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

Wonchan Lee
Stanford University
wonchan@cs.stanford.edu

Elliott Slaughter
SLAC National Accelerator Laboratory
slaughter@cs.stanford.edu

Michael Bauer
NVIDIA
mbauer@nvidia.com

Sean Treichler
NVIDIA
sean@nvidia.com

Todd Warszawski
Stanford University
twarszaw@cs.stanford.edu

Michael Garland
NVIDIA
mgarland@nvidia.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

***Abstract*—Many recent programming systems for both supercomputing and data center workloads generate task graphs to express computations that run on parallel and distributed machines. Due to the overhead associated with constructing these graphs the dependence analysis that generates them is often statically computed and memoized, and the resulting graph executed repeatedly at runtime. However, many applications require a dynamic dependence analysis due to data dependent behavior, but there are new challenges in capturing and re-executing task graphs at runtime. In this work, we introduce *dynamic tracing*, a technique to capture a dynamic dependence analysis of a trace that generates a task graph, and replay it. We show that an implementation of dynamic tracing improves strong scaling by an average of 4.9× and up to 7.0× on a suite of already optimized benchmarks.**

I. Introduction

Programming large distributed systems is a challenging problem in both high performance computing (HPC) and data center environments today. To address this problem, there has recently been a resurgence of programming systems that target these machines by expressing computations as task graphs [1]–[7]. The nodes in a task graph represent opaque computations and edges represent either control or data dependences constraining the execution order of the graph. Many problems in parallel programming such as mapping [1], scheduling [8], load balancing [9], fault tolerance [10], and resource allocation [1], [7] have been successfully addressed using task graphs.

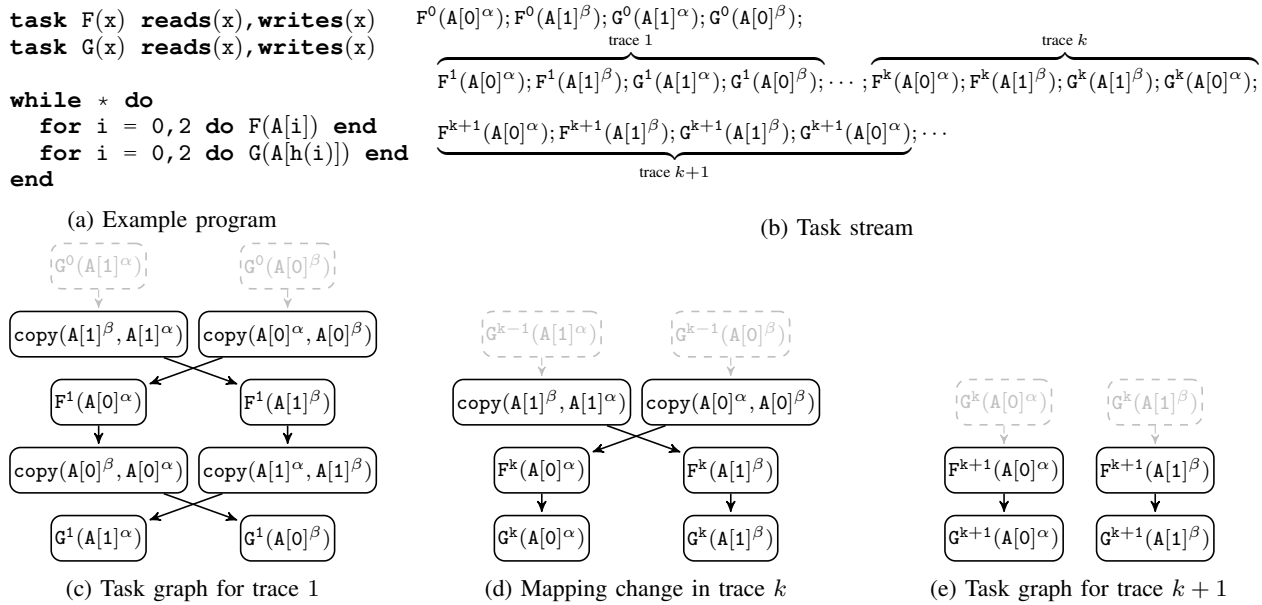
While tasks graphs are useful tools for programming systems, they can be difficult to work with directly for application developers as they can be challenging to correctly generate and compose by hand. For this reason, many systems provide higher-level abstractions and rely on an additional translation layer to perform a dependence analysis that constructs an underlying task graph [1], [3]–[5], [11]. In some cases the task graph is static and can be constructed once at the beginning of an application [5]–[7]. However, in more dynamic applications, the computation can be dependent on application data and therefore dependence analysis and the shape of the task graph must also be computed at runtime. Systems such as Legion [1], StarPU [3], and PaRSEC [4] are built around a continuously running dependence analysis that constructs a task graph on the fly, enabling them to adapt to applications with changing requirements.

While a fully dynamic dependence analysis is very flexible, it also incurs runtime overhead that can limit performance. The cost of dynamic dependence analysis can be hidden (by running the analysis in parallel with the application) only if the cost of analyzing a task is on average less than the task's execution time [1]. Therefore the cost of dynamic dependence analysis places a lower bound on the granularity of tasks that can be handled efficiently and how well applications strong scale. Consequently dynamic dependence analysis must be as efficient as possible to avoid limiting system performance.

The crucial insight of this work is that, while some applications require a fully dynamic dependence analysis, they often have *traces* of repetitive tasks for which we can memoize the results of the dynamic dependence analysis and therefore reduce the overhead of executing tasks in a trace. While similar in spirit to trace-based JIT-compilation systems [12]–[17], specializing dynamic dependence analysis for parallel and distributed systems raises new correctness and performance issues, because unlike programming systems for shared-memory machines, distributed task graphs must express both parallelism and the *coherence* of data. For example, dynamic dependence analyses for distributed systems can generate different subgraphs for the same trace based on the location(s) of the most recent version of data. Therefore every replay of a specialized trace needs to maintain the coherence of data, an issue that does not arise in shared-memory environments where data coherence is maintained by the underlying hardware.

To address these issues, we present *dynamic tracing*, a technique to efficiently and correctly memoize a dynamic dependence analysis and generate a task graph semantically equivalent to (but also often syntactically different from) the original. Dynamic tracing achieves this goal in three steps. First, it records the analysis of a trace as a sequence of *graph calculus* commands; graph calculus is a simple imperative language with commands that directly construct task graphs. The recorded graph calculus commands are associated with a *precondition* that must be satisfied for the commands to correctly replay

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00



(a) Example program

(b) Task stream

(c) Task graph for trace 1

(d) Mapping change in trace k

(e) Task graph for trace $k+1$

Fig. 1: Example program and task graphs for traces

the task graph, and a *postcondition* that must be applied to make the dependence analysis state consistent with the replayed graph. Second, it optimizes the commands to minimize the cost of replay and eliminate unnecessary synchronizations in the replayed subgraph. Third, whenever a previously recorded trace appears during program execution, the recorded graph calculus commands are replayed to replace the dependence analysis as long as the trace's precondition is satisfied.

This paper makes three contributions:

- To the best of our knowledge, dynamic tracing is the first technique to just-in-time specialize task graphs in distributed task-based runtimes with dynamic dependence analysis. We present a complete design of dynamic tracing with several key optimizations.
- We describe an implementation of dynamic tracing embedded in the Legion runtime system.
- For five already optimized applications, we demonstrate that dynamic tracing improves strong scaling performance up to 7.0×, and by 4.9× on average, when running on up to 256 nodes.

The rest of this paper is organized as follows. In Section II, we give an informal overview of dynamic tracing. Section III describes our programming model and defines basic concepts. Then, we present dynamic tracing in Section IV. Section V discusses the implementation of dynamic tracing in Legion and Section VI presents experiment results. We survey related work in Section VII and conclude in Section VIII.

II. Overview

We briefly motivate the need for dynamic tracing with a small example designed to illustrate the salient issues. The program in Figure 1a issues four tasks $\texttt{F}(\texttt{A}[0])$, $\texttt{F}(\texttt{A}[1])$, $\texttt{G}(\texttt{A}[\texttt{h}(0)])$, $\texttt{G}(\texttt{A}[\texttt{h}(1)])$ for every iteration of the while loop. Static dependence analyses will give imprecise results because of the indices computed using the opaque function h. In contrast, precise dynamic dependence analysis is straightforward. For example, if $\texttt{h}(0) = 1$ and $\texttt{h}(1) = 0$, dynamic dependence analysis shows there are dependences between $\texttt{F}(\texttt{A}[0])$ and $\texttt{G}(\texttt{A}[\texttt{h}(1)])$, and between $\texttt{F}(\texttt{A}[1])$ and $\texttt{G}(\texttt{A}[\texttt{h}(0)])$.

In a distributed system, data dependences may require data movement. For example, if $\texttt{F}(\texttt{A}[0])$ and $\texttt{G}(\texttt{A}[0])$ execute on different nodes of the machine, and since $\texttt{F}(\texttt{A}[0])$ writes to $\texttt{A}[0]$, the updated value of $\texttt{A}[0]$ must be copied to the node where $\texttt{G}(\texttt{A}[0])$ will run. We use node identifiers α, β, etc. as superscripts to data elements to distinguish different *instances* of the same data on different nodes. In Figure 1c, for example, the upper left operation copies an instance of $\texttt{A}[1]$ on node α to an instance of A[1] on node β. Tasks execute on the node where their arguments are placed.

