Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises

Feiyang Jin
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Lechen Yu
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Tiago Cogumbreiro
College of Science and Mathematics, University of Massachusetts Boston, MA, USA

Jun Shirako
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Vivek Sarkar
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Abstract

Much of the past work on dynamic data-race and determinacy-race detection algorithms for task parallelism has focused on structured parallelism with fork-join constructs and, more recently, with future constructs. This paper addresses the problem of dynamic detection of data-races and determinacy-races in task-parallel programs with *promises*, which are more general than fork-join constructs and futures. The motivation for our work is twofold. First, promises have now become a mainstream synchronization construct, with their inclusion in multiple languages, including C++, JavaScript, and Java. Second, past work on dynamic data-race and determinacy-race detection for task-parallel programs does not apply to programs with promises, thereby identifying a vital need for this work.

This paper makes multiple contributions. First, we introduce a featherweight programming language that captures the semantics of task-parallel programs with promises and provides a basis for formally defining *determinacy* using our semantics. This definition subsumes functional determinacy (same output for same input) and structural determinacy (same computation graph for same input). The main theoretical result shows that the absence of data races is sufficient to guarantee determinacy with both properties. We are unaware of any prior work that established this result for task-parallel programs with promises. Next, we introduce a new Dynamic Race Detector for Promises that we call DRDP. DRDP is the first known race detection algorithm that executes a task-parallel program sequentially without requiring the serial-projection property; this is a critical requirement since programs with promises do not satisfy the serial-projection property in general. Finally, the paper includes experimental results obtained from an implementation of DRDP. The results show that, with some important optimizations introduced in our work, the space and time overheads of DRDP are comparable to those of more restrictive race detection algorithms from past work. To the best of our knowledge, DRDP is the first determinacy race detector for task-parallel programs with promises.

2012 ACM Subject Classification Software and its engineering → Software creation and management; Software and its engineering → Software verification and validation; Software and its engineering → Software defect analysis; Software and its engineering → Software testing and debugging; Software and its engineering → Software notations and tools; Software and its engineering → General programming languages; Software and its engineering → Concurrent programming languages

Keywords and phrases Race detection, Promise, Determinism, Determinacy-race, Serial projection

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.13

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.24`

Funding This material is based upon work supported by the National Science Foundation under Grant No. 2204986.

```
 1  x ← alloc
 2  y ← new_promise
 3  async {
 4    store 5 to x
 5    y.set
 6    return
 7  }
 8  a ← load x
 9  y.get
10  b ← load x
11  return
```

Figure 1 Example program.

1 Introduction

In recent years, promises have been incorporated as a general synchronization construct into multiple mainstream languages, including C++ [17], Java [28], and JavaScript [27]. A promise is a wrapper for a data payload that is initially empty. It typically has two operations which we refer to as `set` and `get`. Each `get` operation blocks until the promise receives a value for its payload; multiple `get` operations may be performed on the same promise from multiple tasks, and they all return the same value. Figure 1 shows the usage: an async task sets the promise at line 5, and the main task gets the promise at line 9.

A promise with a set payload is referred to as a *fulfilled promise*. Following standard conventions, we assume that promises obey the single-assignment policy, where an invocation of `set` on a fulfilled promise (i.e., a second assignment) will induce a runtime error. Compared to futures [20], promises generalize the semantics for synchronization in that a promise need not be bound to the return value of a specific task; instead, any task can choose to perform a `set` operation on a given promise. Promises support arbitrary point-to-point synchronization wherein one or more tasks can await the arrival of a payload for which the producer task is not known in advance. However, it has been observed that the convenience of this generality may also be accompanied by the increasing complexity of dynamic analysis for bug detection in parallel programs [42].

As with any source of parallelism, accesses to shared memory locations must be correctly ordered to avoid *determinacy races* [15], defined as race conditions causing non-determinism. A determinacy race often results from a *data race* [26, 31], which occurs when two concurrent memory accesses operate on the same memory location and at least one of them is a write. A key result in our paper is that the absence of data races is sufficient to guarantee determinacy for task-parallel programs with promises; in contrast, this property does not hold for programs that use mutual exclusion constructs such as locks or transactions.

For dynamic race detectors, enumerating all possible inputs is usually intractable; therefore, they are typically *per-input* or *per-schedule* race detectors. Per-input race detectors report all potential races for a given input by covering all possible thread schedules [15, 30, 31, 35, 43–45], whereas per-schedule race detectors only cover the observed schedule when analyzing a program's execution [1, 16, 33]. However, prior work related to dynamic determinacy race detection has some major limitations:

1. None of these race detectors support promises.
2. No formal definition of "determinacy" is provided. For example, in the SP-Bags paper [15], the authors state "[determinacy race]... may cause the program to behave nondeterministically. That is, different runs of the same program may produce different behaviors". However, no formal definition was provided for what is meant by "different behaviors."
3. It has been observed in some past work (e.g., [7,31,35]) that for certain classes of task-parallel programs, data-race freedom leads to determinacy but no formal proof was given for this observation.

In this paper, we introduce a featherweight programming language that captures the semantics of Task-Parallel Programs with Promises (TP3) and use it as a basis for formally defining determinacy, along with a proof that data-race freedom implies both structural and functional determinacy for TP3. We also designed and implemented a race detector for TP3. A major obstacle is that the tasks issuing `get`/`set` operations on a given promise cannot be identified in advance. As a result, TP3 do not satisfy the *serial-projection* property [32], i.e., the property that a sequential execution of the program with all parallel constructs removed is guaranteed to be a legal execution of the original parallel program. This feature is utilized by a number of dynamic race detectors [4,15,30,35,37]. Without this property, the sequential execution of a parallel program may be blocked by a `get` operation. To address this and other challenges, we extended the Habanero-C/C++ library [8] to enable correct single-worker execution of such programs via cooperative task switching. However, keeping track of happens-before relationships also becomes more challenging in the presence of task switching. Efficient data structures used previously [4,15,30,35,37] rely on the serial-projection property to maintain happens-before information correctly and cannot directly be used for programs with promises.

In summary, the key contributions of this paper are as follows:

1. A formalization of determinacy for TP3 using a featherweight programming language (Section 2).
2. A proof that TP3 are guaranteed to be determinate in the absence of data races (Section 3).
3. A new dynamic race detection algorithm for TP3 called Determinacy Race Detector for Promises (DRDP, Section 4). To the best of our knowledge, this is the first precise and efficient per-input race detector for TP3.
4. An implementation of DRDP on top of the Habanero-C/C++ library and its evaluation on a set of benchmarks using promises (Section 5). The results show that, with some essential optimizations introduced in our work, the space and time overheads of DRDP are comparable to those of more restrictive race detection algorithms from past work.

2 A Featherweight Language for Task-Parallel Programs with Promises

In this section, we introduce a featherweight language that features task parallelism and promises to establish determinacy. Our language is Turing-complete. We do not include functions and types because we aim to use this core language to prove properties of dynamic program executions that are possible from a given static program, rather than using this language for static program analysis.

Program	P	::=	$s\,;P$ | `return`	
Statement	s	::=	m	
		|	`async`$\{P\}$	(create asynchronous task)
		|	$x \leftarrow$ `new_promise`	
		|	x`.set`	
		|	x`.get`	
		|	$x \leftarrow \mathtt{e}$	
		|	`while` $(0 \neq$ `load` $y)$ $\{P\}$	
Memory Operation	m	::=	$y \leftarrow$ `alloc`	
		|	$x \leftarrow$ `load` y	
		|	`store e to` y	
Expression	e	::=	x	(local variable)
		|	y	(local variable that saves shared variable name)
		|	c	(integer constant)
		|	$\mathtt{e}_1 + \mathtt{e}_2$	
Runtime value	k	::=	c | r | p	

Figure 2 Syntax of the core language.

2.1 Language Syntax

Figure 2 lists the syntax of our language. A program consists of a sequence of statements terminated by a `return`. A single statement can be task creation (`async`)[1], a memory operation (`alloc`, `load`, and `store`), a promise operation (`new_promise`, `set`, `get`), a local variable assignment (`x ← e`), or a conditional loop (`while`). We use symbols x and y to denote local variables within a specific task, along with an LLVM-style syntax and convention.

The language syntax introduces two types of variables: local variables and shared variables. Each task has a set of scoped local variables which must satisfy the single-assignment rule; apart from local variables, tasks may also access shared variables using memory operations. In fact, the only way for two tasks[2] to share data is via memory operations.

Shared variables are modeled as a global map (i.e., *memory*), in which each instance has a unique name assigned during its allocation; this name serves as a *reference* for operating on the corresponding shared variable. Statement $y \leftarrow$ `alloc` allocates a new shared variable, with its reference saved into y. Statement $x \leftarrow$ `load` y retrieves a shared variable's content using the reference in local variable y and saves the retrieved content into another local variable x. Likewise, statement `store e to` y locates a shared variable using the reference in y and updates the shared variable's content with value `e`.

