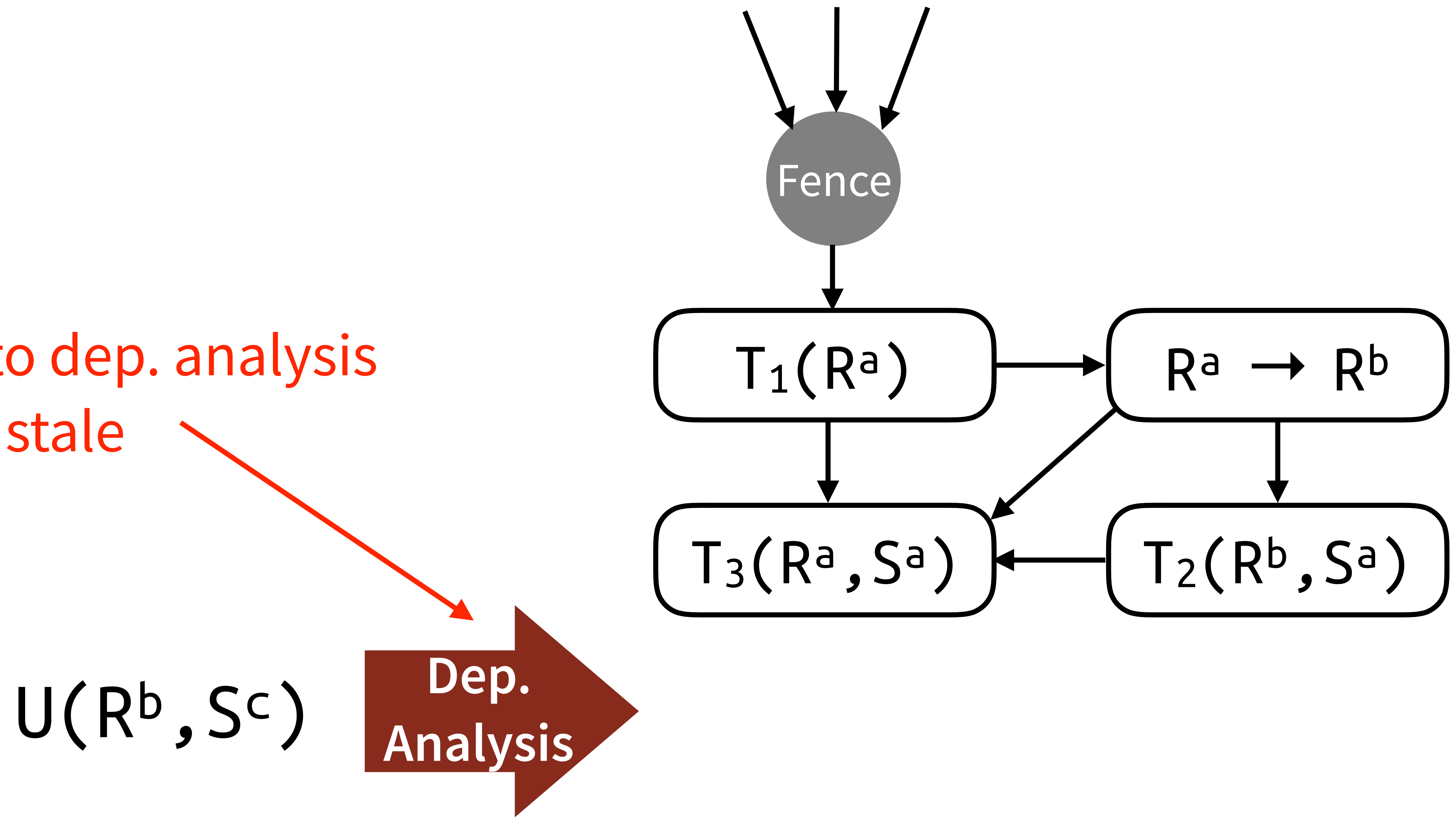


Challenges in Dynamic Tracing

- Challenge 3: transition back to normal dep. analysis

Two inconsistencies

1. T_1, T_2, and T_3 are unknown to dep. analysis
2. The list of valid instances is stale



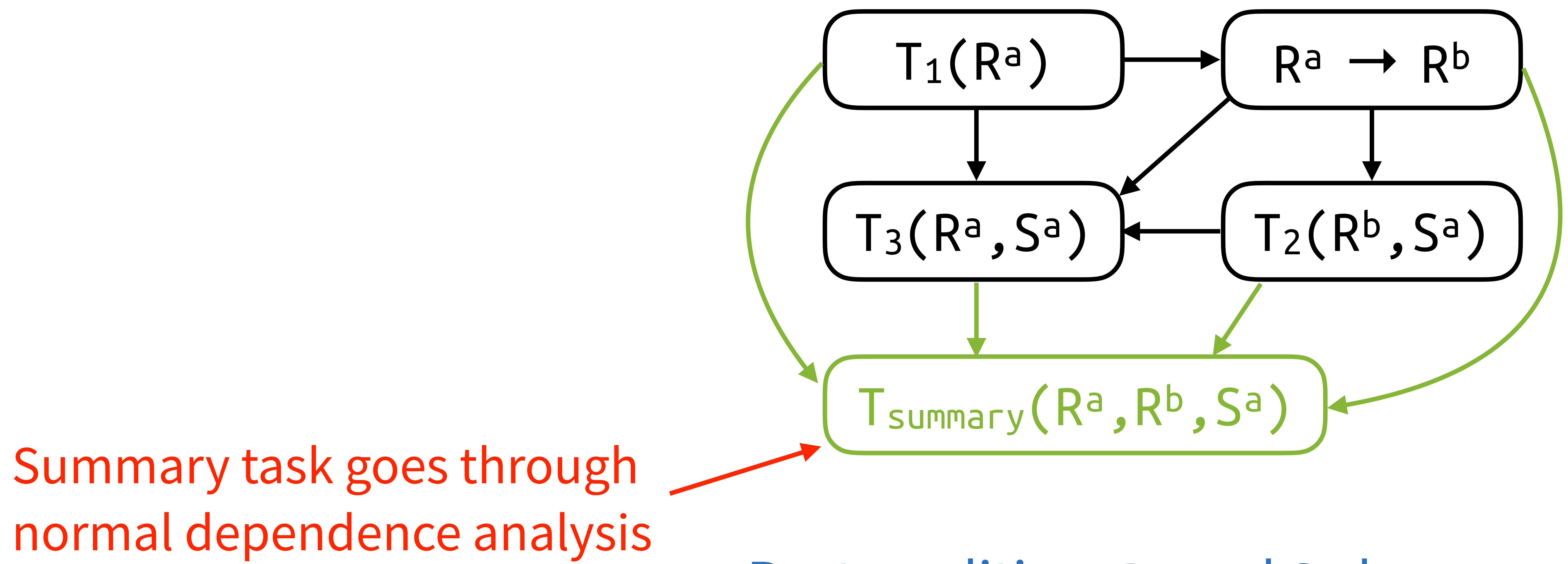
Challenges in Dynamic Tracing

- Solution
  - Make a summary task
  - Compute postcondition to apply after each replay