Figure 1b shows some *traces*, which are sequences of tasks issued by the program. The dependence analysis of trace 1, which corresponds to the while loop's second iteration, generates the task graph in Figure 1c. To ensure correctness, copy operations are added to the task graph where necessary. During task graph generation, dynamic tracing memoizes the task graph using graph calculus commands (discussed in Section IV-B) so the graph can be regenerated for later executions of the same trace.

Dynamic tracing detects when recorded commands can be reused using two criteria. First, the subsequent trace must be exactly the same as the one from the second iteration; this requires that the tasks have the same data dependences and the choice of data placement is the same so that the set of required copies is the same. Second, the set of instances that hold the most recent version of input data to the trace must be the same; in this example $\texttt{A}[1]^{\alpha}$ and $\texttt{A}[0]^{\beta}$ are the input data during trace 1 when the graph is captured. For this example, dynamic tracing can replace the dynamic dependence analysis

for each trace from iteration 2 to $k-1$ using the graph calculus commands captured during trace 1.

Suppose now that during trace k the choices of nodes for the data of tasks `G(A[h(0)])` and `G(A[h(1)])` are swapped as shown in Figure 1b. (While this change of data placement is not a realistic scenario, it illustrates the issues that arise in real applications.) Trace k then looks different from the first $k-1$ traces because the location of the instances for tasks `G(A[h(0)])` and `G(A[h(1)])` have changed, leading dynamic tracing to reject replaying the capture of trace 1 and instead capture a new trace. However, there is an important subtlety that occurs when dynamic tracing encounters trace $k+1$. Dynamic tracing cannot replay the commands from trace k for trace $k+1$ because the location of the input instances are different; trace k has input instances $\texttt{A[1]}^{\alpha}$ and $\texttt{A[0]}^{\beta}$ while trace $k+1$ has input instances $\texttt{A[1]}^{\beta}$ and $\texttt{A[0]}^{\alpha}$ necessitating input copy operations. Therefore dynamic tracing will also need to capture commands for trace $k+1$ and will be able to replay them starting with trace $k+2$, assuming the instance placement remains stable.

III. Programming Model

We consider a task-based programming model where a program is decomposed into *tasks*. A task is a unit of computation that runs to completion once scheduled on a processor. Tasks store data in *regions*; a region is simply a named collection of data used by a task. For dynamic tracing the concept of a region is flexible and can be used to name any arbitrary collection of data including, but not limited to, opaque serialized data, an array, an arena, a relation, etc. The example in Figure 1a has four regions: `A[0]`, `A[1]`, `B[0]`, and `B[1]`. Tasks declare *permissions* on regions (line 1–2 in Figure 1a), which describe how they access data. For simplicity, we consider only *read* and *write* permissions, though some systems [1], [18] provide a reduction permission for updates with commutative and associative operators.

A region can be represented by multiple *region instances* in different memories. In Figure 1a, region `A[0]` has two region instances $\texttt{A[0]}^{\alpha}$ and $\texttt{A[0]}^{\beta}$.

When a program is executed, it makes a sequence of task calls, each of which goes through a standard pipeline of phases [1], [4], [19]. First, regions are *mapped* to region instances (assigned to physical memories). An invocation of a task whose regions are mapped is a *task instance*. The mapping does not change during execution of a task instance, but can be different in different task instances of the same task. In Figure 1b, region `A[1]` for task `G(A[1])` is mapped to $\texttt{A[1]}^{\alpha}$ in trace $k-1$, whereas it is mapped to $\texttt{A[1]}^{\beta}$ in trace k. The *mapper* is the pipeline stage that makes mapping decisions for tasks according to some (possibly dynamic) policy.

The next stage in processing a task is *dependence analysis*. Two task instances have a *dependence* when they access the same region instance and at least one of them has write permissions on the region. In Figure 1e, task instances $\texttt{F(A[0]}^{\alpha}\texttt{)}$ and $\texttt{G(A[0]}^{\alpha}\texttt{)}$ are dependent because both write to the same region instance $\texttt{A[0]}^{\alpha}$, while $\texttt{F(A[0]}^{\alpha}\texttt{)}$ and $\texttt{F(A[1]}^{\beta}\texttt{)}$ are independent, and thus can run in parallel, as they write to two different region instances.

Any access to a region in a task must be *coherent*. If a task instance updates a region instance, any subsequent task instances reading region instances of the same region must see the update. In our model maintaining coherence is the responsibility of the system. The program specifies what data is to be used, and the programming system manages coherence by automatically generating copies and inserting synchronization to ensure the data is current when and where it is needed.

Once dependence analysis is complete for a task instance, the task instance and any required copies are inserted into the *task graph*, a DAG where nodes are *operations* (task instances and copies) and edges are dependences between operations. The runtime's execution of the graph is concurrent with the graph generation. The runtime finds operations that have no predecessors in the task graph, and schedules their execution on processors. The mapper's choice of instances for regions constrain a task instance to only run on processors that are able to directly access those instances.

We assume that traces are explicitly delimited in a program. A trace is a sequence of task instances that are issued between a **`begin_trace`** and a matching **`end_trace`** statement. At least some of the places that tracing can be beneficial are obvious, such as around important loops. Consider the following example from Figure 1a, which delimits all traces in Figure 1b:

```
4  while * do
5    begin_trace
6    for i = 0,2 do F(A[i]) end
7    for i = 0,2 do G(A[h(i)]) end
8    end_trace
9  end
```

IV. Dynamic Tracing

In this section, we describe dynamic tracing, a technique to JIT specialize dependence analysis for traces. Dependence analysis "interprets" a trace to generate a task graph. For each task instance in the trace, this interpreter analyzes its dependences on previous task instances and updates the graph. If instead our goal is to build a specific graph, we can specialize the interpreter's analysis to a process that builds just that one graph. Dynamic tracing achieves this specialization by recording the dependence analysis of a trace and replaying it whenever the recorded trace appears again during execution to replace the dependence analysis.

A. Baseline Dependence Analysis

Dynamic tracing can specialize any *correct* dependence analysis that generates a task graph as its result. A correct dependence analysis satisfies the following two conditions.

First, a task graph from a correct dependence analysis of tasks captures all dependences between them. Specifically, if task instances T_1 and T_2 are dependent and T_2 is issued after T_1, there must be at least one path from T_1 to T_2 in the task graph. However, task graphs may have edges for *transitive* dependences, i.e, dependences that are transitively expressed

Tasks: **task** $\mathtt{T_1}(\mathtt{R},\mathtt{S})$ **reads writes**$(\mathtt{R})$,**writes**$(\mathtt{S})$
task $\mathtt{T_2}(\mathtt{R},\mathtt{S})$ **reads**$(\mathtt{R})$,**reads**$(\mathtt{S})$
task $\mathtt{T_3}(\mathtt{R},\mathtt{S})$ **writes**$(\mathtt{R})$,**reads**$(\mathtt{S})$

Task Instance	Valid Instances	Task Graph
$\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$	$\mathtt{R} \mapsto \mathtt{R}^\alpha$ (rule $\mathbf{R_1}$,**W**) $\mathtt{S} \mapsto \mathtt{S}^\alpha$ (rule **W**)	$\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$
$\mathtt{T_2}(\mathtt{R}^\beta,\mathtt{S}^\alpha)$	$\mathtt{R} \mapsto \{\mathtt{R}^\beta,\mathtt{R}^\alpha\}$ (rule $\mathbf{R_2}$) $\mathtt{S} \mapsto \mathtt{S}^\alpha$ (rule $\mathbf{R_1}$)	$\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$ → $\mathtt{copy_1}(\mathtt{R}^\beta,\mathtt{R}^\alpha)$ → $\mathtt{T_2}(\mathtt{R}^\beta,\mathtt{S}^\alpha)$
$\mathtt{T_3}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$	$\mathtt{R} \mapsto \mathtt{R}^\alpha$ (rule **W**) $\mathtt{S} \mapsto \mathtt{S}^\alpha$ (rule $\mathbf{R_1}$)	$\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$ → $\mathtt{T_3}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$; $\mathtt{copy_1}(\mathtt{R}^\beta,\mathtt{R}^\alpha)$ → $\mathtt{T_2}(\mathtt{R}^\beta,\mathtt{S}^\alpha)$

Fig. 2: Dependence analysis of a trace

by other dependences. For example, the task graph is permitted to have an edge between task instances T_1 and T_3 even if it already has edges between T_1 and T_2, and between T_2 and T_3. Transitive dependences are not harmful for parallelism, because they impose no additional constraints, and dependence analysis algorithms often include them in the result as additional analysis would be required to remove them.