Statement $x \leftarrow \mathtt{e}$ initializes single-assignment local variable x with the value `e`. According to the syntax, `e` could be an integer, a local variable, or a sum of two expressions. Statement `async`$\{P\}$ spawns a concurrent task to execute the body P. Statement `while` $(0 \neq$ `load` $y)$ $\{P\}$ will continuously execute the loop body P until the condition no longer holds. Statement $x \leftarrow$ `new_promise` creates a new promise, and saves a reference to the promise in local variable x. Statement x`.set` signals promise x, and statement x`.get` blocks until a task issues a `set` operation on promise x. To simplify the presentation, we eschew the data payload of promises, thus only offering synchronization functionality. Communicating data through a promise is still possible, but must be encoded using additional shared variables.

[1] We do not include join operations like `finish` and `sync` in our language, since they can be modeled using sets of promises.

[2] For convenience, the main program is considered to be a root task.

2.2 Runtime State

The runtime state σ of our language is a pair $\sigma = (M, G)$, where M maps shared variables into runtime values, and a *computation graph* G. We denote a map M as $\{r_1 : k_1, r_2 : k_2, \ldots r_n : k_n\}$. We use the notation $M[r := k]$ to extend M. We define computation graphs G inductively using the rules in Figure 3.

$n ::= t \mid \text{Unful}(p) \mid \text{Ful}(p)$	Runtime nodes
$t ::= [f, P, L]$	Task nodes
$G ::= t$	
$\mid G + t_1 \xrightarrow{cont} t_2$	Continue edges
$\mid G + t_2 \xleftarrow{fork} t_1 \xrightarrow{cont} t_3$	Fork edges
$\mid G + \text{Unful}(p) \xleftarrow{np} t_1 \xrightarrow{cont} t_2$	New promise edges
$\mid G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} t_1 \xrightarrow{cont} t_2$	Set edges
$\mid G + t_1 \xrightarrow{cont} t_2 \xleftarrow{join} \text{Ful}(p)$	Join edges

$$\text{R-NAT}\;\frac{}{c \Downarrow_{t_{mem}} c}$$

$$\text{R-ADD}\;\frac{e_1 \Downarrow_{t_{mem}} c_1 \qquad e_2 \Downarrow_{t_{mem}} c_2}{e_1 + e_2 \Downarrow_{t_{mem}} c_1 + c_2}$$

$$\text{R-VAR}\;\frac{}{x \Downarrow_{t_{mem}} t_{mem}[x]}$$

Figure 3 Computation graph and reduction rule.

In a computation graph G, nodes represent the states of tasks and promises across time, and edges represent happens-before relations. There are three types of nodes in a computation graph: a *task node* t, an unfulfilled promise Unful(p), and fulfilled promise Ful(p). A task node t is a triple that includes a task name f, a program P to execute, and a map L representing the task's local variables. We define three helper functions to obtain the content of a task node t: t_{tid}, t_{code}, and t_{mem}, which return the task name ($t.f$), the program ($t.P$), and the local variable map ($t.L$), respectively.

The base case of a computation graph is a task node t. Notation $G + t_1 \xrightarrow{cont} t_2$ depicts a computation graph acquired by adding a new continue edge from t_1 to t_2 on top of the original graph G. Notation $G + t_2 \xleftarrow{fork} t_1 \xrightarrow{cont} t_3$ captures the semantics of task creation: node t_1 issues the task creation, node t_2 represents the child task, and node t_3 represents the continuation after spawning t_2. Since t_2 and t_3 may happen in parallel, there exists no path between the two nodes. Notation $G + \text{Unful}(p) \xleftarrow{np} t_1 \xrightarrow{cont} t_2$ represents the semantics of promise creation. Node Unful(p) represents the spawned instance of promise and t_2 represents the continuation after promise creation. Notation $G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} t_1 \xrightarrow{cont} t_2$ depicts an invocation of `set` issued by node t_1. Node Ful(p) represents the fulfilled promise after the `set`, and t_2 represents the task state upon the `set` operation. Finally, Notation $G + t_1 \xrightarrow{cont} t_2 \xleftarrow{join} \text{Ful}(p)$ signifies a `get` operation. Node t_1 issues the synchronization, while node t_2 observes the synchronization.

Example. The corresponding computation graph of the example program in Figure 1 is shown in Figure 4. Note that there is a determinacy race between line 4 and line 8 in the program. In different executions, variable a can be either 0 or 5. The race is also reflected in the computation graph, as no path connects the two nodes in red. On the other hand, the store in line 4 happens before the load on line 10 because of the `get` operation in line 9; the final value of variable b will always be five, regardless of the actual task schedule.

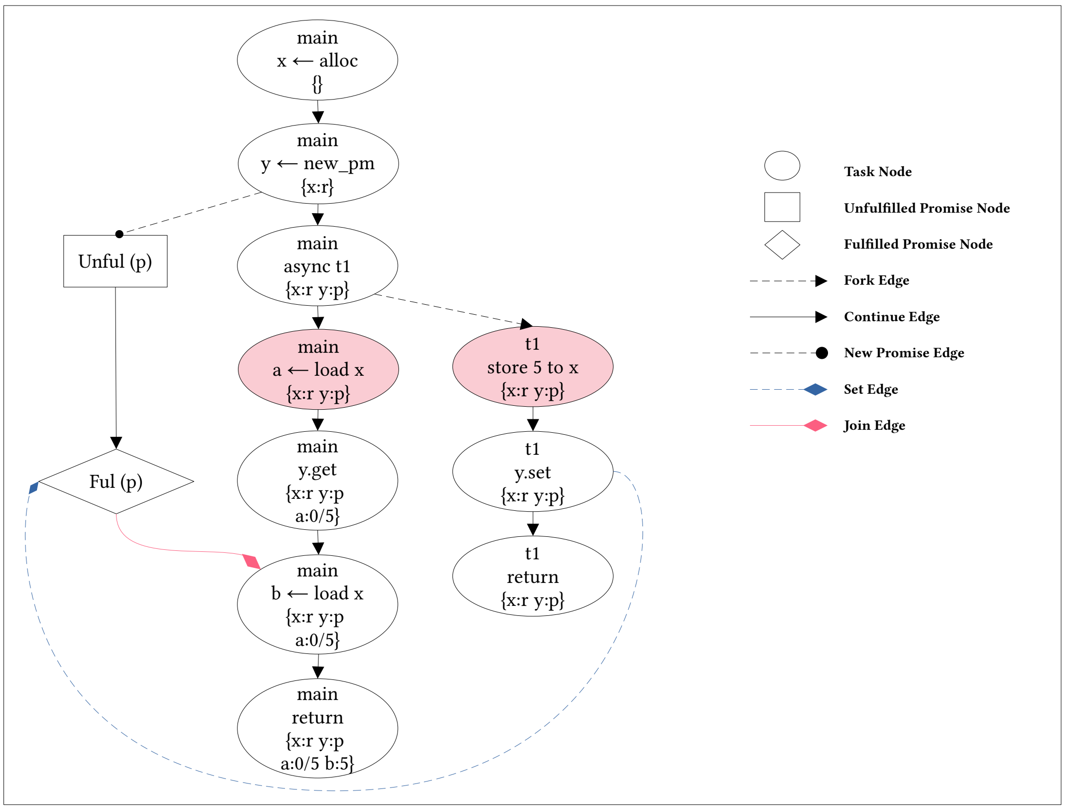


Figure 4 Associated computation graph of Figure 1.

Leveraging the formal definition in Figure 3, we formalize the happens-before relation between two task nodes and memory accesses performed by task nodes.

▶ **Definition 1** (Happens-before). We say node v *precedes* or *happens before* node u if and only if one directed path from v to u exists in the computation graph. We denote the happens-before relation as $v \rightsquigarrow u$. We use $v \not\rightsquigarrow u$ to indicate that there exists no path from u to v.

▶ **Definition 2** (May-happen-in-parallel). Node v *may happen in parallel* with node u, denoted by $v \parallel u$, iff $u \not\rightsquigarrow v$ and $v \not\rightsquigarrow u$.