Tasks: $T_1(R^a)$ $T_2(R^b,S^a)$ $T_3(R^a,S^a)$

Precondition: R^a is valid

Task graph:



Postcondition: R^a and S^a become valid

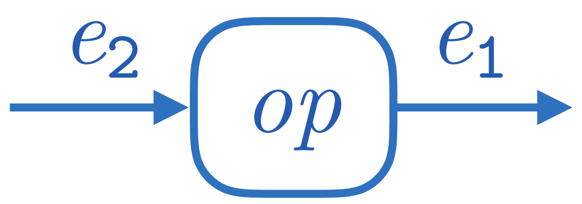
Graph Calculus

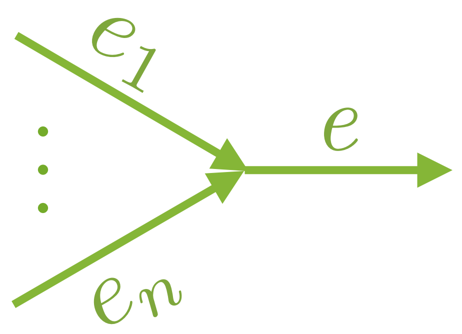
- Simple graph construction language
  - Use *events* that signify termination of operations
  - Syntax: $c ::= e_1 := \texttt{op}(op, e_2) \mid e := \texttt{merge}(\overline{e_i}) \mid e := \texttt{fence} \mid c; c$

op waits until e_2 is signaled and triggers e_1 when it's done

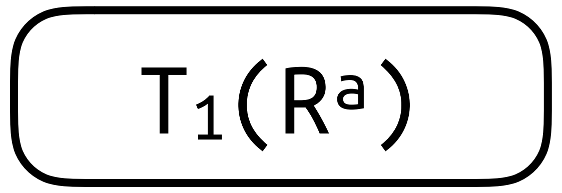
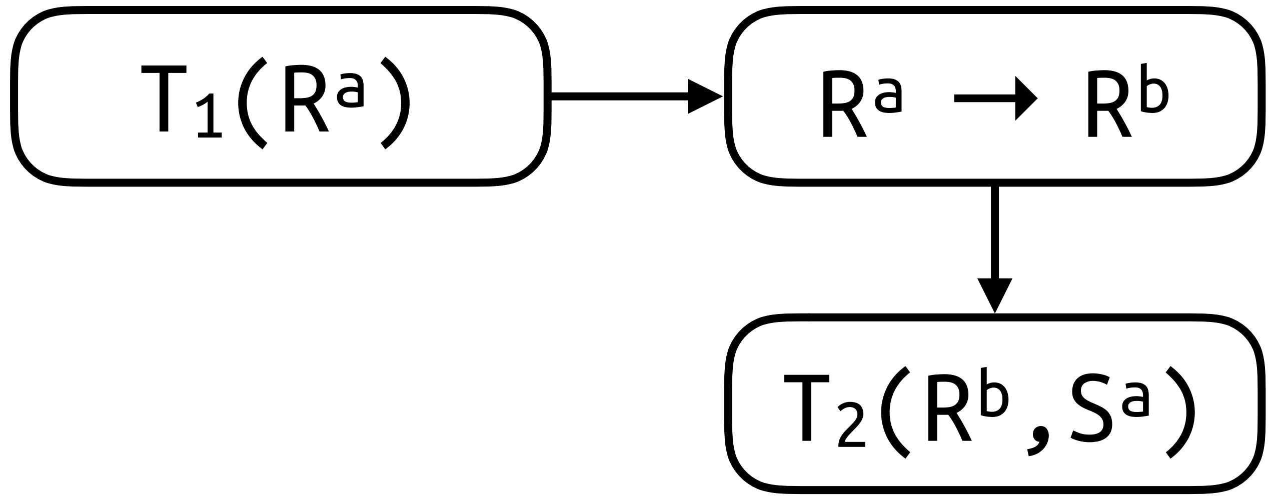
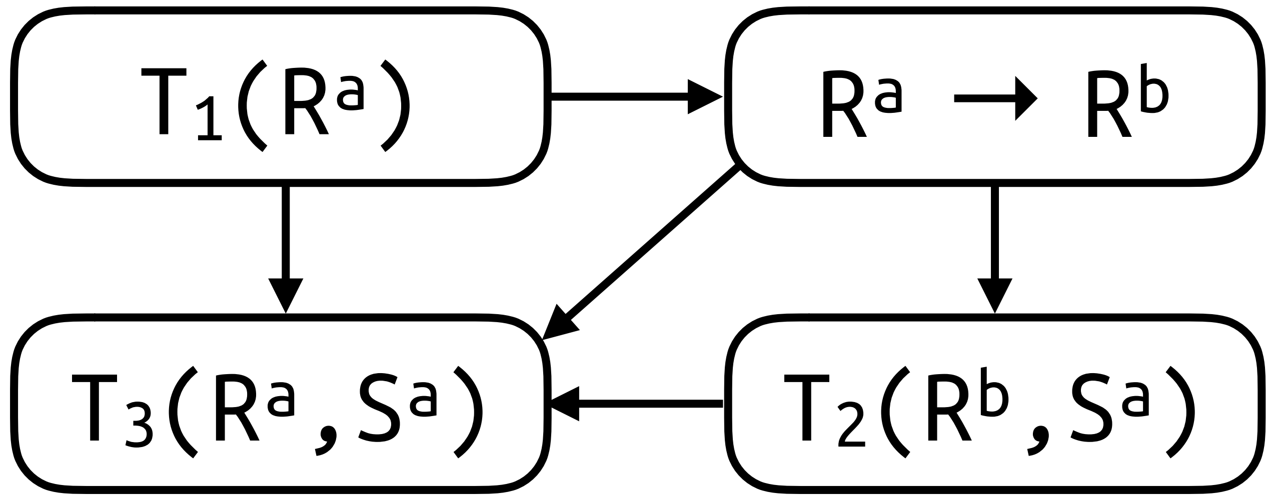
e is triggered when $e_1, \ldots, e_n$ are