Second, a correct dependence analysis guarantees coherence. Each region instance used in a task instance must be *valid*, i.e., containing the latest updates to the region. We assume that a correct dependence analysis algorithm keeps track of valid instances of each region using the following rules:

- **$\mathbf{R_1}$** If the task has read permission on the region and the region instance is already valid, no data movement is necessary for coherent read access.
- **$\mathbf{R_2}$** If the task has read permission on the region and the region instance r is not valid, then r is made valid by issuing a copy from an existing valid instance to r. The issued copy is treated as a task that reads from the source instance and writes to r, except it does not invalidate valid instances.
- **W** If the task has write permission on the region, all other valid instances are invalidated and the written region instance becomes the sole valid instance of the region.

Figure 2 illustrates one possible dependence analysis of three task instances. For each task instance, the figure shows changes in the list of valid instances and the resulting task graph. The line under each valid instance denotes the coherence rule applied. Note that task graphs contain edges for transitive dependences between $\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$ and $\mathtt{T_2}(\mathtt{R}^\beta,\mathtt{S}^\alpha)$, and between $\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$ and $\mathtt{T_3}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$.

B. Recording Dependence Analysis

Dynamic tracing starts with the *recorder* recording the dependence analysis of a trace. A recording for a trace is initiated in two cases: when a trace has appeared for the first time, or when no recording of a trace passes the precondition check described in Section IV-D.

$$
\begin{aligned}
&T \in \mathit{TaskId} \quad I \in \mathit{RegionInstance}\\
&e \in \mathit{Event} \quad op \in \mathit{Operation} \quad c \in \mathit{Command}\\
&c ::= e := \mathtt{op}(op, e) \mid e := \mathtt{merge}(\overline{e}) \mid e := \mathtt{fence} \mid c; c\\
&op ::= T(\overline{I}) \mid \mathtt{copy}(I, I) \mid \cdots
\end{aligned}
$$

Fig. 3: Syntax of graph calculus

The recorder uses *graph calculus*, whose syntax is shown in Figure 3, to express task graphs. Graph calculus uses *events* that signal the termination of operations. An op command has the form $e_2 := \mathtt{op}(o, e_1)$. The operation o begins execution after the event e_1 triggers, and the event e_2 triggers when o terminates. To express multiple predecessors for an operation, the merge command merges a set of events into an event that is triggered when the events being merged are all triggered. A fence command creates a *fence*, an operation that finishes only after all preceding operations terminate. Fences allow graph calculus commands to work correctly with earlier untraced parts of the execution, as the previous dependent operations potentially include operations not in the trace. Finally, the calculus has command sequencing.

The recorder generates graph calculus commands from a dependence analysis of a trace as follows. Each trace operation o has a corresponding command $e_2 := \mathtt{op}(o, e_1)$. The termination event e_2 is unique (is not used on the left-hand side of any other op command). The event e_1 is the merge (using a merge command) of the termination events of o's dependence predecessors in the trace. For example, in Figure 4, task instance $\mathtt{T_2}(\mathtt{R}^\beta,\mathtt{S}^\alpha)$ has two predecessors $\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$ and $\mathtt{copy}(\mathtt{R}^\beta,\mathtt{R}^\alpha)$, whose events $\mathtt{e_2}$ and $\mathtt{e_3}$ are merged into $\mathtt{e_4}$. If there is no predecessor (e.g., because this is the first operation of the trace), a fence is introduced to safely capture any dependences on those operations that are not recorded. Task instance $\mathtt{T_1}(\mathtt{R}^\alpha,\mathtt{S}^\alpha)$ in Figure 4 uses fence $\mathtt{e_1}$ as it has no predecessor in the trace.

When the recorder reaches the end of the trace, the recorder inserts an op statement for a *summary* operation, a task instance that writes to all region instances used in the trace. The key difference between a fence and a summary operation is that a fence waits on all the preceding operations, both within and out of the current trace, whereas the summary operation has dependences only on operations within the trace. Any subsequent operation that has dependences on any of the replayed operations can safely catch the dependences transitively through the summary operation.

The recorder also computes the *precondition* and *postcondition* of recorded commands, which are used in the replaying stage; the precondition is a set of region instances that must be valid for recorded commands to replay the same subgraph as the original dependence analysis; the postcondition is a set of region instances that become valid after recorded commands replay a subgraph. The precondition and postcondition are computed by processing trace operations in order, beginning with empty pre and postconditions, and applying the following rules:

- If rule $\mathbf{R_1}$ was applied to the region instance and the

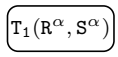
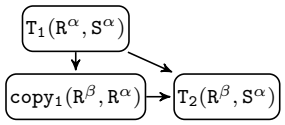
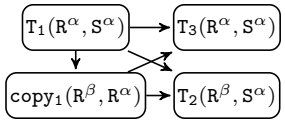
Task Instance	Task Graph	Recorder State	Recorded Commands
$\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$		Events: $\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha) \mapsto \mathtt{e}_2$ Preconditions: $\mathtt{R} \mapsto \mathtt{R}^\alpha$ Postconditions: $\mathtt{R} \mapsto \mathtt{R}^\alpha, \mathtt{S} \mapsto \mathtt{S}^\alpha$ Region Instances: $\mathtt{R}^\alpha, \mathtt{S}^\alpha$	$\mathtt{e}_1 := \mathtt{fence};$ $\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha), \mathtt{e}_1);$
$\mathtt{T}_2(\mathtt{R}^\beta, \mathtt{S}^\alpha)$		Events: $\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha) \mapsto \mathtt{e}_2$ $\mathtt{copy}_1(\mathtt{R}^\beta, \mathtt{R}^\alpha) \mapsto \mathtt{e}_3$ $\mathtt{T}_2(\mathtt{R}^\beta, \mathtt{S}^\alpha) \mapsto \mathtt{e}_5$ Preconditions: $\mathtt{R} \mapsto \mathtt{R}^\alpha$ Postconditions: $\mathtt{R} \mapsto \{\mathtt{R}^\beta, \mathtt{R}^\alpha\}, \mathtt{S} \mapsto \mathtt{S}^\alpha$ Region Instances: $\mathtt{R}^\beta, \mathtt{R}^\alpha, \mathtt{S}^\alpha$	$\mathtt{e}_3 := \mathtt{op}(\mathtt{copy}(\mathtt{R}^\beta, \mathtt{R}^\alpha), \mathtt{e}_2);$ $\mathtt{e}_4 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3);$ $\mathtt{e}_5 := \mathtt{op}(\mathtt{T}_2(\mathtt{R}^\beta, \mathtt{S}^\alpha), \mathtt{e}_4);$
$\mathtt{T}_3(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$		Events: $\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha) \mapsto \mathtt{e}_2$ $\mathtt{copy}_1(\mathtt{R}^\beta, \mathtt{R}^\alpha) \mapsto \mathtt{e}_3$ $\mathtt{T}_2(\mathtt{R}^\beta, \mathtt{S}^\alpha) \mapsto \mathtt{e}_5$ $\mathtt{T}_3(\mathtt{R}^\alpha, \mathtt{S}^\alpha) \mapsto \mathtt{e}_7$ Preconditions: $\mathtt{R} \mapsto \mathtt{R}^\alpha$ Postconditions: $\mathtt{R} \mapsto \mathtt{R}^\alpha, \mathtt{S} \mapsto \mathtt{S}^\alpha$ Region Instances: $\mathtt{R}^\beta, \mathtt{R}^\alpha, \mathtt{S}^\alpha$	$\mathtt{e}_6 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3);$ $\mathtt{e}_7 := \mathtt{op}(\mathtt{T}_3(\mathtt{R}^\alpha, \mathtt{S}^\alpha), \mathtt{e}_6);$
(End of the trace)		Insert a summary operation:	$\mathtt{e}_8 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_5, \mathtt{e}_7);$ $\mathtt{e}_9 := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}(\mathtt{R}^\beta, \mathtt{R}^\alpha, \mathtt{S}^\alpha), \mathtt{e}_8);$