▶ **Definition 3** (Read and Write). A task node t *reads* from a shared variable r if 1). $t_{code} = \{x \leftarrow \texttt{load}\ y\ ;\ P\}$, $t_{mem}[y] = r$, or 2). $t_{code} = \{\texttt{while}\ (0 \neq \texttt{load}\ y)\ \{P'\}\ ;\ P\}$, $t_{mem}[y] = r$. A task node t *writes* to a shared variable r if $t_{code} = \{\texttt{store}\ x\ \texttt{to}\ y\ ;\ P\}$ and $t_{mem}[y] = r$. Node t *accesses* a shared variable r if node t reads from or writes to r.

$$\text{(1) G-ASYNC}\ \frac{t_{code} = \texttt{async}\{P'\}\,;P \qquad g \text{ does not occur in } G \qquad t_{tid} = f}{(M, G) \to (\ M, G + [g, P', t_{mem}] \xleftarrow{fork} t \xrightarrow{cont} [f, P, t_{mem}]\ )}$$

$$\text{(2) G-ALLOC}\ \frac{t_{code} = y \leftarrow \texttt{alloc}\,;P \qquad r \notin M \qquad t_{tid} = f}{(M, G) \to (\ M[r := 0], G + t \xrightarrow{cont} [f, P, t_{mem}[y := r]]\ )}$$

$$\text{(3) G-LOAD}\ \frac{t_{code} = x \leftarrow \texttt{load}\ y\,;P \qquad t_{mem}[y] = r \qquad M[r] = k \qquad t_{tid} = f}{(M, G) \to (\ M, G + t \xrightarrow{cont} [f, P, t_{mem}[x := k]]\ )}$$

$$\text{(4) G-STORE}\ \frac{t_{code} = \texttt{store e to}\ y\,;P \qquad \texttt{e} \Downarrow_{t_{mem}} k \qquad t_{mem}[y] = r \qquad t_{tid} = f}{(M, G) \to (\ M[r := k], G + t \xrightarrow{cont} [f, P, t_{mem}]\ )}$$

$$\text{(5) G-PROMISE}\ \frac{t_{code} = x \leftarrow \texttt{new\_promise}\,;P \qquad \text{Unful}(p) \notin G \qquad t_{tid} = f}{(M, G) \to (\ M, G + \text{Unful}(p) \xleftarrow{np} t \xrightarrow{cont} [f, P, t_{mem}[x := p]]\ )}$$

$$\text{(6) G-SET}\ \frac{t_{code} = x.\texttt{set}\,;P \qquad t_{mem}[x] = p \qquad \text{Unful}(p) \text{ no outgoing edges in } G \qquad t_{tid} = f}{(M, G) \to (\ M, G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} t \xrightarrow{cont} [f, P, t_{mem}]\ )}$$

$$\text{(7) G-GET}\ \frac{t_{code} = x.\texttt{get}\,;P \qquad t_{mem}[x] = p \qquad \text{Ful}(p) \in G \qquad t_{tid} = f}{(M, G) \to (\ M, G + t \xrightarrow{cont} [f, P, t_{mem}] \xleftarrow{join} \text{Ful}(p)\ )}$$

$$\text{(8) G-ASSIGN}\ \frac{t_{code} = x \leftarrow \texttt{e}\,;P \qquad \texttt{e} \Downarrow_{t_{mem}} k \qquad t_{tid} = f}{(M, G) \to (\ M, G + t \xrightarrow{cont} [f, P, t_{mem}[x := k]]\ )}$$

$$\text{(9) G-WHILE-1}\ \frac{t_{code} = \texttt{while}\ (0 \neq \texttt{load}\ y)\ \{P'\}\,;P \qquad M[t_{mem}[y]] = 0 \qquad t_{tid} = f}{(M, G) \to (\ M, G + t \xrightarrow{cont} [f, P, t_{mem}]\ )}$$

$$\text{(10) G-WHILE-2}\ \frac{t_{code} = \texttt{while}\ (0 \neq \texttt{load}\ y)\ \{P'\}\,;P \qquad M[t_{mem}[y]] \neq 0 \qquad t_{tid} = f}{(M, G) \to (\ M, G + t \xrightarrow{cont} [f, P'\,; \texttt{while}\ (0 \neq \texttt{load}\ y)\ \{P'\}\,;P, t_{mem}]\ )}$$

Figure 5 Small-step semantics.

2.3 Small-step Operational Semantics

We introduce the small-step operational semantics, denoted by $\sigma \to \sigma'$, in Figure 5. Spawning a task g with async creates a new node for the child task, which inherits the local memory of the parent task f (Rule 1). Memory allocation creates a new shared variable r, initializes it with 0, and assigns its name r to y in the local memory (Rule 2). A load retrieves the content of shared variable r from the shared memory M. A store writes the value e to shared variable r (variable name r is stored in local variable y). Creating a promise adds a new node that marks promise p as unfulfilled (Rule 5). Our semantics only allows a single `set` per promise; thus, a pre-condition of Rule 6 is to ensure that promise p is unfulfilled. Getting a promise links the fulfilled promise Ful(p) to the continuation node $[f, P, t_{mem}]$, thus adding a happens-before relation from the `set` to the `get` (Rule 7). Assignment extends local memory with a new variable x, which has the value of evaluating expression `e` (Rule 8). Rules 9 and 10 handle a while loop in a standard way.

Let root(P) denotes runtime state $(m, [main, P, l])$: a runtime state that holds the initial memory m and the initial computation graph with a single vertex $[main, P, l]$, where l is the local memory for the single vertex. When it is clear from the context we may say P to signify the runtime state root(P). Let notation $P \Downarrow \sigma$ be defined as root$(P) \rightarrow^{\star} \sigma$ and $\sigma \not\rightarrow \sigma'$ for any σ'.

We use a naming convention that gives *unique* and *consistent* name to each variable in the operational semantics. “unique” means whenever a new task, variable, or promise is being created, it will be given a unique name that does not yet exist in σ; “consistent” means the name is the same name that other program runs will have when creating this new task, variable or promise.

3 Proof of Determinism

In this section, we show that determinacy-race freedom implies determinism for programs in our featherweight language. Our proof structure is adapted from prior work on Concurrent Collection (CnC) [6, Theorem 1]. CnC applies a single-assignment policy on all shared variables (called *data collections* in CnC) to assure determinism. An important distinction between this work and [6] is that our formalism expresses both promises and shared memory, whereas [6] only expresses a construct akin to promises. Our proof utilizes the property of determinacy-race freedom to show that the shared memory and the computation graph will be determinate for a given input (the initial program state root(P)).

▶ **Definition 4** (Determinacy Race and Determinacy-Race Freedom)**.** A *determinacy race* is a triple (r, t_1, t_2); it happens on a shared variable r if and only if two task nodes t_1 and t_2 access r, at least one of them conducts a write, and $t_1 \parallel t_2$. A computation graph G is *determinacy-race-free* if and only if for any task nodes t_1, t_2 in G, there is no determinacy race (r, t_1, t_2) for any shared variable r. A program P is *determinacy-race-free* if for any G and M such that $P \rightarrow^{\star} (M, G)$, G is determinacy-race-free.

We define an ordering $\leq$ on runtime states such that $\sigma \leq \sigma'$ if and only if $dom(\sigma.M) \subseteq dom(\sigma'.M)$ and $\sigma.G$ is a subgraph of $\sigma'.G$. Next, we establish the necessary lemmas to prove our main result: if $P \Downarrow \sigma$ and $P \Downarrow \sigma'$, then $\sigma = \sigma'$.

▶ **Lemma 5** (Monotonicity)**.** *If $\sigma \rightarrow \sigma'$, then $\sigma \leq \sigma'$*

Proof. The proof follows by case analysis on the derivation of $\sigma \rightarrow \sigma'$. Let $\sigma = (M, G)$. The key insight is that nodes and edges are only added to G; nodes and edges are never removed. Similarly, the domain of M either grows or remains the same. We omit the proof details. ◀

We use $\sigma \rightarrow_v \sigma'$ to denote that executing node v triggers the state transition.

▶ **Lemma 6** (Independence)**.** *Let $\sigma \rightarrow_v \sigma'$, $\sigma \rightarrow_u \sigma''$ and $\sigma' \neq \sigma''$. We have $v \parallel u$.*

Proof. If $v \rightsquigarrow u$ or $u \rightsquigarrow v$, we can only derive either σ' or σ'' from σ, but not both. ◀

The next Lemma proves what we call strong local confluence. This property essentially implies that from the same program state σ, if there exists more than one choice to proceed, those different choices will eventually proceed to the same state σ_c. The proof also reveals why determinacy-race freedom is necessary for this property.

▶ **Lemma 7** (Strong Local Confluence). *Let P be determinacy-race-free and $P \to^* \sigma$. If $\sigma \to_v \sigma'$ and $\sigma \to_u \sigma''$, then there exists σ_c, i, j such that $\sigma' \to^i \sigma_c$, $\sigma'' \to^j \sigma_c$, $i \leq 1$ and $j \leq 1$.*

Proof. If $v = u$, we have $\sigma' = \sigma''$; in this case $\sigma_c = \sigma', i = 0, j = 0$. If $v \neq u$, we claim $\sigma' \to_u \sigma_c, \sigma'' \to_v \sigma_c, i = 1, j = 1$. To prove the claim, we do a case analysis on the rule used to derive $\sigma \to_v \sigma'$.