issue a fence that signals e





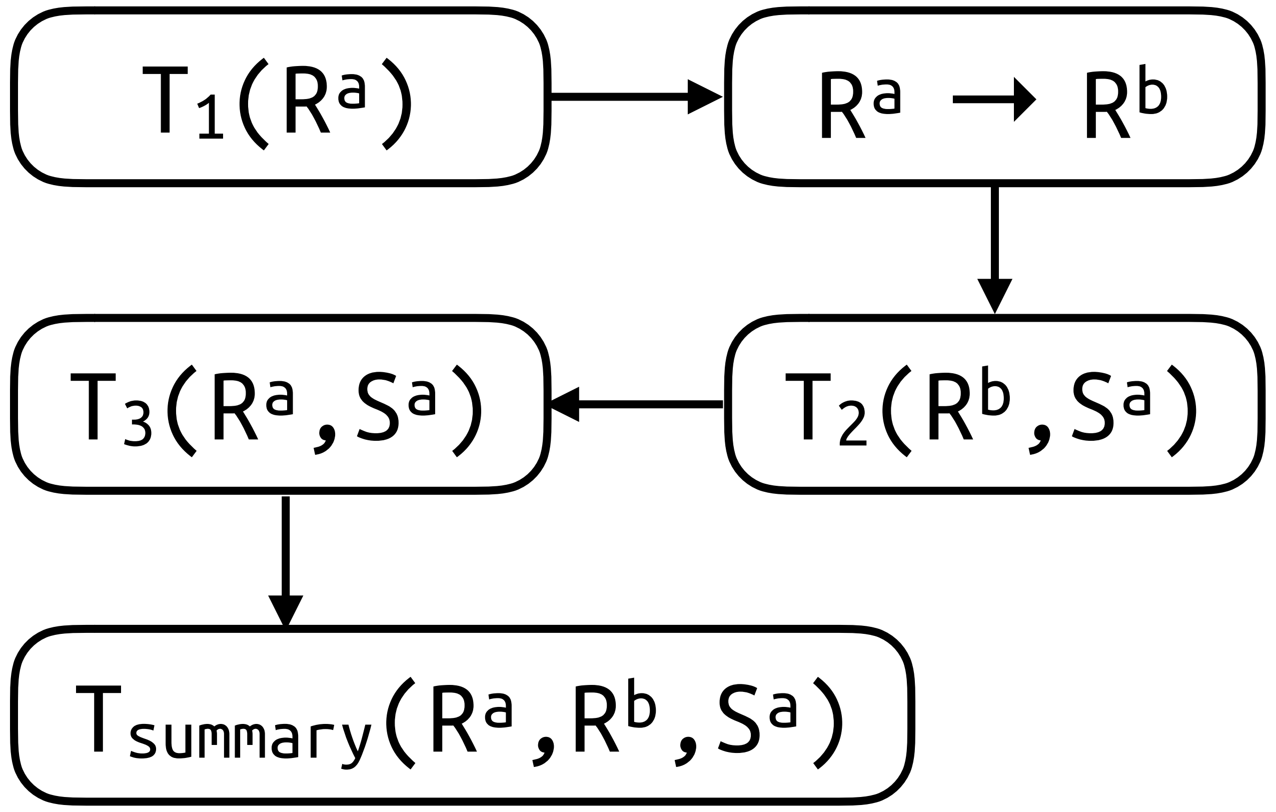
Trace Recording Example

Task	Task graph	Command	Recording state
$T_1(R^a)$		$e_1 := \text{fence}$ $e_2 := \text{op}(T_1(R^a),\ e_1)$	$T_1(R^a) = e_2$
$T_2(R^b,S^a)$		$e_3 := \text{op}(R^a \rightarrow R^b,\ e_2)$ $e_4 := \text{op}(T_2(R^b,S^a),\ e_3)$	$R^a \rightarrow R^b = e_3$ $T_2(R^b,S^a) = e_4$
$T_3(R^a,S^a)$		$e_5 := \text{merge}(e_2,\ e_3,\ e_4)$ $e_6 := \text{op}(T_3(R^a,S^a),\ e_5)$	$T_3(R^a,S^a) = e_6$
Insert summary task		$e_7 := \text{merge}(e_2,\ e_3,\ e_4,\ e_6)$ $e_8 := \text{op}(T_{summary}(R^a,R^b,S^a),\ e_7)$	Pre: R^a Post: R^a,S^a

Idempotent Recordings

- When the postcondition subsumes the precondition

Task graph G:



Precondition: R^a is valid

Postcondition: R^a and S^a become valid

→ Precondition is satisfied immediately after postcondition is applied

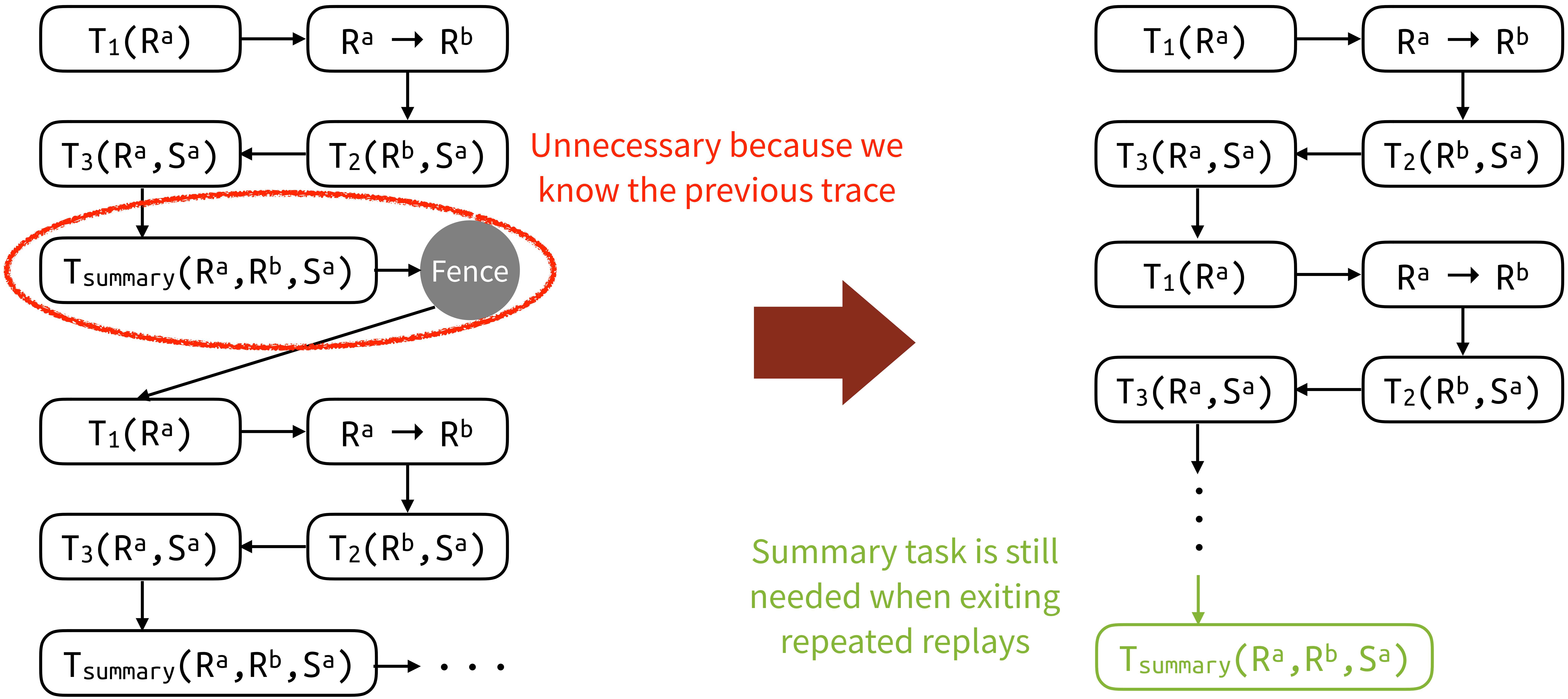
- Optimization: precondition check elision (when the same trace repeatedly appears)

(Check pre. → Replay → Apply post.)*

Check pre. → (Replay)* → Apply post.