Fig. 4: Recording of the dependence analysis in Figure 2

Original	Dataflow Analysis	Transitive Reduction	Copy Propagation
$\mathtt{e}_1 := \mathtt{fence};$		$\mathtt{e}_1 := \mathtt{fence};$	$\mathtt{e}_1 := \mathtt{fence};$
$\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_1, \mathtt{e}_1);$	$\mathtt{e}_2 \mapsto \mathtt{e}_1$	$\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_1, \mathtt{e}_1);$	$\mathtt{e}_2 := \mathtt{op}(\mathtt{T}_1, \mathtt{e}_1);$
$\mathtt{e}_3 := \mathtt{op}(\mathtt{copy}, \mathtt{e}_2);$	$\mathtt{e}_3 \mapsto \mathtt{e}_1, \mathtt{e}_2$	$\mathtt{e}_3 := \mathtt{op}(\mathtt{copy}, \mathtt{e}_2);$	$\mathtt{e}_3 := \mathtt{op}(\mathtt{copy}, \mathtt{e}_2);$
$\mathtt{e}_4 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3);$	$\mathtt{e}_4 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3$	$\mathtt{e}_4 := \mathtt{merge}(\mathtt{e}_3);$	
$\mathtt{e}_5 := \mathtt{op}(\mathtt{T}_2, \mathtt{e}_4);$	$\mathtt{e}_5 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4$	$\mathtt{e}_5 := \mathtt{op}(\mathtt{T}_2, \mathtt{e}_4);$	$\mathtt{e}_5 := \mathtt{op}(\mathtt{T}_2, \mathtt{e}_3);$
$\mathtt{e}_6 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3);$	$\mathtt{e}_6 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3$	$\mathtt{e}_6 := \mathtt{merge}(\mathtt{e}_3);$	
$\mathtt{e}_7 := \mathtt{op}(\mathtt{T}_3, \mathtt{e}_6);$	$\mathtt{e}_7 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_6$	$\mathtt{e}_7 := \mathtt{op}(\mathtt{T}_3, \mathtt{e}_6);$	$\mathtt{e}_7 := \mathtt{op}(\mathtt{T}_3, \mathtt{e}_3);$
$\mathtt{e}_8 := \mathtt{merge}(\mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_5, \mathtt{e}_7);$	$\mathtt{e}_8 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5, \mathtt{e}_6, \mathtt{e}_7$	$\mathtt{e}_8 := \mathtt{merge}(\mathtt{e}_5, \mathtt{e}_7);$	$\mathtt{e}_8 := \mathtt{merge}(\mathtt{e}_5, \mathtt{e}_7);$
$\mathtt{e}_9 := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}, \mathtt{e}_8);$	$\mathtt{e}_9 \mapsto \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5, \mathtt{e}_6, \mathtt{e}_7, \mathtt{e}_8$	$\mathtt{e}_9 := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}, \mathtt{e}_8);$	$\mathtt{e}_9 := \mathtt{op}(\mathtt{T}_{\mathtt{summary}}, \mathtt{e}_8);$

Fig. 5: Optimizations on the commands in Figure 4 (region instances in operations are elided.)

region instance is not in the postcondition, that region instance is added to the pre and postcondition.

- If rule $\mathbf{R_2}$ was applied to the region instance and the source instance of the copy is not in the postcondition, that source instance is added to the pre and postcondition. The target instance of the copy is added to the postcondition.
- If rule $\mathbf{W}$ was applied to the region instance, the postcondition of that region is cleared and that region instance is added to the postcondition.

C. Optimizing Graph Calculus Commands

After a trace is recorded, and before it can be used, we apply two standard compiler passes to optimize the trace: transitive reduction and copy propagation.

Transitive reduction optimizes graph calculus commands by removing transitive dependences. We run a dataflow analysis that discovers all transitive predecessors for each event and then, among the events being merged by each merge command, we remove those that are transitive predecessors of any other event. In Figure 5, event $\mathtt{e}_2$ is removed in the first and second merge commands because it is a transitive predecessor of event $\mathtt{e}_3$, and $\mathtt{e}_2$ and $\mathtt{e}_3$ are removed from the third merge command because they are transitive predecessors of $\mathtt{e}_5$. Removing transitive dependences reduces the cost of replaying the graph.

Transitive reductions sometimes leave only a single event in a merge command, which is equivalent to a copy assignment. We run copy propagation to eliminate those unnecessary copies. For example, in Figure 5, the merge command $\mathtt{e}_4 := \mathtt{merge}(\mathtt{e}_3)$ is removed and all occurrences of event $\mathtt{e}_4$ are replaced by $\mathtt{e}_3$.

D. Replaying Dependence Analysis

The next component of dynamic tracing is to replay dependence analysis for a trace. Figure 6 illustrates how the *replayer* replays dependence analysis for the second appearance of trace $\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha); \mathtt{T}_2(\mathtt{R}^\beta, \mathtt{S}^\alpha); \mathtt{T}_3(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$ using a recording from the first appearance of the trace. First, the replayer checks that each region instance in the precondition is currently valid (Step 1). If any region instance in the precondition is not valid, the replayer cannot reuse recorded commands, because the original dependence analysis of the trace would issue a copy to make that region instance valid, which is not replayed by the commands. If all recordings fail to pass the precondition check, the replayer stops the current replay and the recorder starts a new recording session. Otherwise, the replayer proceeds with a recording whose precondition is satisfied. In Figure 6, the set of valid instances after task instance $\mathtt{T}_2(\mathtt{R}^\gamma, \mathtt{S}^\alpha)$ is analyzed subsumes the precondition and therefore the recording can be replayed.

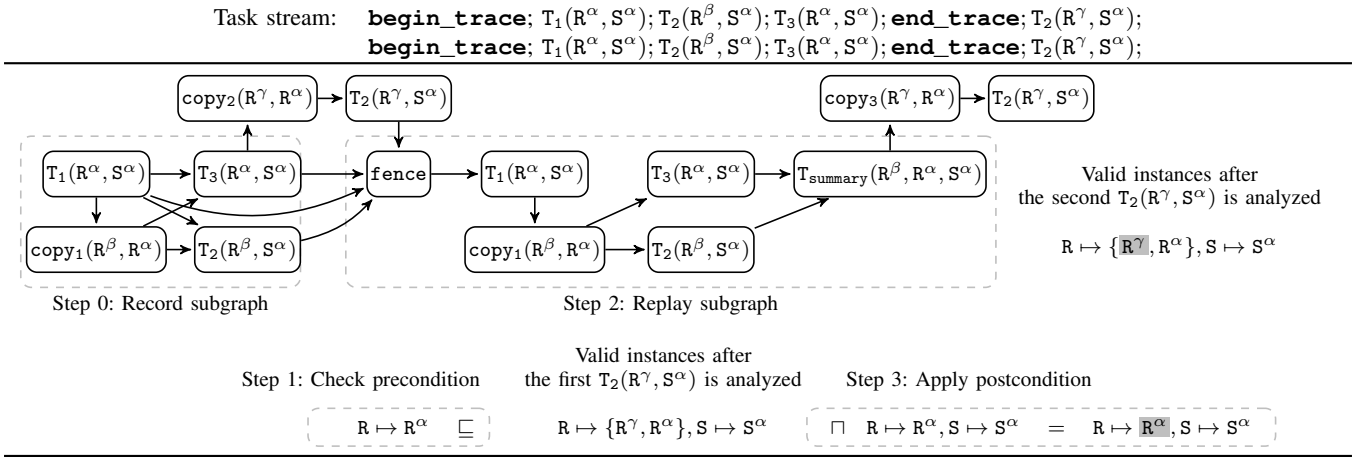


Fig. 6: Replay of dependence analysis using the recording in Figure 4

Next, the replayer runs recorded commands to reconstruct a subgraph (Step 2). Any explicitly parallel runtime system that supports a synchronization primitive such as an event or stream that can be used to express dependences between tasks and data movement operations can implement graph calculus. Many common runtime APIs support the requirements for graph calculus. For example, both CUDA [20] and OpenCL [21] can support graph calculus via their use of streams and events respectively to mediate dependences between kernels and copy operations. Furthermore, for distributed memory cases, systems like Realm [2] and OCR [22] have event primitives that can be used on any node to handle distributed execution of graph calculus commands for computation and data movement.

When replaying a trace, graph calculus commands execute sequentially to construct a subgraph equivalent to the one produced by the original dependence analysis. The semantics of graph calculus commands is straightforward, except for the fence command. A fence command creates a new fence with dependences on all operations that use any region instance used by commands in the trace. However, the fence is not connected to operations that do not access any region instances used in the trace. This is to prevent those operations, which are independent of the replayed subgraph, from being unnecessarily blocked by that fence. In Figure 6, all users of region instances $\mathtt{R}^\alpha$, $\mathtt{R}^\beta$, and $\mathtt{S}^\alpha$, which are the ones used in the recorded commands, are connected to the new fence `fence`. Note that the replayed subgraph does not contain transitive dependences between $\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$ and $\mathtt{T}_2(\mathtt{R}^\beta, \mathtt{S}^\alpha)$, and between $\mathtt{T}_1(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$ and $\mathtt{T}_3(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$, unlike the subgraph for the first trace, due to the optimizations in Section IV-C.

Finally, the replayer updates the list of valid instances using the postcondition (Step 3). The known valid instances after a replay of a subgraph may be incorrect because the replayed commands are not analyzed again by dependence analysis. The replayer ensures the system has the correct set of valid instances after replay by tagging region instances in the postcondition as valid and invalidating all other instances. In Figure 6, region instance $\mathtt{R}^\gamma$ is invalidated after the replay.

Before restarting dependence analysis, the replayer reinitializes the dependence analysis state using the summary operation. This makes the dependence analysis aware of the net effect of the replayed operations; any subsequent operation can catch its dependences on any of the replayed operations transitively through this summary operation. For example, the dependence between task instance $\mathtt{T}_3(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$ in the replayed graph and the subsequent copy $\mathtt{copy}_3(\mathtt{R}^\gamma, \mathtt{R}^\alpha)$ is captured by those between $\mathtt{T}_3(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$ and the summary operation $\mathtt{T}_{\mathtt{summary}}(\mathtt{R}^\beta, \mathtt{R}^\alpha, \mathtt{S}^\alpha)$, and between $\mathtt{T}_{\mathtt{summary}}(\mathtt{R}^\beta, \mathtt{R}^\alpha, \mathtt{S}^\alpha)$ and $\mathtt{copy}_3(\mathtt{R}^\gamma, \mathtt{R}^\alpha)$.