- (3) G-LOAD: We know $v_{code} = x \leftarrow \texttt{load}\ y\,; P'$ and $v_{mem}[y] = r$ and $M[r] = k$. We have $\sigma'.G = \sigma.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$, $\sigma'.M = \sigma.M$. Let us do a case analysis of the rule used to derive $\sigma \to_u \sigma''$.
  1. G-ALLOC: We know $u_{code} = y' \leftarrow \texttt{alloc}\ ; P''$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}[y' := r'], P'']$, $\sigma''.M = \sigma.M[r' := 0]$. Because our naming system is unique, we have $r' \neq r$. We can pick σ_c such that $\sigma_c.M = \sigma''.M$, $\sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$. It is clear that $\sigma'' \to_v \sigma_c$. It is also true that $\sigma' \to_u \sigma_c$ because our naming system is consistent among different execution. In this case, $i = 1, j = 1$.
  2. G-LOAD: $u_{code} = x' \leftarrow \texttt{load}\ y'\,; P''$ and $u_{mem}[y'] = r'$ and $M[r'] = k'$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}[x' := k'], P'']$, $\sigma''.M = \sigma.M$. In this case, it is fine that $r' = r$ because concurrent reads on the same memory location are allowed. We can pick σ_c such that $\sigma_c.M = \sigma''.M$, $\sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$. It is clear that $\sigma' \to_u \sigma_c$ and $\sigma'' \to_v \sigma_c$. In this case, $i = 1, j = 1$.
  3. G-STORE: We know $u_{code} = \texttt{store e' to}\ y'\,; P''$ and $\texttt{e}' \Downarrow k'$ and $u_{mem}[y'] = r'$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P'']$, $\sigma''.M = \sigma.M[r' := k']$. If $r' = r$, this is a determinacy race by Definition 4 and Lemma 6. Because $r' \neq r$, we can pick σ_c such that $\sigma_c.M = \sigma''.M, \sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$. It is clear that $\sigma' \to_u \sigma_c$ and $\sigma'' \to_v \sigma_c$. In this case, $i = 1, j = 1$.
  4. Any other rules: we omit the details for other rules because the proof is similar to the above cases.
- (4) G-STORE: We know $v_{code} = \texttt{store e to}\ y\,; P'$ and $\texttt{e} \Downarrow k$ and $v_{mem}[y] = r$. We have $\sigma'.G = \sigma.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$, $\sigma'.M = \sigma.M[r := k]$. Let us do a case analysis of the rule used to derive $\sigma \to_u \sigma''$.
  1. G-STORE: We know $u_{code} = \texttt{store e' to}\ y'\,; P''$ and $\texttt{e}' \Downarrow k'$ and $u_{mem}[y'] = r'$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P'']$, $\sigma''.M = \sigma.M[r' := k']$. If $r' = r$, this is a determinacy race by Definition 4 and Lemma 6. Because $r' \neq r$, we can pick σ_c such that $\sigma_c.M = \sigma''.M[r := k], \sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \to_u \sigma_c$ and $\sigma'' \to_v \sigma_c$. In this case, $i = 1, j = 1$.
  2. G-WHILE-*: We know $u_{code} = \texttt{while}\ (0 \neq \texttt{load}\ y')\ \{P'''\}\,; P''$ and $u_{mem}[y'] = r'$. If $r' = r$, this is a determinacy race by Definition 4 and Lemma 6. Because $r' \neq r$, we can pick σ_c such that $\sigma_c.M = \sigma''.M[r := k], \sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \to_u \sigma_c$ and $\sigma'' \to_v \sigma_c$. In this case, $i = 1, j = 1$.
  3. Any other rules: we omit the details for other rules because the proof is similar to the above cases.
- (6) G-SET: we know $v_{code} = x.\texttt{set}\ ; P'$ and $v_{mem}[x] = p$ and Unful(p) has no outgoing edge. We have $\sigma'.G = \sigma.G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$, $\sigma'.M = \sigma.M$. Let us do a case analysis of the rule used to derive $\sigma \to_u \sigma''$.
  1. G-Set: we know $u_{code} = x'.\texttt{set}; P''$ and $u_{mem}[x'] = p'$ and Unful(p') has no outgoing edge. We have $\sigma''.G = \sigma.G + \text{Unful}(p') \xrightarrow{cont} \text{Ful}(p') \xleftarrow{set} u \xrightarrow{cont} [u_{tid}, u_{mem}, P'']$, $\sigma''.M = \sigma.M$. If $p = p'$, this violates the single set policy for promises. Because $p \neq p'$, we can pick σ_c such that $\sigma_c.M = \sigma''.M, \sigma_c.G = \sigma''.G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \to_u \sigma_c$ and $\sigma'' \to_v \sigma_c$. In this case, $i = 1, j = 1$.

2. G-GET: we know $u_{code} = x'.\texttt{get}\ ;\ P''$ and $u_{mem}[x'] = p'$ and $\text{Ful}(p') \in G$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P''] \xleftarrow{join} \text{Ful}(p')$, $\sigma''.M = \sigma.M$. In this case, we must have $p \neq p'$ because otherwise, we cannot make the step $\sigma \to_u \sigma'$ until v is executed. We can pick σ_c such that $\sigma_c.M = \sigma''.M$, $\sigma_c.G = \sigma''.G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \to_u \sigma_c$ and $\sigma'' \to_v \sigma_c$. In this case, $i = 1, j = 1$.
3. Any other rules: we omit the details for other rules because the proof is similar to the above cases.

- (7) G-GET: we know $v_{code} = x.\texttt{get}\ ;\ P'$ and $v_{mem}[x] = p$ and $\text{Ful}(p) \in G$. We have $\sigma'.G = \sigma.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P'] \xleftarrow{join} \text{Ful}(p)$, $\sigma'.M = \sigma.M$. Let us do a case analysis of the rule used to derive $\sigma \to_u \sigma''$.
  1. G-GET: we know $u_{code} = x'.\texttt{get}\ ;\ P''$ and $u_{mem}[x'] = p'$ and $\text{Ful}(p') \in G$ We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P''] \xleftarrow{join} \text{Ful}(p')$, $\sigma''.M = \sigma.M$. In this case, it is fine if $p = p'$ because concurrent `get` operations performed on the same promise is allowed. We can pick σ_c such that $\sigma_c.M = \sigma''.M, \sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P'] \xleftarrow{join} \text{Ful}(p)$. It is clear that $\sigma' \to_u \sigma_c$ and $\sigma'' \to_v \sigma_c$. In this case, $i = 1, j = 1$.
  2. Any other rules: we omit the details for other rules because the proof is similar to the above cases.
- Any other rules v and u could execute: we omit the details for other rules because the proof is similar to the above cases. ◀

▶ **Lemma 8** (Strong One-Sided Confluence). *Let P be determinacy-race-free and $P \to^* \sigma$. If $\sigma \to \sigma'$, $\sigma \to^m \sigma''$, where $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \to^i \sigma_c$, $\sigma'' \to^j \sigma_c$, $i \leq m$ and $j \leq 1$.*

Proof. We prove it by inducting on m.

Base case: $m = 1$. Proved by Lemma 7.

Induction step: suppose $\sigma \to^m \sigma'' \to \sigma'''$. By our induction hypothesis, the lemma holds for m. We have σ'_c, i', j' such that $\sigma' \to^{i'} \sigma'_c$ and $\sigma'' \to^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq 1$. We want to prove the lemma holds for $m + 1$. There are two cases based on the value of j':

1. $j' = 0$. In this case, $\sigma'_c = \sigma''$. We pick $\sigma_c = \sigma''', i = i' + 1, j = 0$. Because $i' \leq m$, we have $i \leq m + 1$, which obeys the lemma.
2. $j' = 1$. In this case, we have $\sigma'' \to \sigma'''$ and $\sigma'' \to \sigma'_c$. By Lemma 7, there exist σ_d, a, b such that $\sigma''' \to^a \sigma_d$ and $\sigma'_c \to^b \sigma_d$ and $a \leq 1$ and $b \leq 1$. So we also have $\sigma' \to^{i'} \sigma'_c \to^b \sigma_d$. As a result, we pick $\sigma_c = \sigma_d, i = i' + b, j = a$. This is fine because $i = i' + b \leq m + 1$ and $j = a \leq 1$. ◀

▶ **Lemma 9** (Strong Confluence). *Let P be determinacy-race-free and $P \to^* \sigma$. If $\sigma \to^n \sigma'$, $\sigma \to^m \sigma''$, where $1 \leq m, 1 \leq n$, then there exist σ_c, i, j such that $\sigma' \to^i \sigma_c$, $\sigma'' \to^j \sigma_c$, $i \leq m$ and $j \leq n$.*

Proof. We prove it by inducting on n.

Base case: $n = 1$. Proved by Lemma 8.

Induction step: suppose $\sigma \to^n \sigma' \to \sigma'''$. By our induction hypothesis, the lemma holds for n. We have σ'_c, i', j' such that $\sigma' \to^{i'} \sigma'_c$ and $\sigma'' \to^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq n$. We want to prove the lemma holds for $n + 1$. There are two cases based on the value of i':

1. $i' = 0$. In this case, $\sigma' = \sigma'_c$. We pick $\sigma_c = \sigma''', i = 0, j = j' + 1$. Because $j' \leq n$, we have $j \leq n + 1$, which obeys the lemma.