Fence Elision

- We can remove summary tasks and fences when we replay the same trace repeatedly

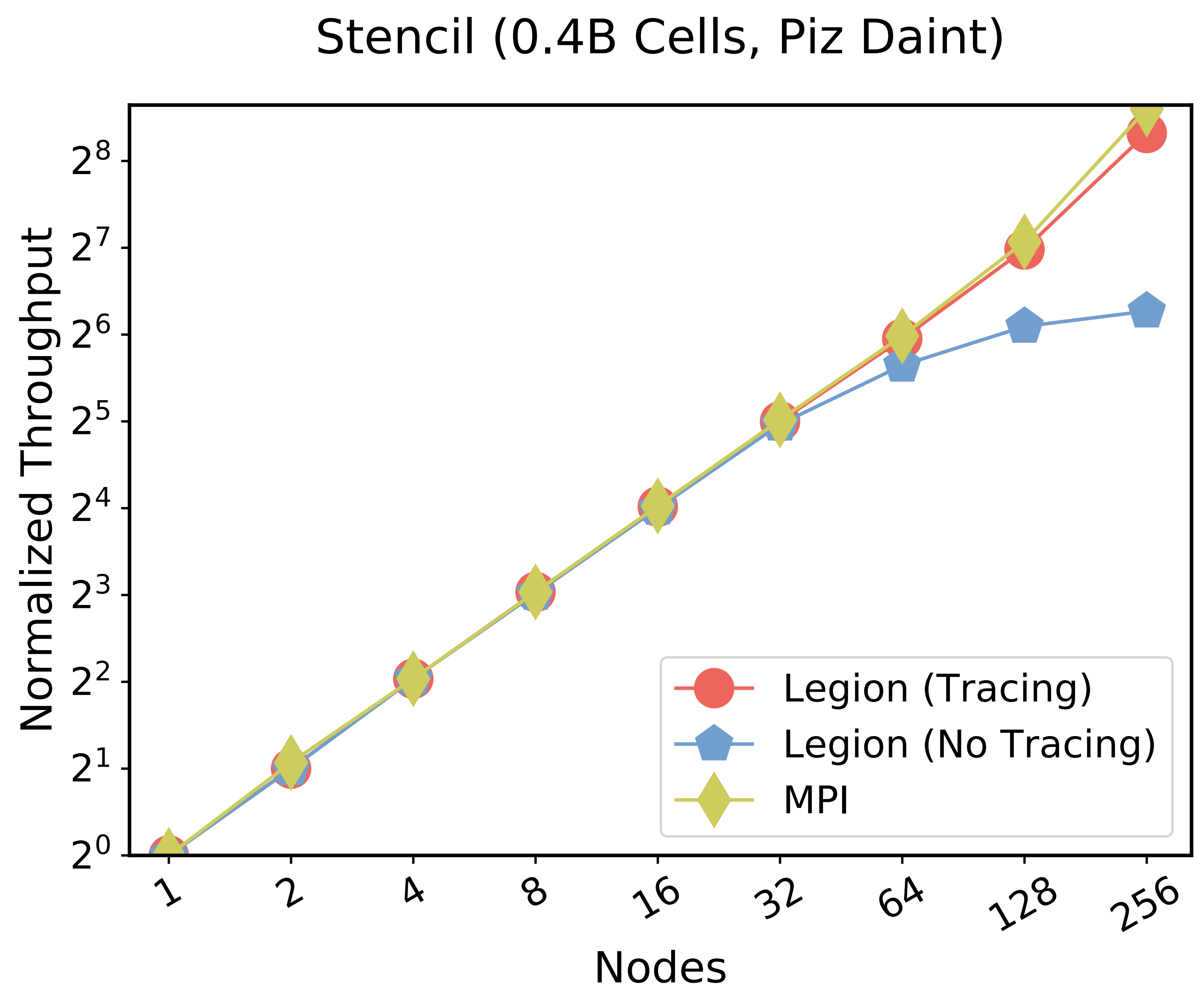


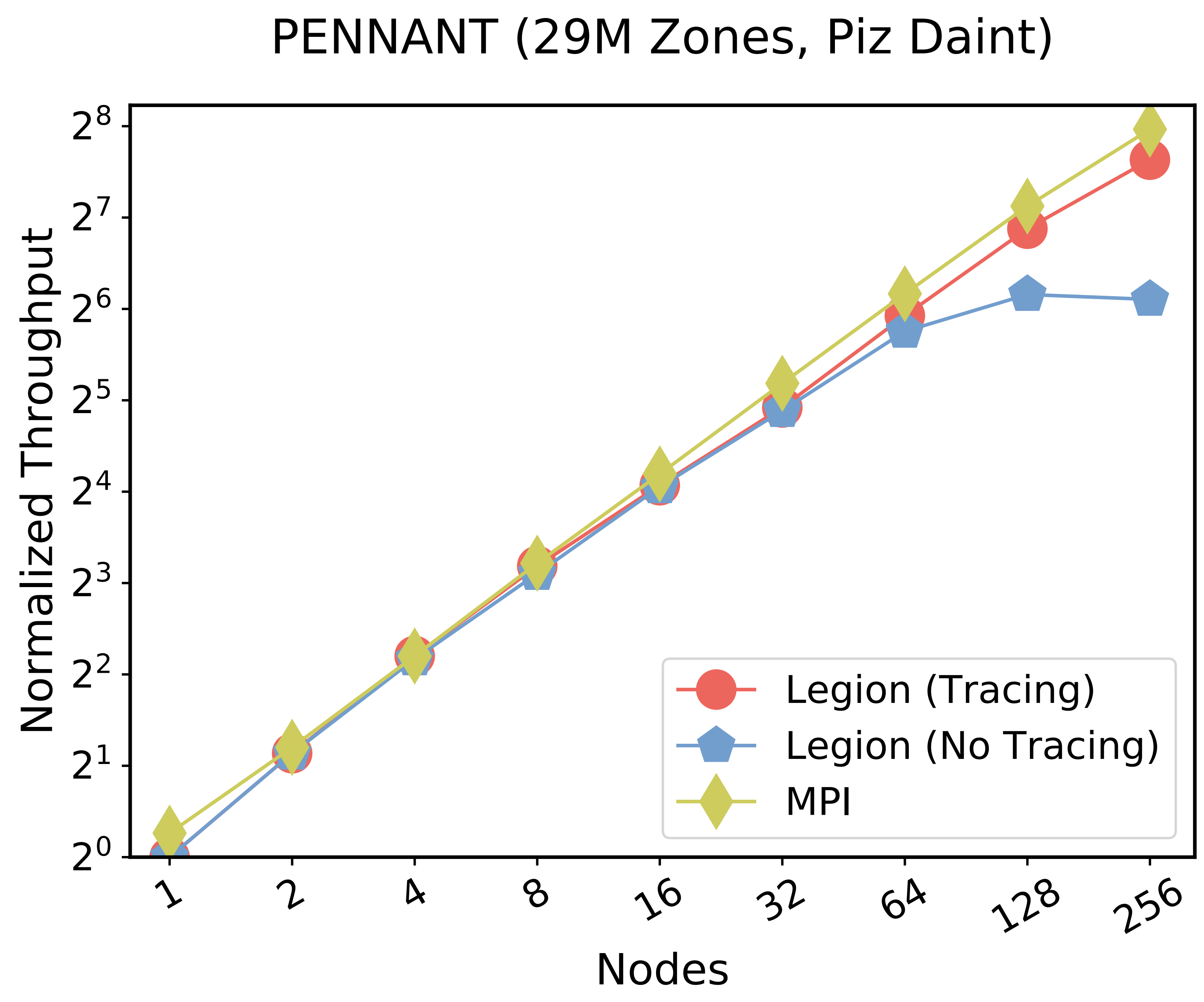
Experiment Results

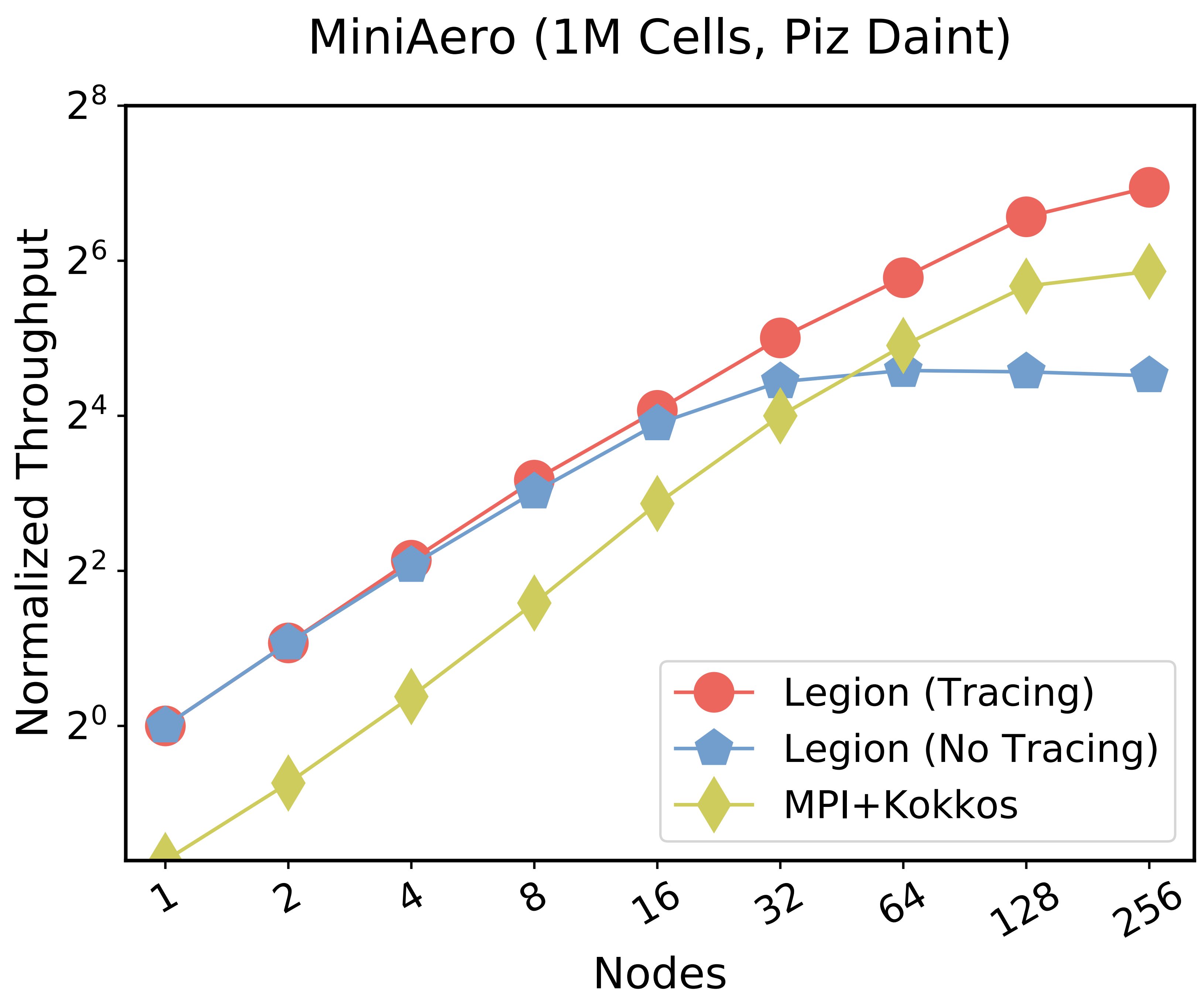
- Implemented dynamic tracing in Legion
- Measure strong scaling performance of five Legion applications
  - Varying complexity (from 9-point stencil to multi-physics solver)
  - Already optimized for weak scaling performance†
  - Machine: Piz Daint (Cray XC50, Xeon E5-2690 with 12 cores & 64 GB memory per node)
- Compare with MPI references for Stencil, MiniAero, and PENNANT

† E. Slaughter, W. Lee, S. Treichler, W. Zhang, M. Bauer, G. Shipman, P. McCormick, and A. Aiken, "Control replication: Compiling implicit parallelism to efficient SPMD with logical regions," SC'17