Algorithm 1 shows the complete dynamic tracing algorithm. The algorithm has two modes: analysis mode (DEP) and tracing mode (TRACE). If it is in analysis mode, the algorithm maps each task call to a task instance that goes through the normal dependence analysis. Otherwise, the algorithm builds a trace of task instances until it hits the end of that trace (line 11), and it either records or replays the trace (`RecordOrReplay`), based on the criteria described in this section. The algorithm changes from analysis mode to tracing mode when it sees the beginning of a trace (line 9), and from tracing mode to analysis mode once it finishes either a recording or a replay (line 13).

E. Optimizing Replay Using Idempotent Recordings

Recognizing *idempotent recordings* is crucial to providing an optimized implementation of dynamic tracing. A recording of a trace is idempotent when its postcondition implies its precondition. For example, the recording in Figure 4 is idempotent as its postcondition $\mathtt{R} \mapsto \mathtt{R}^\alpha, \mathtt{S} \mapsto \mathtt{S}^\alpha$ contains its precondition $\mathtt{R} \mapsto \mathtt{R}^\alpha$.

The most important property of idempotent recordings is that once an idempotent recording is replayed for a trace, it becomes replayable without having to apply its postcondition and check its precondition again for another replay that immediately follows. In other words, a list of valid instances that satisfies the precondition of an idempotent recording once will still satisfy that precondition no matter how many times the recording is replayed. This allows two further optimizations:

- Once the precondition of an idempotent recording passes, the algorithm never checks the precondition for future consecutive replays of the same trace.

- The algorithm can delay applying the postcondition of an idempotent tracing until it gets a different trace or a task that is not in any trace.

Algorithm 2 shows a modified algorithm to incorporate these optimizations. There are several differences in Algorithm 2 from Algorithm 1. First, Algorithm 2 keeps the previous trace and recording to check that the same trace is repeatedly replayed (line 30–31). Next, it replays a recording without any check when it realizes it is replaying an idempotent recording repeatedly (line 18–19). Finally, it applies the pending postcondition in cases when the current trace is different from the previous one (line 21–22) or when the task does not belong to any trace (line 5–6).

F. Fence Elision

Another important optimization that idempotent recordings allow is *fence elision*. Although the fence and the summary operation safely connect a subgraph replayed by graph calculus commands to that generated by dependence analysis and vice versa, they may introduce spurious dependences between operations because they are a join point in the task graph. For example, in Figure 7c, the summary operation $\mathtt{T_s}(\mathtt{R}^\alpha, \mathtt{S}^\alpha)$ and the fence `fence` add spurious dependences between the first $\mathtt{B}(\mathtt{R}^\alpha)$ and the second $\mathtt{A}(\mathtt{S}^\alpha)$, and between the first $\mathtt{B}(\mathtt{S}^\alpha)$ and the second $\mathtt{A}(\mathtt{R}^\alpha)$. In case of repeatedly replaying the same trace with an idempotent recording, the replayer can keep appending the subgraph from each replay without needing to issue a fence and register the summary operation as these replays do not require precondition checks.

Figure 7 illustrates fence elision. First, we "extend" the trace by unrolling the recorded commands in Figure 7b once, as in Figure 7d. Events that belong to the second trace are renamed to those with a prime, to distinguish them from those in the first trace. Second, dependences on the fence in the second trace are replaced with the actual dependences on operations in the first trace. In the unrolled commands, each operation that belongs to the second trace either immediately or transitively depends on fence $\mathtt{e}'_1$ that blocks operations in the first trace. After we remove that fence, each operation individually waits for dependent operations in the first trace. For each region instance of a task instance, the predecessors from the first trace are identified as follows:

- If the task instance can write to the region instance r, all readers and writers of r are added to the predecessors.
- If the task instance only reads from r, only the writers of r are added to the predecessors.

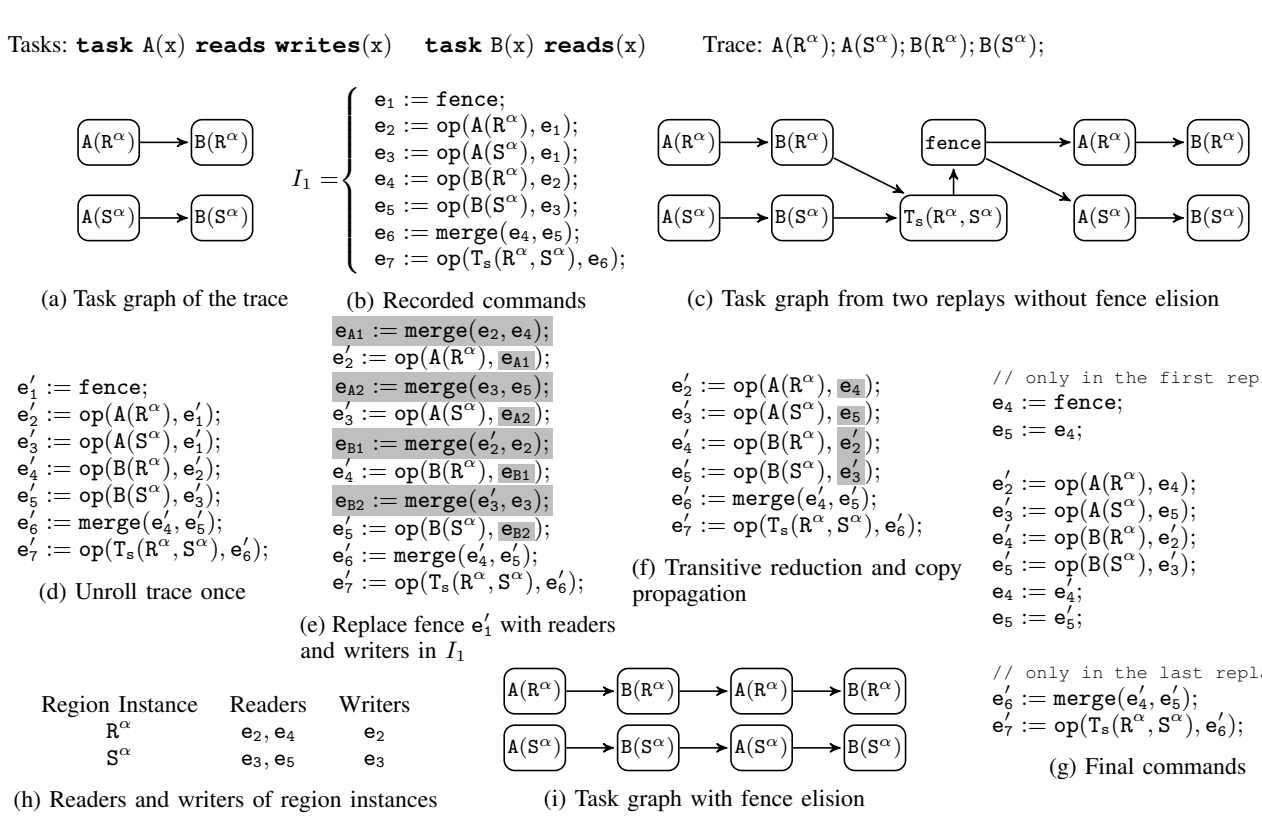
For example, the original predecessor event $\mathtt{e}'_2$ of task instance $\mathtt{B}(\mathtt{R}^\alpha)$ is merged with event $\mathtt{e}_2$, which is the writer of $\mathtt{R}^\alpha$ in the first trace, to get a new predecessor event $\mathtt{e}_{\mathtt{B1}}$ in Figure 7e. Once all uses of the fence are replaced with individual events, transitive reduction and copy propagation are applied to the commands. (The result is in 7f.) Finally, we generalize the optimized commands to get the final commands in Figure 7g

Algorithm 1: Dynamic tracing algorithm

```
   Data: A tracing state ST ∈ {DEP, TRACE}, initially DEP
   Data: A current trace TR, initially ∅
 1 Procedure DynamicTracing(call):
 2     if call is a task :
 3         T ← Map(call)
 4         if ST is DEP :
 5             AnalyzeDependence(T)
 6         elseif ST is TRACE :
 7             TR ← TR;T
 8     elseif call is begin_trace :
 9         ST ← TRACE
10         TR ← ∅
11     elseif call is end_trace :
12         RecordOrReplay()
13         ST ← DEP
14 Procedure RecordOrReplay():
15     if ∃ recording R for TR that passes precondition check :
16         Replay(R)
17         ApplyPostcondition(R)
18     else:
19         R ← Record(TR)
20         register R to the runtime system
```

Algorithm 2: Dynamic tracing algorithm with optimizations for idempotent recordings

```
   Data: A tracing state ST ∈ {DEP, TRACE}, initially DEP
   Data: A current trace TR, initially ∅
   Data: A previous trace TR', initially ∅
   Data: A previous recording R', initially ∅
 1 Procedure DynamicTracing(call):
 2     if call is a task :
 3         T ← Map(call)
 4         if ST is DEP :
 5             if R' is idempotent :
 6                 ApplyPostcondition(R')
 7                 R' ← ∅
 8             AnalyzeDependence(T)
 9         elseif ST is TRACE :
10             TR ← TR;T
11     elseif call is begin_trace :
12         ST ← TRACE
13         TR ← ∅
14     elseif call is end_trace :
15         RecordOrReplay()
16         ST ← DEP
17 Procedure RecordOrReplay():
18     if TR = TR' ∧ R' is idempotent :
19         Replay(R')
20     else:
21         if R' is idempotent :
22             ApplyPostcondition(R')
23         if ∃ recording R for TR that passes precondition
            check :
24             Replay(R)
25             if R is not idempotent :
26                 ApplyPostcondition(R)
27         else:
28             R ← Record(TR)
29             register R to the runtime system
30         R' ← R
31         TR' ← TR
```



(a) Task graph of the trace

(b) Recorded commands

(c) Task graph from two replays without fence elision

(d) Unroll trace once

(e) Replace fence $\mathtt{e}'_1$ with readers and writers in I_1

(f) Transitive reduction and copy propagation

(g) Final commands

(h) Readers and writers of region instances

(i) Task graph with fence elision

Fig. 7: Fence elision

for repeated replays. Note that these commands can be used in both one-time replay and repeated replays. Figure 7i shows a task graph from two replays with fence elision.