2. $i' \geq 1$. In this case, we have $\sigma' \to \sigma'''$ and $\sigma' \to^{i'} \sigma'_c$. By Lemma 8, there exist σ_d, a, b such that $\sigma''' \to^a \sigma_d$ and $\sigma'_c \to^b \sigma_d$ and $a \leq i'$ and $b \leq 1$. So we also have $\sigma'' \to^{j'} \sigma'_c \to^b \sigma_d$. As a result, we pick $\sigma_c = \sigma_d, i = a, j = j' + b$. This is fine because $i = a \leq i' \leq m$ and $j = j' + b \leq n + 1$. ◀

▶ **Lemma 10** (Confluence). *Let P be determinacy-race-free and $P \to^* \sigma$. If $\sigma \to^* \sigma'$ and $\sigma \to^* \sigma''$, then there exists σ_c such that $\sigma' \to^* \sigma_c$ and $\sigma'' \to^* \sigma_c$.*

Proof. Implied by Lemma 9 ◀

With Lemma 10, we are ready to present our main theorem. Notice that we do not assume deadlock freedom for the program P. The notation $P \Downarrow \sigma$ is defined as P cannot make further progress; some `get` operations never resume after blocking because no task elects to set the required promises. However, our main theorem reveals that even if deadlock(s) exists in a determinacy-race-free program P, any execution of P will still reach the same final state (same shared memory, same computation graph).

▶ **Theorem 11** (Determinism). *Let P be determinacy-race-free. If $P \Downarrow \sigma$ and $P \Downarrow \sigma'$, then $\sigma = \sigma'$.*

Proof. By Lemma 10, we have σ_c such that $\sigma \to^* \sigma_c$ and $\sigma' \to^* \sigma_c$. Given that neither σ nor σ' can proceed, we must have $\sigma = \sigma' = \sigma_c$. ◀

4 DRDP Race Detection Algorithm

Race detection for parallel programs has evolved with the development of parallel programming models. The widely-used ThreadSanitizer [33] and other vector-clock-based race detectors [16, 21] work well for multithreaded programs with lock-based synchronization. More recently, task-based parallel programming models have gained popularity for developing parallel programs intended to be determinate, i.e., these programs are always expected to compute the same results when given the same inputs. The work to be carried out is decomposed into a large number of user-defined fine-grained tasks, and dependencies among tasks are specified using join operations/futures/promises rather than locks. Task-parallel programs execute on a group of *worker* threads, with the actual schedule of tasks on worker threads determined adaptively and automatically by a runtime system. Although vector-clock-based race detectors can be applied to task-parallel programs at the worker-thread level, such an approach may also exhibit false negative results. For two tasks executing on the same worker thread, a vector-clock-based race detector may enforce a happens-before relation between them, and then fail to identify potential data races[3]. On the other hand, it is not practical to use such race detectors by treating each task as a thread. Task-parallel programs may create millions of tasks at runtime, making it intractable to store associated vector clocks of spawned tasks in the memory space. Other researchers have also made similar observations about the limitations of using the vector clock approach for task parallelism [29, 44, 45].

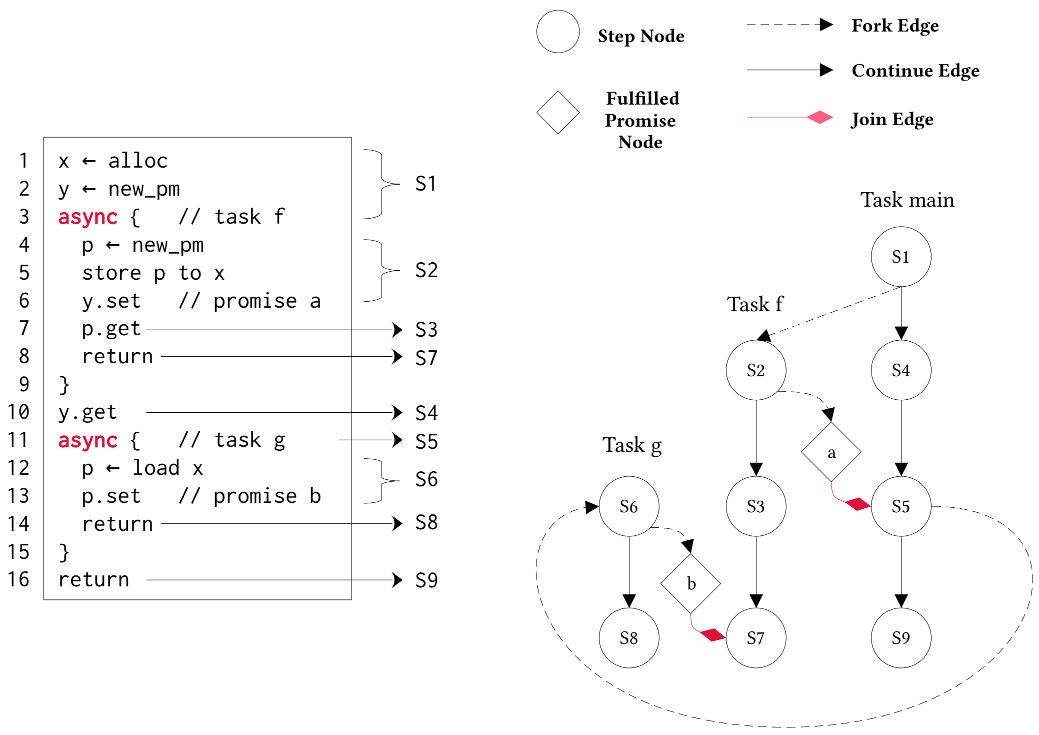
Per-input dynamic race detectors designed for task parallelism can usually be classified by the task-parallel constructs they support. Different task-parallel constructs impose different structural constraints on the computation graphs generated by programs, and the determinacy race detection problem becomes more challenging as the computation

[3] ThreadSanitizer limits the vector clock size to 256 [11], and can also exhibit false negatives for programs with larger numbers of threads.

graphs become more general [4, 15, 26, 30, 31]. For example, SP-Bags is a race detection algorithm designed for spawn-sync task-parallel programs which only generate fully strict computation graphs [15]. ESP-Bags is an extension of SP-Bags that can support the more general terminally strict computation graphs [30] generated by async-finish task parallelism. More recent algorithms have been introduced for task-parallel programs with futures, one as an extension to async-finish constructs [35] and others as an extension to spawn-sync constructs [37, 43].

In this section, we introduce DRDP, a dynamic determinacy race detector taking task parallelism and promises into account. DRDP is based on our theoretical conclusion in Section 3. For better time and memory efficiency, we make two revisions to the notation of computation graph. Such changes do not affect the precision of race detection.

- We introduce *step nodes* to replace task nodes. A step node is a sequence of statements without task creation, `set`, or `get` except the last statement. For example, node $s1$ in Figure 6 is a step node ending with a task creation.
- We simplify the computation graph construction related to promises. Every promise has only one corresponding node in the computation graph (see promises a and b in Figure 6), created when the `set` happens.



```
x ← alloc
y ← new_pm
async {   // task f
  p ← new_pm
  store p to x
  y.set   // promise a
  p.get
  return
}
y.get
async {   // task g
  p ← load x
  p.set   // promise b
  return
}
return
```

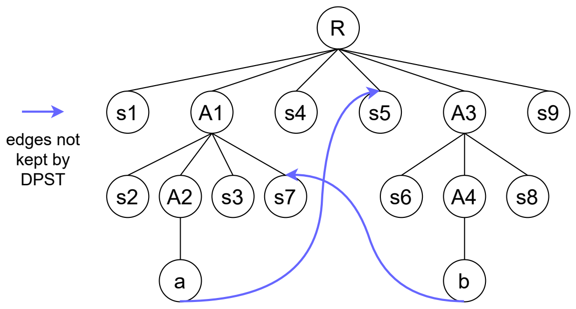
Figure 6 Example program and its computation graph.

As with other dynamic race detectors, DRDP consists of 1). a *reachability data structure* that keeps track of the happens-before relationship and 2). a *shadow memory* that records access history for every memory location. We first introduce our reachability structure, which is built on-the-fly as the program executes.

4.1 DRDP Reachability Data Structures

The reachability data structure of DRDP is adapted from previous work [31, 35] on async-finish task parallelism (but no support for promise). The statement "finish { s }" causes the current task to execute the body s and wait for all spawned tasks, including transitively created tasks within the finish statement.

We leverage *Dynamic Program Structure Tree* (DPST) [31] to encode happens-before relations created by async-finish parallelism efficiently. Figure 7 shows an instance of DPST associated with the example program in Figure 6. DPST resembles the computation graph by nature. Each leaf node in the DPST represents a step node. Each internal node in the DPST is either an `async` or a `finish` node, denoting an instance of such construct in the program. Node $A1$ in Figure 7 represents the `async` in line 3 of the example program and $A3$ represents the one in line 11. The other two `async` nodes, $A2$ and $A4$, represent the promise `set` in lines 6 and 13.



set	task	nt	lsa
0	[main]	[a]	
1	[f]	[b]	
2	[g]		(main, [a])
3	[a]		
4	[b]		(main, [a])

Figure 7 DPST and set information for the example program.