Strong Scaling Performance







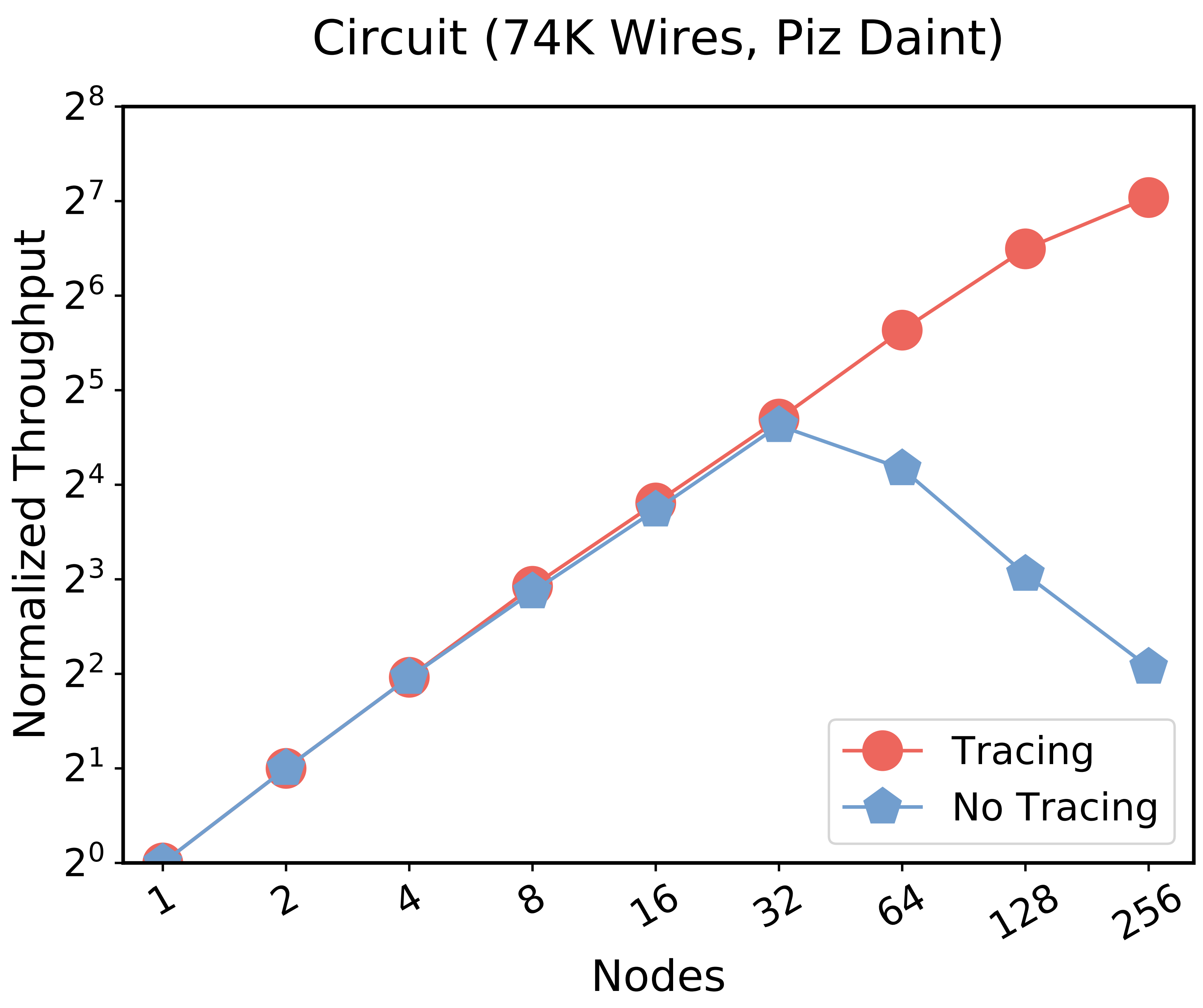
	Stencil	PENNANT	MiniAero
Trace size / node	47	121	210
Improvement	4.2X	2.8X	5.1X
DT / MPI	82%	79%	212%

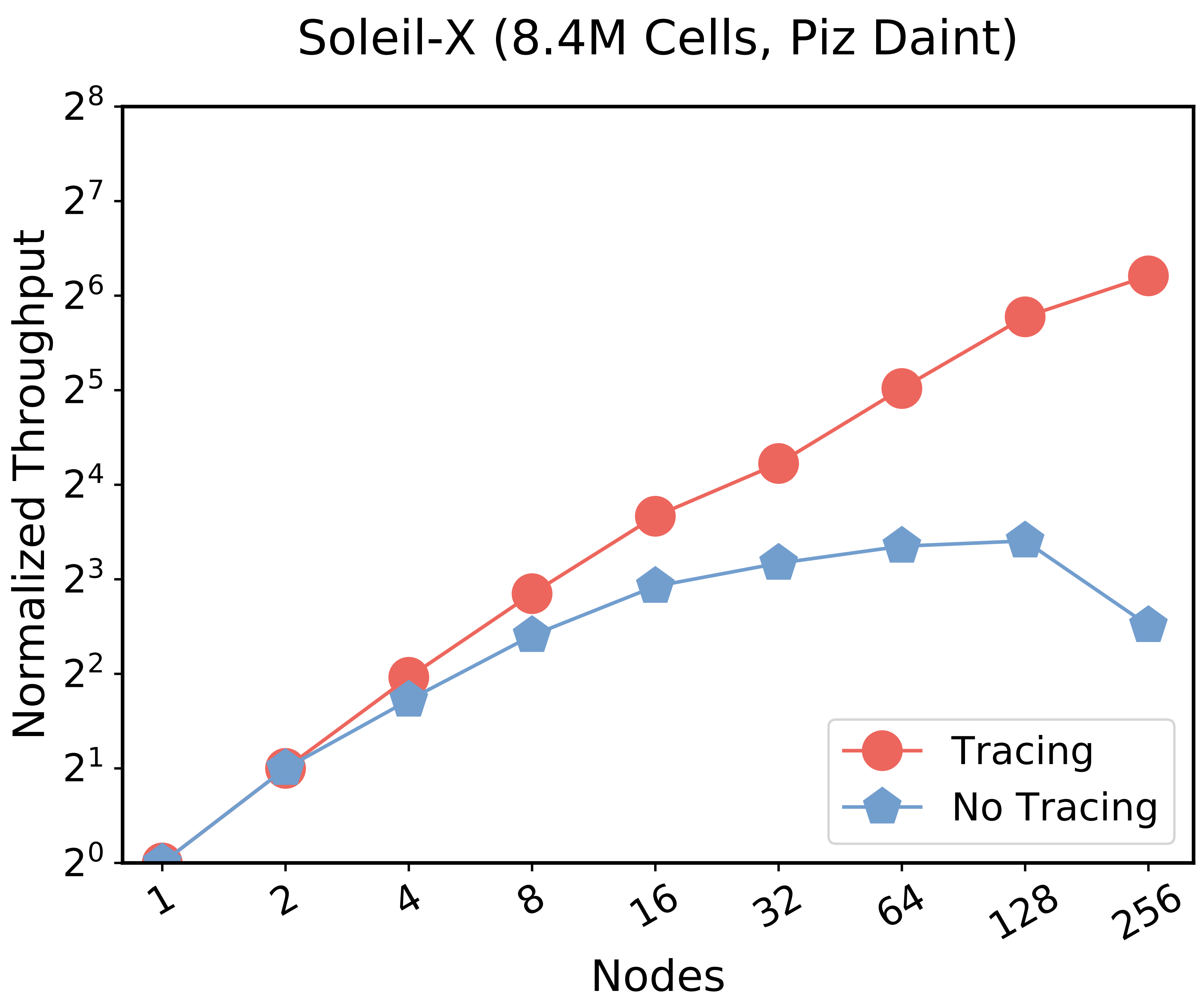
Dynamic time stepping blocks runtime analysis every iteration

Legion spares 3 cores for runtime

Legion version uses a better data layout

Strong Scaling Performance





	Circuit	Soleil-X
Trace size / node	76	344
Improvement	5.3X	7.0X

Conclusion

- **Dynamic tracing brings performance of explicit task graph construction to dynamic task-based runtimes**
  - Strong scaling performance is improved by 4.9X on average
- **Feel free to try out Dynamic Tracing!**
  - Checked in to the Legion repository: https://github.com/StanfordLegion/legion
  - Experiment scripts are here: https://gitlab.com/StanfordLegion/legion/tree/tracing-sc18

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, award DE-NA0002373-1 from the Department of Energy National Nuclear Security Administration, NSF grant CCF-1160904, an internship at NVIDIA Research, and a grant from the Swiss National Supercomputing Centre (CSCS) under project ID d80.

Questions?