We can also concatenate two different traces in a similar way when one's postcondition subsumes the precondition of another. However, concatenating two different traces is of less use than unrolling the same trace as the latter appears more frequently in real applications.

V. Implementation

We have implemented dynamic tracing in Legion, a C++ runtime system for task parallelism [1]. Legion has a dependence analysis pipeline similar to the one described in Section III and builds a task graph using Realm [2], a low-level system for building and executing distributed task graphs. We augment Legion's existing dependence analysis to generate graph calculus programs for traces. Graph calculus is implemented as a set of commands that internally call the Realm API to construct task graphs.

In the rest of this section, we briefly discuss the ways in which our implementation for Legion extends the algorithm presented in this paper.

A. Overlapping Regions

For simplicity of exposition, we have assumed that regions and region instances are unique and each region instance represents only one region. In the Legion programming model, regions can be partitioned into subregions, and thus can overlap with each other in non-trivial ways. This complicates the dependence analysis, but has no fundamental impact on how dynamic tracing generates graph calculus commands.

B. Fills and Reductions

Legion provides *fills*, which are lazy copies from a constant value. We incorporate fills as another kind of operation that can be used in the `op` command.

As discussed in Section 3, Legion tasks can request reduction permission on regions. Reduction tasks are parallelized by summarizing the update from each task into a temporary instance and lazily aggregating such instances to compute the final value when it is requested in subsequent tasks.

C. Parallel Dependence Analysis

A task can launch subtasks, and thus tasks that run in parallel can generate their own streams of tasks. The programming model guarantees the children of independent tasks are also independent and thus concurrent task streams are independent of each other [1]. We have extended dynamic tracing to support concurrent, distributed task streams by taking separate recordings per stream and replaying them independently.

The Legion runtime also pipelines dependence analysis of task streams. The recording procedure of graph calculus commands is divided into several steps, one for each pipeline phase.

D. Parallel Trace Replay

As sequential replay of a trace can become a performance bottleneck, we implemented parallel replay of a trace. Figure 8

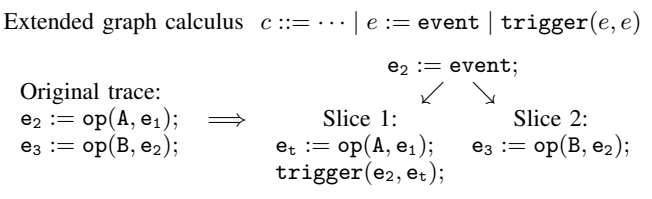


Fig. 8: Transformation for parallel trace replay

illustrates the key transformation for parallel replay. A trace is split into *slices*. Commands appear in slices in the same order as the original trace and any events that are created in one slice and referenced in other slices are connected using the graph calculus extension shown in the figure. A command $\texttt{e} := \texttt{event}$ creates a new untriggered event and assigns it to an event variable $\texttt{e}$. A command $\texttt{trigger}(\texttt{e}_1, \texttt{e}_2)$ registers an event dependence such that event $\texttt{e}_1$ is notified as soon as $\texttt{e}_2$ is triggered, which simply corresponds to adding an edge between the operations represented by $\texttt{e}_2$ and $\texttt{e}_1$. Slices generated from a trace can be replayed in parallel.

Minimizing events that "cross" the slice boundary is important for reducing the number of intermediate events for parallel replay, for which we exploit the implicit knowledge encoded in an application's task mappings: We put tasks mapped to the same processor in the same slice as much as possible because in a well-mapped program they are more likely to have dependences on one another.

VI. Evaluation

We evaluate dynamic tracing on five programs ranging from quite small and regular and static, to full applications that are complex and irregular: Stencil [23], a 9-point stencil benchmark on 2D grids; Circuit [24], a circuit simulator for unstructured circuit graphs; PENNANT [25] and MiniAero [26], proxy applications for unstructured meshes from Los Alamos and Sandia National Laboratories; and Soleil-X [27], [28], a compressible fluid solver on 3D grids developed to study turbulent fluid flow in channels. All programs are written in Regent [29], a high-level programming language that targets Legion, and were run with *control replication* [24], an optimization that is orthogonal to dynamic tracing. These programs have competitive or better weak scaling performance than reference implementations [30], where reference implementations are available. Table I shows benchmark metrics, the number of the tasks and copies each node must analyze per iteration. Programs using unstructured meshes have indirect indexing on regions, which require dependences be resolved dynamically. For the two biggest programs, MiniAero and Soleil-X, Legion's dynamic task scheduling plays a crucial role in overlapping communication with computation as their tasks have parallelism due to field non-interference [31]; i.e., a task accessing field f of a region can run in parallel with a copy for field g of the same region initiated by another task. For three programs (Stencil, PENNANT, and MiniAero), we are able to compare with publicly available reference MPI versions.

	Stencil	Circuit	PENNANT	MiniAero	Soleil-X
Num. tasks	16	27	67	288	448
Num. copies	31	49	54	552	928

TABLE I: Number of tasks and copies per iteration

Due to their iterative nature, all five programs have a "main" loop where they spend most of their execution time. For Stencil, Circuit, and PENNANT, we annotate the body of this main loop. For MiniAero and Soleil-X, which implement a fourth-order Runge-Kutta time marching scheme, we set the annotation on the body of this time marching loop nested within the main loop. Each application has only one trace because there is no change in the task mapping and dynamic tracing is able to find one idempotent recording of the trace. Identifying loops that merit annotation was trivial for these programs and could easily be automated.

We measured performance when the program reached steady state; i.e., the state where the program starts replaying a recording repeatedly.

We use GCC 5.3 to compile the Legion runtime and the MPI reference implementations. Regent uses LLVM for code generation; we use LLVM 3.8. We report performance for each application on up to 256 nodes of the Piz Daint supercomputer [32], a Cray XC50 system; nodes are connected by an Aries interconnect and each node has 64 GB of memory and one Intel Xeon E5-2690 CPU with 12 physical cores.

For strong scaling measurements, we chose problem sizes for which runs stop scaling ideally without dynamic tracing at 32 or fewer nodes. Table II summarizes the results as throughput normalized by single node throughput without dynamic tracing. Dynamic tracing improves the speedup of applications by 4.2× or more, except for PENNANT, which is improved by 2.8×. Unlike the other programs, the main loop in PENNANT is guarded by a convergence predicate that in turn prevents a replay of the trace until the condition is resolved. A trace replay overlaps with tasks only for 25% or less of the time per iteration, which explains an improvement that is 4× off of the improvement in the runtime overhead. Circuit shows the biggest discrepancy between the improvement in the runtime overhead and strong scaling simply because the runs did not reach a point where they are limited by the replay overhead.

To study the effect of optimizations for idempotent recordings, we also measure the performance of runs where dynamic tracing is used without those optimizations (column Tr.\Opt.). The use of idempotent recordings improves performance by an average of 5% and a maximum of 19%. More importantly, dynamic tracing without optimizations sometimes perform worse than the run without dynamic tracing because of spurious dependences introduced by fences, which means fence elision is crucial. The only program immune to the absence of optimizations is Circuit, which has all-to-all dependences between tasks on each node, which results in sightly longer sequences of graph calculus commands after fence elision.