For any two step nodes s_i, s_j in a DPST, their happens-before relation ($\rightsquigarrow$) can be determined by examining the children of their least common ancestor (LCA). Assume s_i is to the left of s_j in the DPST. Among all children of the two node's LCA (denoted as $lca(s_i, s_j)$), a node V (denoted as $lca\_lc(s_i, s_j)$) must exist such that V is a's ancestor or a itself. If V is not an `async` node, $a \rightsquigarrow b$; otherwise, $a \parallel b$.

As an example, in Figure 7, step nodes $s1$ and $s2$ have the root R as their LCA. The node V, in this case, is $s1$ itself because $s1$ is a child of R. Since $s1$ is a step node, DPST will report $s1 \rightsquigarrow s2$, which is confirmed by the path from $s1$ to $s2$ in Figure 6.

Limitation of DPST. DPST may report incorrect happens-before relationships when naively applied to programs with promises. Let us consider step nodes $s2$ and $s6$ in Figure 7. Their LCA is R, and among R's children, the one that is $s2$'s ancestor is $A1$. The DPST-based happens-before check will decide that $s2 \parallel s6$. However, Figure 6 clearly shows a path from $s2$ to $s6$, so the happens-before check returns an incorrect result. The reason is that DPST does not consider synchronization semantics brought by promises.

Thus, we need to maintain additional information for those happens-before relations incurred by promises. We refer to those promise join edges in a computation graph as *non-tree joins* (nt) because DPST does not store them. For other task joins kept by DPST, we call them *tree joins*. The problem turns into how to store these non-tree joins efficiently.

Inspired by previous work [35], we use disjoint sets [12] to effectively save non-tree joins information. Tasks synchronized by tree joins will be grouped into the same set. Each set will maintain its non-tree joins, plus the lowest ancestor with at least one non-tree join, which we refer to as the *least-significant ancestor* (LSA). How do we use and maintain set, non-tree joins, and LSAs will be introduced in Section 4.3.

DRDP Example. From a high-level perspective, DRDP checks happens-before relations in two stages. It first carries out the DPST-based happens-before check. If the happens-before check returns that the two steps may happen in parallel, DRDP conducts an additional graph traversal on the computation graph. The graph traversal loops through non-tree joins and LSAs to see if the two steps are ordered by non-tree joins.

We now explain how DRDP can find out $s2 \rightsquigarrow s6$ by the information in Figure 7. DRDP first checks DPST and finds $lca\_lc(s2, s6)$ is an `async` node, so it continues with the graph traversal. Because $s6$ is in task g and task g is in set 2, DRDP examines the column nt of set 2. In this case, we do not have any non-tree join for set 2. Finally, DRDP checks the column lsa of set 2, realizes set 2's lsa is the main task, and the main task has a non-tree join from task a. Now, DRDP knows $a \rightsquigarrow s6$. The goal becomes deciding if $s2 \rightsquigarrow a$. DRDP inspects DPST and finds $lca\_lc(s2, a)$ is a step node, which indicates $s2 \rightsquigarrow a$. Now DRDP can conclude $s2 \rightsquigarrow s6$.

4.2 DRDP Task Scheduling and Shadow Memory

Existing race detectors [4, 15, 30, 35, 37] that execute programs sequentially usually rely on the *serial-projection property*, which ensures that there is a legal scheduling strategy that executes a parallel program sequentially without any blocking. Crucially, however, programs with promises do not enjoy the serial-projection property.

As a solution, our runtime follows a depth-first execution order and switches to task T's parent task recursively when T is blocked. As soon as some other task S sets the promise that T is waiting on, the worker thread will suspend task S and resume the execution of task T. If the fulfilled promise P enables more than one task, we execute these enabled tasks in the same order as they were placed in the waiting queue for promise P. The worker switches back to task S after all tasks previously blocked on promise P are finished. If a deadlock exists in the program, our race detector works up to the point of deadlock.

Shadow memory also needs careful design due to possible blockings. For each memory location, we save the last step node that writes to it, plus a read list that records all step nodes reading from the memory location after the last write. When a write occurs, if no race is found with the recorded historical accesses, we empty the reader list and save the current step node as the last write. When a read occurs, if no race is found with the last write, we add it to the reader list.

4.3 Algorithm

DRDP algorithm is presented in Figure 8 and Figure 9. Figure 8 explains how we maintain the reachability structure and the shadow memory, which happens per instruction of the program being executed; Figure 9 defines procedure $PRECEDE$ for happens-before check. When encountering a read or write to memory location M, DRDP will do race checks as in lines 40 - 44 and lines 45 - 55; if the function reports a race, it means a race exists between current access and previous access to M.

The built-in function $run\_eager(U)$ suspends the execution of current task T and starts executing task U. For tasks returning from blocking, run_eager will resume their execution from the first statement after blocking.

Now we briefly explain each callback executed by DRDP.

Task creation: When a task is created, we initialize some information and create a disjoint set. The worker then starts executing the task until finished or blocked.

Task termination: We set the task state from active to finished.

```
// Task creation:
// task $T$ creates task $U$
$S_U$ = $new\ set\_info(U)$
$set[U.id]$ = $S_U$
$U.state$ = $ACTIVE$
$U.parent$ = $T.id$

$S_T$ = $set[T.id]$
if $S_T.nt.isEmpty()$ then
    $S_U.lsa$ = $S_T.lsa$
else
    $S_U.lsa$ = $lsa\_info\ (T.id, S_T.nt)$
end
$run\_eager(U)$

// Task termination:
// task $T$ terminates
$T.state$ = $FINISHED$

// Promise set:
// task $T$ sets promise $P$
$P.setter\_task\_id$ = $T.id$
$P.empty\_task\_id$ = $(new\ task()).id$
foreach $task\ X\ waiting\ on\ P$ do
    $run\_eager(X)$
end

// Promise get:
// task $T$ gets promise $P$
if $P.satisfied$ == $false$ then
    $T.state$ = $BLOCKED$
    $run\_eager(T.parent)$
end
$x$ = $new\ nt\_info\ (P.empty\_task.id)$
$S_T$ = $set[T.id]$
$S_T.nt$ = $S_T.nt \cup x$

// Finish end:
// Finish $F$ ends in task $T$
$S_T$ = $set[T.id]$
foreach $task\ X\ in\ F$ do
    $S_X$ = $set[X.id]$
    $nt$ = $S_T.nt \cup S_X.nt$
    $lsa$ = $S_T.lsa$
    $S_T$ = $Union(S_T, S_X)$
    $S_T.nt$ = $nt$
    $S_T.lsa$ = $lsa$
end

// Read check:
// step s reads from memory_location
    M
$w$ = $M.writer$
if $PRECEDE(w, s)$ == $false$ then
    report race
end
$M.reader\_list$ = $M.reader\_list + s$

// Write check:
// step s writes to memory_location
    M
$w$ = $M.writer$
if $PRECEDE(w, s)$ == $false$ then
    report race
end
foreach $r\ in\ M.reader\_list$ do
    if $PRECEDE(r, s)$ == $false$ then
        report race
    end
end
$M.reader\_list$ = {}
$M.writer$ = $s$
```

Figure 8 DRDP algorithm parts that maintain data structures and shadow memory.

Promise set: We create an empty task, and if any task is enabled, the worker will begin to execute enabled tasks. The worker will go back to the current task sometime in the future.

Promise get: If the promise is not yet set, we block the current task and switch to its parent recursively; if the promise is set, we add its empty task to the current set's nt.

Finish end: This happens at the end of a finish statement. The finish's owner task will merge all spawned tasks in the finish by keeping its original LSA plus unionizing the tasks and non-tree joins (nt) from the merged sets.

Read check: If current step s reads from memory location M, we first check race against M's $writer$. If no race is reported, we directly add s to the $reader\_list$.

Write check: If current step s writes to memory location M, we check race against M's $writer$ and all recorded steps in M's $reader\_list$. If no race is reported, we clear the $reader\_list$ and update M's $writer$ to be the current step.

Reachability query: Figure 9 elucidates how we perform reachability checks in DRDP. Given two step nodes a, b, we first examine if a precedes b by inspecting DPST. If $lca\_lc(a, b)$ is not an `async` node, we return true; this is reflected in lines 3-5. Next, we check if the previous task is still active starting from line 7. If a's task is still active, either b is a's descendant, or a's task sets the promise that b depends on; both cases indicate $a \rightsquigarrow b$. The remaining code in Figure 9 conducts a breadth-first search on the computation graph based on fields nt and lsa. Lines 23-39 search through non-tree joins. Lines 41 to 57 search through lsa's non-tree joins. We return false if there are no more step nodes to search.