Programming Model

- Traces are annotated in programs
  - Places where tracing is beneficial are often obvious
  - Finding such places is important, but an orthogonal issue

```
task T(x,y) writes(x),reads(y)
task U(x,y) reads(x), reads(y)

while (*):
  begin_trace
  T(A,B); T(C,D)
  U(A,D); U(C,B)
  end_trace
```

Optimizing Graph Calculus Commands

- Two standard optimizations: transitive reduction and copy propagation
  - The overhead is amortized by repeated replays

```
e1 := fence
e2 := op(T1(R^a), e1)
e3 := op(R^a → R^b, e2)
e4 := op(T2(R^b,S^a), e3)
e5 := merge(e2, e3, e4)
e6 := op(T3(R^a,S^a), e5)
e7 := merge(e2, e3, e4, e6)
e8 := op(Tsummary(R^a,R^b,S^a), e7)
```

```
e1 := fence
e2 := op(T1(R^a), e1)
e3 := op(R^a → R^b, e2)
e4 := op(T2(R^b,S^a), e3)
e5 := merge(e2, e3, e4)
e6 := op(T3(R^a,S^a), e4)
e7 := merge(e2, e3, e4, e6)
e8 := op(Tsummary(R^a,R^b,S^a), e6)
```

Parallel Replays

- Trace replay can be a bottleneck if the trace is long
- We can parallelize trace replay by slicing the trace

$$\texttt{e}_2 := \texttt{event};$$

$$\swarrow \quad \searrow$$

Original trace:

$\texttt{e}_2 := \texttt{op}(\texttt{A}, \texttt{e}_1);$

$\texttt{e}_3 := \texttt{op}(\texttt{B}, \texttt{e}_2);$

$\Longrightarrow$

Slice 1:

$\texttt{e}_\texttt{t} := \texttt{op}(\texttt{A}, \texttt{e}_1);$

$\texttt{trigger}(\texttt{e}_2, \texttt{e}_\texttt{t});$

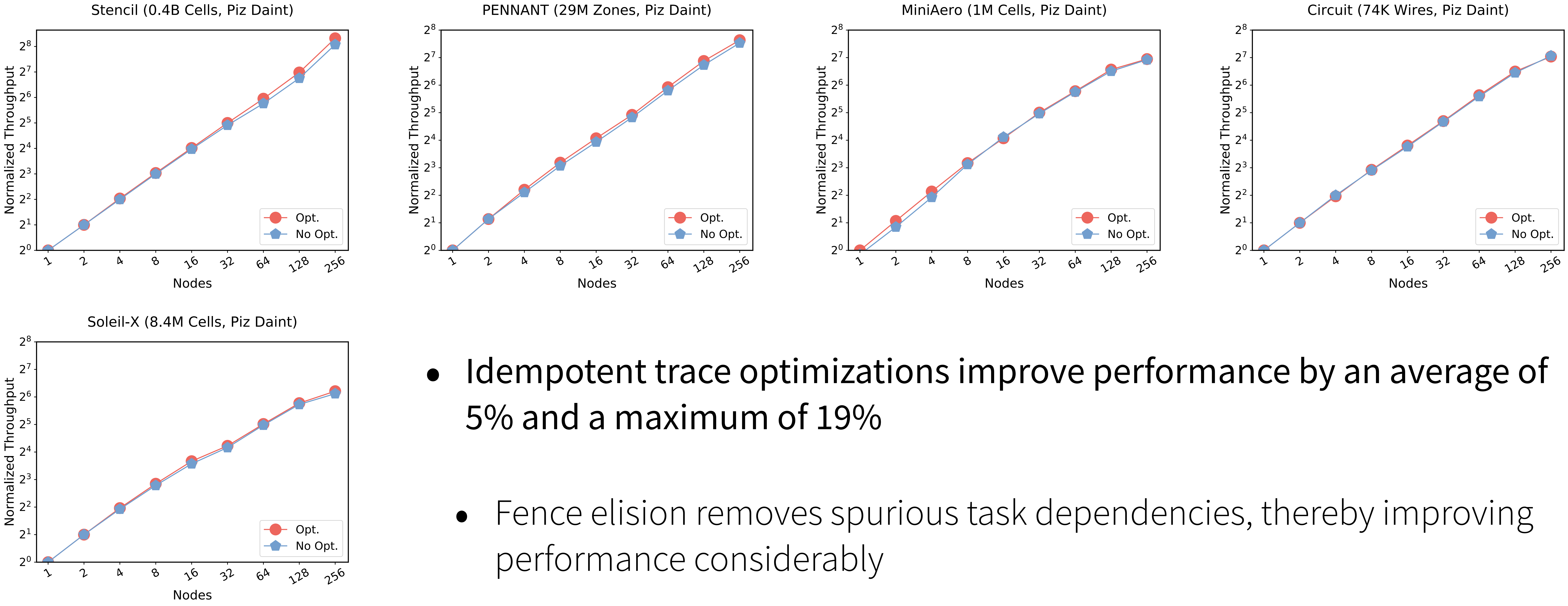
Slice 2:

$\texttt{e}_3 := \texttt{op}(\texttt{B}, \texttt{e}_2);$

$$\text{Extended graph calculus} \quad c ::= \cdots \mid e := \texttt{event} \mid \texttt{trigger}(e, e)$$

- Balanced slicing uses the implicit knowledge encoded in the application’s mapping

Effect of Idempotent Trace Optimizations



- Idempotent trace optimizations improve performance by an average of 5% and a maximum of 19%
  - Fence elision removes spurious task dependencies, thereby improving performance considerably
  - No benefit on Circuit as it has all-to-all task dependencies on each node