For Stencil, PENNANT, and MiniAero, we also compare performance with expert-written MPI reference versions (column MPI); these applications are static and well suited to MPI-style programming. Note that the MPI versions of Stencil and

Nodes	Normalized Throughputs																			
	Stencil (0.4B cells)					Circuit (74K wires)			PENNANT (29M zones)					MiniAero (1M cells)				Soleil-X (8.4M cells)		
	No Tr.	Tr.\Opt.	Tr.	MPI 9R	MPI 12R	No Tr.	Tr.\Opt.	Tr.	No Tr.	Tr.\Opt.	Tr.	MPI 9R	MPI 12R	No Tr.	Tr.\Opt.	Tr.	MPI 8R	No Tr.	Tr.\Opt.	Tr.
1	1.0	1.0	1.0	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.9	1.2	1.0	*0.9*	1.0	0.3	1.0	1.0	1.0
2	2.0	2.0	2.0	1.9	2.1	2.0	2.0	2.0	2.2	2.2	2.2	1.9	2.3	2.1	*1.8*	2.1	0.6	2.0	2.0	2.0
4	4.1	*4.0*	4.1	3.7	4.1	3.9	4.0	3.9	4.5	*4.3*	4.6	3.8	4.6	4.2	*3.8*	4.4	1.3	3.3	3.8	3.9
8	8.1	*8.0*	8.2	7.4	8.2	7.3	7.5	7.6	8.6	*8.4*	9.1	7.6	9.3	8.1	8.7	9.0	3.0	5.3	6.9	7.2
16	16.0	*15.7*	16.2	14.8	16.3	13.3	13.6	14.0	16.6	*15.3*	16.8	14.9	18.3	14.9	17.3	16.8	7.3	7.6	11.9	12.7
32	31.3	*30.1*	32.0	29.2	32.4	**24.7**	25.5	25.9	29.6	*28.4*	30.3	29.2	36.4	21.7	31.3	32.1	16.0	9.0	17.9	18.7
64	50.1	54.3	61.9	57.9	63.3	18.0	48.1	49.7	54.0	55.8	60.8	56.7	71.8	**24.0**	54.0	55.0	30.0	10.2	31.5	32.4
128	68.3	108.0	126.3	116.6	134.3	8.3	87.4	90.2	**71.4**	106.2	117.6	115.6	139.6	23.7	90.9	94.8	51.0	**10.6**	53.0	54.8
256	**77.2**	**269.7**	**320.0**	**259.8**	**387.5**	4.2	**133.7**	**131.4**	68.8	**185.0**	**198.5**	**211.4**	**250.1**	22.9	**121.4**	**123.4**	**58.2**	5.7	**69.5**	**74.0**
$\frac{\max(\text{Tr.})}{\max(\text{No Tr.})}$	4.2					5.3			2.8					5.1				7.0		

TABLE II: Strong scaling performance. Numbers in bold face show the maximum throughput achieved in each configuration. Underlined numbers mean that the runs performed worse than those without dynamic tracing. Columns 8R, 9R, and 12R show numbers from runs with 8, 9, and 12 ranks, respectively.

PENNANT are 21-26% faster than the Legion versions. Legion requires resources for its runtime system to make dynamic decisions (e.g., about tracing). In these experiments the Legion runtime is configured to use 3 CPUs (out of 12) per node. When the MPI versions use the same number of application processors as Regent counterparts (column 9R), MPI Stencil performs worse than the Regent version and MPI PENNANT is slower up to 128 nodes and becomes 6% better on 256 nodes. The MPI reference of MiniAero, which only allows the number of ranks to be a power of 2, starts 3× slower than the Regent version, which is consistent with [24], and loses scalability earlier.

Lastly, we use MiniAero and Soleil-X to calculate the average task granularity supported by dynamic tracing. In these two applications, tasks are almost completely overlapped with the runtime overhead and copies, which make them suitable for studying task granularity. Table III shows the minimum time per iteration and the number of tasks each processor runs, from which we derive the average task granularity. The task granularity for MiniAero is half that for Soleil-X because Soleil-X has roughly twice as many regions per task as MiniAero, leading to twice as many copies on average to replay per task (5.4 regions per task on average vs. 3.1).

VII. Related Work

Dynamic tracing can be applied to any task-based system that constructs task graphs using dynamic dependence analysis [4], [5], [11], [19]. Hoque et al. [4] reports that dynamic task-based systems require a larger granularity of tasks than explicitly parallel programs to be efficient; our results show that dynamic tracing can eliminate most of that overhead.

Execution templates [6] have a goal similar to dynamic tracing. Both aim to reduce the overhead for executing tasks on distributed memory systems. However, the program representation used by execution templates is explicitly parallel as it requires each command to specify a *before set*: a set of previous commands for which the command must wait; in contrast dynamic tracing takes a stream of implicitly parallel tasks. Furthermore, execution templates map nodes in a task graph directly onto workers, thereby requiring *edits* to the graph for any subsequent changes in scheduling, etc. In dynamic tracing the execution of a task graph is decoupled from graph construction, and therefore dynamic tracing still provides scheduling flexibility.

	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

TABLE III: Average task granularity

Dynamic tracing and Inspector-Executor (I/E) methods [33], [34] are based on the same record-and-replay idea, but to achieve orthogonal goals; I/E methods record working sets of irregular array accesses, whereas dynamic tracing memoizes dynamic tasks dependences. As a result the details are very different; for example, I/E methods do not need to deal with the possibility of dynamic changes in runtime mapping decisions for data.

VIII. Conclusion

Dynamic tracing improves strong scaling performance by efficiently capturing and soundly replaying task graphs of traces. We have presented a complete design of dynamic tracing with several key optimizations. We have also demonstrated that an implementation of dynamic tracing improves strong scaling performance of already optimized Regent applications by an average of 4.9× at 256 nodes.

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.

REFERENCES

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing (SC)*, 2012.

[2] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[3] C. Augonnet *et al.*, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 187–198, Feb. 2011.

[4] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17, 2017.

[5] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," http://tensorflow.org/, 2015.

[6] O. Mashayekhi, H. Qu, C. Shah, and P. Levis, "Execution templates: Caching control plane decisions for strong scaling of data analytics," in *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[7] D. Sbîrlea, Z. Budimlić, and V. Sarkar, "Bounded memory scheduling of dynamic task graphs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, 2014.

[8] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling for heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15.

[9] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé, "Periodic hierarchical load balancing for large supercomputers," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 371–385, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010394383

[10] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal, "Fault-tolerant dynamic task graph scheduling," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014.

[11] Q. Meng, J. Luitjens, and M. Berzins, "Dynamic task scheduling for the uintah framework," in *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.

[12] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009.

[13] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz, "Tracing for web 3.0: Trace compilation for the next generation web applications," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09, 2009.

[14] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz, "Trace-based compilation in execution environments without interpreters," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, ser. PPPJ '10, 2010.

[15] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter, "Spur: A trace-based jit compiler for cil," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10, 2010.

[16] S.-y. Guo and J. Palsberg, "The essence of compiling with traces," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11, 2011.

[17] S. Dissegna, F. Logozzo, and F. Ranzato, "Tracing compilation by abstract interpretation," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14, 2014.

[18] M. Tillenius, "Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization," *SIAM J. Scientific Computing*, vol. 37, no. 6, 2015. [Online]. Available: https://doi.org/10.1137/140989716

[19] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA, Research Report RR-8927, Jun. 2016.

[20] "CUDA programming guide 9.1," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Sept. 2013.

[21] "The OpenCL Specification, Version 1.0," The Khronos OpenCL Working Group, December 2008.

[22] "The Open Community Runtime interface," https://xstackwiki.modelado.org/images/1/13/Ocr-v0.9-spec.pdf, 2014.

[23] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels," in *HPEC*, 2014, pp. 1–6.

[24] 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," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*, 2017.

[25] C. R. Ferenbaugh, "PENNANT: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, 2014.

[26] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[27] "Predictive science academic alliance program (psaap) ii," https://exascale.stanford.edu/, 2013.

[28] "Soleil-x," https://github.com/stanfordhpccenter/soleil-x, 2013.

[29] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015*, 2015, pp. 81:1–81:12.

[30] E. Slaughter, "Regent: A high-productivity programming language for implicit parallelism with logical regions," Ph.D. dissertation, Stanford University, 2017.

[31] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Structure slicing: Extending logical regions with fields," in *Supercomputing (SC)*, 2014.

[32] "Piz Daint & Piz Dora - CSCS," http://www.cscs.ch/computers/piz_daint, 2018.

[33] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Supercomputing (SC)*, 2012.

[34] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Distributed memory code generation for mixed irregular/regular computations," ser. PPoPP. ACM, 2015, pp. 65–75.

APPENDIX

In this appendix, we evaluate how much dynamic tracing reduces the cost of dynamic dependence analysis by measuring the runtime overhead with and without dynamic tracing. We use the synthetic benchmark program in Figure 9, which has two desirable properties. First, the program performs no actual computation so we can count all execution time as runtime overhead. Second, the program exhibits a simple pattern of task dependencies, which allows to compute a bound on the possible improvement from dynamic tracing. Each iteration of the outer most loop launches `N` parallel tasks `S` times where `N` is the number of CPUs remaining after allocating some for the runtime. The tasks form `N` chains of dependent tasks, where the `i`th chain consists of `S` tasks that read and write region `A[i]`. Figure 10 illustrates the task graph of the synthetic benchmark program.

We place the tracing annotation on the outer `for` loop (lines 4 and 10) and vary the value of `S` to study the effect of trace size ($\mathtt{S} \cdot \mathtt{N}$) on the reduction of runtime overhead. We also run the program with different numbers of runtime threads to measure the benefit of parallel replay.