▶ **Theorem 12.** *DRDP's race detection algorithm is sound and precise. For a program P, if no execution with input ψ has any determinacy race, DRDP will not report any race (sound). For a program P, if any execution with input ψ has a determinacy race, DRDP will report the race (precise).*

Proof. To relate the theoretical result given by Theorem 11, the input ψ here refers to the initial program state root(P). The proof is presented in Appendix A. ◀

▶ **Theorem 13.** *Given a parallel program consisting of async, finish and promise that runs in time T_1 on one worker. Assume it creates P promises and Q async tasks. Let H be the height of DPST at the end of the execution, and m be the number of non-tree joins. DRDP can be implemented to check this program for determinacy races in $O(T_1 * Q * H * m * \alpha(T_1, P + Q))$ time, where α is the inverse Ackermann function.*

Proof. For a single run of PRECEDE, it could take up to $O(H * m * \alpha(T_1, P + Q))$ in the worst case if it checks all non-tree joins, and each check contains one DPST traversal plus a disjoint set operation. The maximum count of disjoint sets is $(P + Q)$ because we create one set for each task and one set for each empty task.

The PRECEDE routine may be called Q times for each shadow memory location access. This is because the reader list for a single memory location can be as large as size Q if we save all the tasks. We have at most T_1 shared memory access; thus, with DRDP checking races, the original program can be finished in $O(T_1 * Q * H * m * \alpha(T_1, P + Q))$ time. ◀

4.4 Optimizations

We introduce two optimization techniques used for DRDP. The impacts of these optimizations are respectively evaluated in Sections 5.5 and 5.6.

4.4.1 Adaptive Selection of Graph Traversal Order

A complete computation graph traversal can become necessary without restricted graph structures that enable fast encoding and checking reachability. To accelerate this part, how to traverse all non-tree joins and lsa is the key. We optimize the graph traversal in two ways: first, rather than conducting a depth-first search starting from the current node, we apply a breadth-first search instead; second, when iterating through non-tree joins, we start from the latest join to the oldest. The impact of the proper selection of these two choices is evaluated in Section 5.5.

4.4.2 Redundant Check Elimination

A single step node may access the same memory location multiple times. This may introduce a substantial amount of unnecessary duplicate checks. We present the performance improvement by skipping these redundant checks in Section 5.6. Here we introduce our approach based on the polyhedral model, a powerful linear algebraic framework for affine program analysis, transformations, and code generation.

```
Input: step nodes a, b
Procedure PRECEDE(a, b)
    $S_B = set[b.task\_id]$
    if $lca\_lc(a, b).type \neq ASYNC$ then
        return true
    end

    if $task[a.task\_id].state == ACTIVE$ then
        return true
    end

    // breadth-first search the computation graph via nt and lsa
    $visited = set()$
    $nt - steps = deque()$
    $lsa - sets = deque()$
    foreach $t$ in $S_B.nt$ do
        $nt - steps.push\_back(task[t.task\_id].last\_step\_node)$
    end

    if $S_B.lsa \neq NULL$ then
        $lsa - sets.push\_back(S_B.lsa)$
    end

    while true do
        while $nt - steps.size > 0$ do
            $step = nt - steps.pop\_front()$
            if $lca\_lc(a, step).type \neq ASYNC$ then
                return true
            end

            $visited.insert(step.task\_id)$
            $S_{step} = set[step.task\_id]$
            foreach $t$ in $S_{step}.nt$ do
                if $t.task\_id$ not in $visited$ then
                    $nt - steps.push\_back(task[t.task\_id].last\_step\_node)$
                    $visited.insert(t.task\_id)$
                end
            end

            add $S_{step}.lsa$ to $lsa - sets$ if exists
        end

        while $lsa - sets.size > 0$ do
            $lsa = lsa - sets.pop\_front()$
            $S_{lsa} = set[lsa.task\_id]$
            add $S_{lsa}.lsa$ to $lsa - sets$ if exists
            foreach $t$ in $lsa.nts$ do
                $taskt = task[t.task\_id]$
                if $taskt.id$ in $visited$ then
                    Continue
                end
                if $lca\_lc(a, taskt.last\_step\_node) \neq ASYNC$ then
                    return true
                end
                add $set[taskt.id].nt$ to $nt - steps$
                add $set[taskt.id].lsa$ to $lsa - sets$ if exists
                $visited.insert(taskt.id)$
            end
        end

        if $nt - steps.size == 0$ then
            return false
        end
    end
```

Figure 9 Reachability Check.

```
/* Input code */
for(int i = 0; i < n; i++)
  for(int j = 0; j < m; j++)
    for(int k = 0; k < l; k++)
S:    C[i, j] += A[i, k] * B[k, j];
```

```
/* Code to scan written elements */
if (l >= 1)
  for (int c1 = 0; c1 < n; c1 += 1)
    for (int c2 = 0; c2 < m; c2 += 1)
      write(C[c1, c2]);
```

```
/* Code to scan read elements */
if (l >= 1)
  for (int c1 = 0; c1 < n; c1 += 1)
    for (int c2 = 0; c2 < m; c2 += 1)
      read(C[c1, c2]);
if (m >= 1)
  for (int c1 = 0; c1 < n; c1 += 1)
    for (int c2 = 0; c2 < l; c2 += 1)
      read(A[c1, c2]);
if (n >= 1)
  for (int c1 = 0; c1 < l; c1 += 1)
    for (int c2 = 0; c2 < m; c2 += 1)
      read(B[c1, c2]);
```

Figure 10 Matmul input (left), loops to scan written elements (center), and read elements (right).

When the code region of interest are composed of *affine loops* – i.e., their loop bounds and array accesses are affine combinations of symbolic constants and outer loop iterators, that region is converted into SCoP format [3]. This format precisely specifies the set of read/written elements in the region via affine mapping representation. As an example shown in Figure 10, the SCoP representation of matmul after delinearization [18] is:

$$\begin{aligned} Domain_S &= \{S(i,j,k) \ \mid \ 0 \le i < n \ \wedge \ 0 \le j < m \ \wedge \ 0 \le k < l\} \\ Write_S &= \{S(i,j,k) \to C[i,j]\} \\ Read_S &= \{S(i,j,k) \to C[i,j];\ S(i,j,k) \to A[i,k];\ S(i,j,k) \to B[k,j]\} \end{aligned}$$

$Domain_S$ is the iteration space of statement S while $Write_S$ and $Read_S$ are respectively the mappings from statement instance $S(i,j,k)$ to the written and read elements of arrays A, B, C. The set of elements that are written/read by statement S is computed as the projection of $Domain_S$ via $Write_S/Read_S$ mapping.

$$\begin{aligned} Write_S(Domain_S) &= \{C[i,j] \ \mid \ 0 \le i < n \ \wedge \ 0 \le j < m\} \\ Read_S(Domain_S) &= \{C[i,j] \mid 0 \le i < n \wedge 0 \le j < m; A[i,k] \mid 0 \le i < n \wedge 0 \le k < l; \\ &\qquad B[k,j] \mid 0 \le k < l \wedge 0 \le j < m\} \end{aligned}$$

Using the above written & read element sets, as with the abstract memory layout [34] as scanning order, the loop nests that scan all the written and read elements are generated by the polyhedral code generation method [2].

$$\begin{aligned} Layout &= \{C[c_1,c_2] \to (0,c_1,c_2);\ A[c_1,c_2] \to (1,c_1,c_2);\ B[c_1,c_2] \to (2,c_1,c_2);\ \} \\ Code_{write} &= codegen(Layout \cdot Write_S(Domain_S)) \\ Code_{read} &= codegen(Layout \cdot Read_S(Domain_S)) \end{aligned}$$

This polyhedral optimization phase has been implemented as a source-to-source transformation tool using PET [40] and ISL [39], integrated in the overall LLVM-based instrumentation pass (Section 5.1). The LLVM transformation pass first identifies the SCoP-convertible code regions and outputs them as sequential C code with SCoP annotations. Given SCoP region, the polyhedral phase computes the exact sets of read/written array elements and generates the loops that scan all elements only once. Finally, the output scanning loops are fed back to the LLVM instrumentation as the optimized code after array-based redundant check elimination. The code fragments generated by our polyhedral phase are shown in Figure 10 (center and right).

5 Evaluation

In this section, we evaluate a prototype implementation of the DRDP algorithm to address the following research questions:

1. Correctness (Section 5.2). How does DRDP compare with state-of-the-art data race detectors with respect to false positives and false negatives?
2. Performance (Section 5.4). How does DRDP perform in practice, and how does its time and space performance depend on dynamic characteristics of task-parallel programs with promises (DPST height, number of reads/writes, number of join operations, number of tasks created)?
3. What is the impact of graph traversal order on performance? (Section 5.5).
4. What is the impact of redundant check elimination on performance? (Section 5.6).

5.1 DRDP Implementation

We have implemented DRDP in a prototype race detector for task-parallel programs written in Habanero-C/C++Library (HCLIB). Notice that any parallel program with promises that exhibits a determinacy race in the HCLIB version will also exhibit the race if rewritten with C++ promises. Our prototype can be downloaded here[4], which includes

a) an LLVM transform pass for instrumentation, and
b) a C++ library for dynamic analysis.