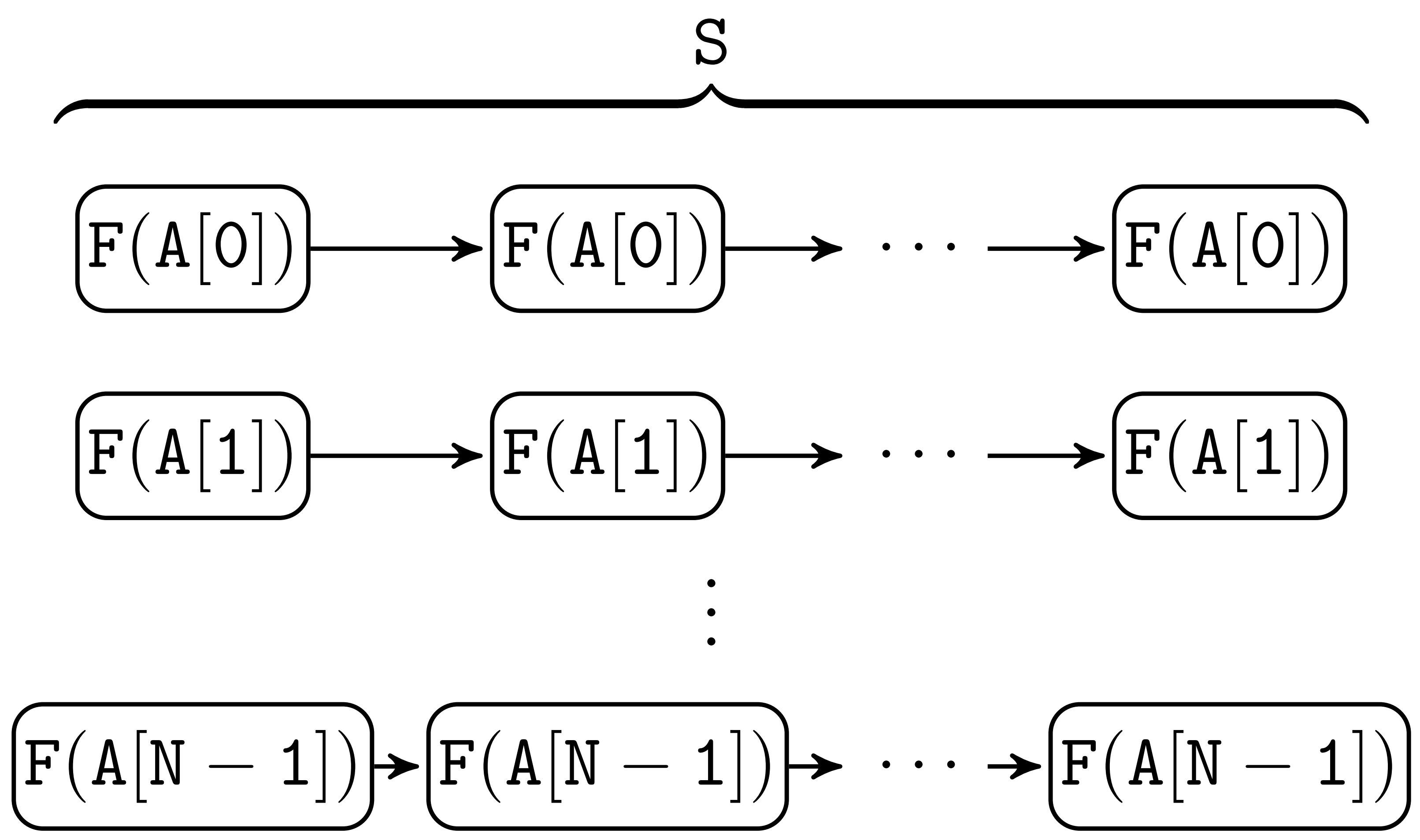
Average Task Granularity

	MiniAero		Soleil-X	
	Tr.	No Tr.	Tr.	No Tr.
Num. tasks per processor	36		56	
Min. time per iteration	6.6ms	33.8ms	23.1ms	161.2ms
Avg. task granularity	183us	940us	413us	2,879us

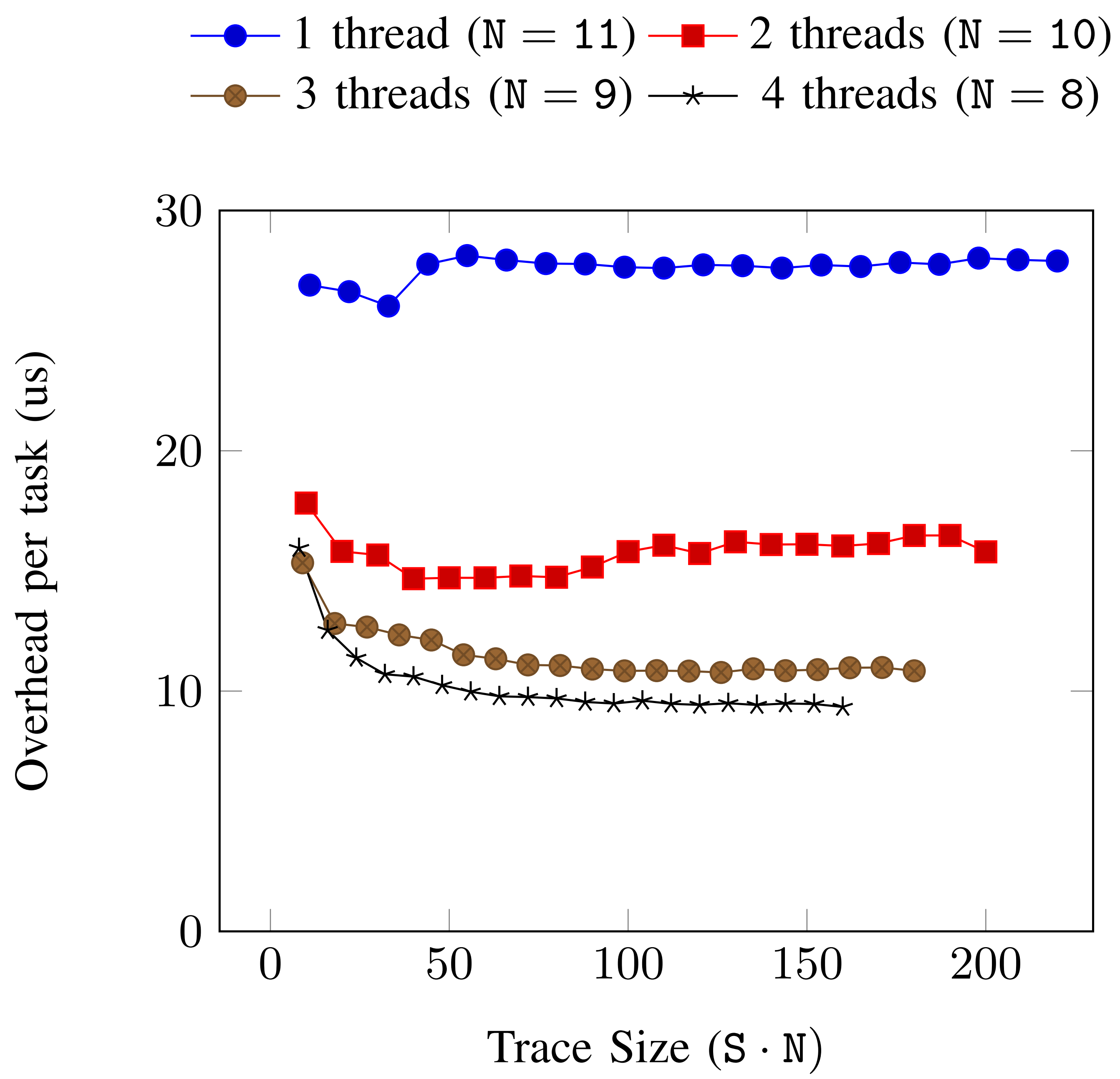
- Achieves sub-millisecond task granularity with dynamic tracing
- Soleil-X tasks take twice more steps on average per replay than MiniAero tasks

Tracing Overhead

Task graph for benchmarking



Trace replay overhead per task



- Using more runtime threads has diminishing return
- Longer traces better amortize the replay overhead

Tracing Overhead

	Stencil	Circuit	PENNANT	MiniAero	Soleil-X
No Tracing	2.23	10.29	10.47	4.99	19.41
Tracing	0.29	0.53	0.86	0.68	2.26
Improv.	7.6×	19.5×	12.2×	7.4×	8.6×
Trace size	47	76	121	210	344
Trace opt.	0.72	1.70	3.90	1.75	5.86

TABLE IV: Runtime overhead per trace (all in milliseconds)