Figure 11a shows the improvement in the runtime overhead for four configurations of parallel replay. The legend shows the number of runtime threads being allocated for parallel dynamic dependence analysis and trace replay, and also the corresponding value of `N`. In all four plots, a longer trace leads to a greater improvement in the runtime overhead as it better amortizes the constant overhead of initializing every trace replay.

The plots also show that increasing the number of runtime threads has diminishing returns, which occurs for two reasons.

```
1  task F(x) reads(x),writes(x) do end
2
3  while * do
4    begin_trace
5    for s = 0,S do
6      for i = 0,N do
7        F(A[i])
8      end
9    end
10   end_trace
11 end
```

Fig. 9: Synthetic benchmark program

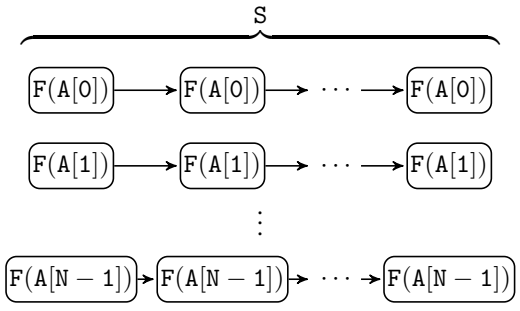


Fig. 10: Task graph of the program in Figure 9

First, dynamic tracing only reduces the runtime overhead for dependence analysis and there are several other steps in Legion's task processing pipeline. Second, the performance of parallel dependence analysis and trace replay scale sub-linearly in the number of runtime threads, because both parallel dependence analysis and trace replay have portions that run sequentially; Legion performs a sequential preliminary analysis on tasks for parallelizing the subsequent dependence analysis and dynamic tracing sequentially initializes crossing events for parallel trace replay. To better understand how these two factors incur diminishing returns, we use the following model $O_{\mathsf{dep}}(T)$ of runtime overhead when the number of runtime threads is T:

$$O_{\mathsf{dep}}(T) = C_{\mathsf{dep}} \cdot s(T) + \frac{C_{\mathsf{pipe}}}{T},$$

where C_{dep} denotes the dependence analysis overhead with one runtime thread, C_{pipe} is all the cost of Legion's task processing pipeline except for dependence analysis, and $s(T)$ models the sub-linear speedup governed by Amdahl's law; i.e.,

$$s(T) = \frac{1}{(1-p) + p/T},$$

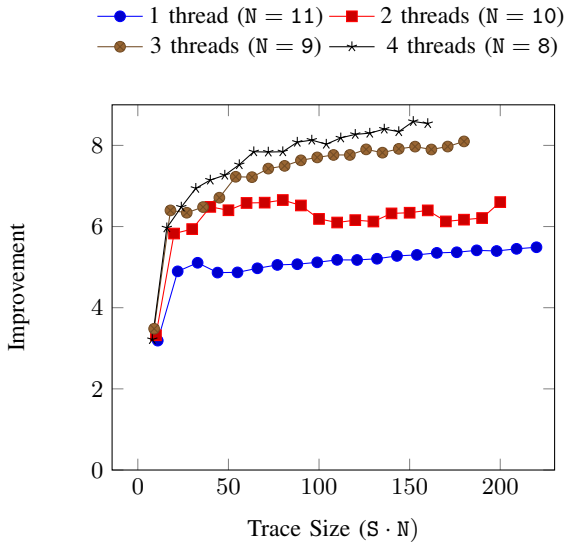
where p is the proportion of dependence analysis that is parallelized ($0 < p < 1$). In the model, we assume the cost C_{pipe} of Legion's task pipeline except for dependence analysis can be perfectly parallelized across T threads as they are embarrassingly parallel. The model $O_{\mathsf{replay}}(T)$ of the trace replay overhead when the number of runtime threads is T is the same as $O_{\mathsf{dep}}(T)$ except that the dependence analysis overhead is replaced with the parallel trace replay overhead $C_{\mathsf{replay}} \cdot s(T)$:

$$O_{\mathsf{replay}}(T) = C_{\mathsf{replay}} \cdot s(T) + \frac{C_{\mathsf{pipe}}}{T},$$

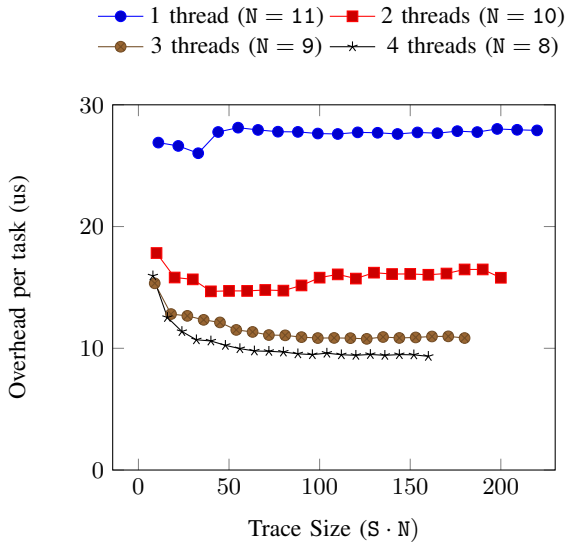
where C_{replay} denotes the trace replay overhead with one runtime thread. (We use the same $s(T)$ to model the sub-linearity of both parallel dependence analysis and trace replay, to simplify the analysis, though using two different models does not change the result.) The improvement $I(T)$ in runtime overhead is a ratio of $O_{\mathsf{dep}}(T)$ to $O_{\mathsf{replay}}(T)$:

$$I(T) = \frac{O_{\mathsf{dep}}(T)}{O_{\mathsf{replay}}(T)} = \frac{C_{\mathsf{dep}} + C_{\mathsf{pipe}}/(s(T) \cdot T)}{C_{\mathsf{replay}} + C_{\mathsf{pipe}}/(s(T) \cdot T)}.$$

Note that as T increases, $I(T)$ approaches asymptote $I = C_{\mathsf{dep}}/C_{\mathsf{replay}}$; this means that the improvement in the dependence analysis overhead becomes a dominant component in $I(T)$. Finally, the return $R(T) = I(T+1) - I(T)$ of using an additional runtime thread when there are T threads reaches 0 as T goes to infinity (i.e., $\lim_{T\to\infty} R(T) = 0$), which implies that $R(T)$ is diminishing as T increases. The plot of $R(T)$ in Figure 12 also clearly shows the trend of diminishing returns. (For the plot, we fit our model to the experiment results by assuming that dependence analysis is 10× heavier than the rest of analysis pipeline, that 90% of parallel dependence analysis and trace replay is perfectly parallelized, and that dynamic



(a) Improvement in the runtime overhead



(b) Average overhead per task with dynamic tracing

Fig. 11: Runtime overhead of the synthetic benchmark program

tracing eliminates 85% of the dependence analysis overhead; i.e., $10C_{\mathsf{pipe}} = C_{\mathsf{dep}}$, $C_{\mathsf{replay}} = 0.15C_{\mathsf{dep}}$, and $p = 0.9$.)

Figure 11b shows the average runtime overhead per task with dynamic tracing. Average overhead per task decreases as trace size increases and eventually saturates once the overhead for initializing trace replay is sufficiently amortized. The plots exhibit a similar trend of diminishing returns as those in Figure 11a, but because of Amdahl's law; the $C_{\mathsf{replay}} \cdot s(T)$ term becomes dominant in $O_{\mathsf{replay}}(T)$ as T increases.

Next, we also evaluate how much dynamic tracing reduces the runtime overhead for five applications used in Section VI. To isolate the runtime overhead from application work or communication, we apply the same methodology used for the synthetic benchmark: We modify applications to only launch tasks and run no actual computations, and we count their execution time as runtime overhead. We allocate three runtime threads, the configuration used in the strong scaling runs. Table IV summarizes the measured runtime overhead per trace. In all five applications, dynamic tracing reduces the runtime overhead by more than 7×. Circuit and PENNANT enjoy noticeably greater improvement than the others because they have reduction tasks and copies that make dynamic dependence analysis more expensive. Table IV also shows the one-time cost for trace optimization, which is just a few milliseconds even for the longest trace.

The improvement in runtime overhead gives an upper bound on the possible improvement in strong scaling performance. The actual strong scaling improvement in Section VI is influenced by many factors (such as inter-node communication) of which runtime overhead is just one, though it is often the most important one.

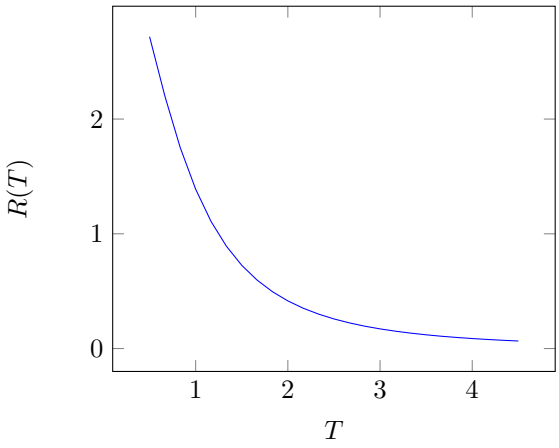


Fig. 12: Diminishing return function $R(T)$

	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)