The instrumentation pass is executed along with LLVM. When compiling a task-parallel program using the Clang/LLVM compiler, the instrumentation pass will inject a call into the dynamic analysis library after each read and write operation. The library also adds hooks into runtime to capture the invocations of HCLIB constructs (async, finish, promise). The library applies a direct-mapping shadow memory implementation [46]. A contiguous memory region shadows the entire address space, and each memory location's shadow memory cell can be efficiently located using pointer arithmetic.

5.2 Correctness Evaluation

DataRaceBench [24] is a micro-benchmark suite designed to gauge the effectiveness of OpenMP data race detectors. In the latest 1.4.0 version, there are 181 C/C++ micro-benchmarks that cover the majority of OpenMP constructs. We leveraged DataRaceBench to conduct a correctness evaluation for DRDP. We found that micro-benchmarks using OpenMP tasking constructs can be transformed into equivalent HCLIB programs, and task dependencies specified by the `depend` clause can be achieved using promises. Therefore, we picked up all C/C++ micro-benchmarks containing the `depend` clause except DRB135 and DRB136. These two micro-benchmarks combine mutexes with the `depend` clause, which go beyond the set of programs captured by TP3 and also do not satisfy the determinacy property of TP3.

The evaluation results for these micro-benchmarks are shown in Table 1. The "yes/no" suffix in the benchmark name indicates whether the benchmark has a known data race. For short, we refer to these two groups of benchmarks as *yes-benchmarks* and *no-benchmarks*. The evaluation results are described using four terms in Table 1. FP and FN stand for *false positive* and *false negative*, respectively. A false-positive result means the race detector

[4] `https://github.com/FeiyangJin/hclib/tree/ecoop`

reports false alarms on a no-benchmark. A false-negative result means the race detector misses potential races on a yes-benchmark. The other two terms, TP and TN, stand for *true positive* and *true negative*. TP/TN indicates that the race detector generates the expected result on a yes-benchmark/no-benchmark.

Evaluation results are shown for five state-of-the-art data-race detection tools, followed by our work (DRDP). The results for the five other tools were downloaded from the DataRaceBench GitHub repository [25] and evaluated on the OpenMP versions of the micro-benchmarks; these results were obtained in 2021 by the authors of DataRaceBench. All five tools were evaluated with the number of worker threads set to 8. Among the five tools, Intel Inspector [21], ThreadSanitizer [33], and ROMP [19] are dynamic race detectors, while Coderrect [36] and LLOV [5] are static race detectors. The DRDP results were obtained by converting the OpenMP benchmarks to HCLIB task-parallel programs with promises. From the results, we observe that DRDP is the only tool that does not report any false-positive or false-negative results for these benchmarks.

We were unable to identify the root cause of false-positive and false-negative results for ROMP and Intel Inspector since their papers did not provide sufficient information on how their dynamic analysis supports the `depend` clause. For ThreadSanitizer, the documentation states that it applies a fixed-size shadow cell to each memory location. When the shadow cell is full, ThreadSanitizer will randomly discard a recorded memory access to reserve space for the latest one. As a result, ThreadSanitizer may miss data races if one of the involved memory accesses has been discarded from the shadow cell thereby leading to false negatives. For Coderrect and LLOV, it appears that their support for tasking constructs and the `depend` clause is still under development which may explain the false-negative results. We would also expect false-positive results from these static analysis tools, when evaluated on larger benchmarks.

Apart from these converted micro-benchmarks from DataRaceBench, we also wrote some additional tests to check DRDP's implementation. All converted micro-benchmarks and additional tests are included in our code repository.

Table 1 Correctness evaluation results on DataRaceBench.

Benchmarks	Has Race?	Intel Inspector	ThreadSanitizer	ROMP	Coderrect	LLOV	DRDP
DRB027-taskdependmissing-orig-yes.c	Yes	TP	TP	TP	TP	**FN**	TP
DRB072-taskdep1-orig-no.c	No	TN	TN	TN	TN	TN	TN
DRB078-taskdep2-orig-no.c	No	TN	TN	TN	TN	TN	TN
DRB079-taskdep3-orig-no.c	No	TN	TN	TN	TN	TN	TN
DRB131-taskdep4-orig-omp45-yes.c	Yes	TP	**FN**	**FN**	TP	**FN**	TP
DRB132-taskdep4-orig-omp45-no.c	No	**FP**	TN	TN	TN	TN	TN
DRB133-taskdep5-orig-omp45-no.c	No	**FP**	TN	TN	TN	TN	TN
DRB134-taskdep5-orig-omp45-yes.c	Yes	TP	TP	**FN**	**FN**	**FN**	TP
DRB173-non-sibling-taskdep-yes.c	Yes	TP	**FN**	TP	**FN**	**FN**	TP
DRB174-non-sibling-taskdep-no.c	No	TN	TN	**FP**	TN	TN	TN
DRB175-non-sibling-taskdep2-yes.c	Yes	TP	TP	**FN**	**FN**	**FN**	TP
DRB176-fib-taskdep-no.c	No	**FP**	TN	TN	TN	TN	TN
DRB177-fib-taskdep-yes.c	Yes	TP	TP	**FN**	**FN**	**FN**	TP

5.3 Performance Evaluation Benchmarks and Setup

Since we could not easily locate an existing set of performance benchmarks for task-parallel programs using promises, we assembled a suite of seven benchmarks from other benchmark sets as follows. We did not convert the program with the largest lines of code for each set.

[4] `https://github.com/LLNL/dataracebench/wiki/Tool-Evaluation-Dashboard`

We first converted the future-based matmul, sort and strassen (shared by Kastors) programs from [43] (originally from the Rodinia suite [9]) to use promises. Next, we examined the Kastors benchmark suite [41] for OpenMP task dependencies and converted two (sparselu, poisson) to use promises instead to implement the same dependencies. Finally, we converted two task-parallel OpenMP programs from the BOTS benchmark suite(health, knapsack) [14]. We only convert two because race detection on pure task-parallel programs has been widely studied before.

The summary of each benchmark is as follows:

- **matmul**: multiplies two matrices of size 2048 * 2048 with base case size 64 * 64.
- **sort**: sorts an array of size 10000000.
- **strassen**: multiplies two matrices of size 2048 * 2048 by Strassen algorithm.
- **poisson**: solves the Poisson equation (aka jacobi iteration) on the unit square. The parameters we have are 8192,128,3 for matrix size, block size and number of iterations.
- **sparselu**: computes the LU decomposition of a sparse matrix. The parameters we have are 128 and 32 for matrix size and submatrix size.
- **knapsack**: calculates the solution of the knapsack problem with 40 items as input.
- **health**: simulates the Colombian health care system. We estimate the running time for the small model input file given in the source.

The evaluation was conducted on a single-node AMD server machine consisting of a 12-core Ryzen9 3900X running at 3.8GHZ with 128GB memory. All benchmarks were compiled using -O3 optimizations by Clang/LLVM 14.0.0 running on Ubuntu 18.04.05. We report each benchmark's mean execution time and memory usage of 5 runs for *base* and *rd* configurations. "base" is the program running time without race detection; "rd" is the one with full race detection. The standard deviations for both configurations are within 5%.

5.4 Performance Evaluation Results

The results of our evaluation are shown in Table 2; the corresponding time overheads are in Figure 11. The first four columns show the running time in seconds and memory usage in GB, for the two configurations mentioned above. The next column "H" shows the height of DPST. Columns "Check Write" and "Check Read" are the numbers of shared memory access conducted for write and read. The following two columns, "Tree Join" and "NT Join", are the numbers of tree and non-tree joins in the programs. Finally, column "Task" contains the number of dynamic tasks created during program execution.

Table 2 DRDP performance and statistics.

Bench	Base Time	RD Time	Base Mem	RD Mem	H	Check Write	Check Read	Tree Join	NT Join	Task
health	1.51	31.99	0.45	11.64	5	49554260	84975199	2253510	0	2253511
knapsack	1.61	8.00	0.38	4.40	37	3939935	19699117	1969965	0	1969966
matmul	2.52	51.84	0.06	13.44	7	134231771	268477586	28086	0	28087
sort	0.99	40.95	0.18	4.91	15	207671935	261760850	118205	0	118206
strassen	1.37	39.97	0.20	26.12	12	123772570	243307611	29182	16380	45563
sparselu	2.46	69.01	0.04	16.70	4	187379732	362287128	12416	357760	195521
poisson	1.56	71.92	0.80	20.09	4	251658242	654802935	0	122870	106497

From the experiment statistics, we can make several observations. First, the sum of joins for sparselu and poisson surpasses the number of tasks; the reason is that each task may set and get more than one promise. This pattern is not achievable by pure task-parallel programs. Even the future construct cannot produce more synchronization than the task number because creating a future requires creating a new task (who is responsible for setting the future). Moreover, from Table 2 and Figure 11, the race detection time overheads increase with the complexity of the parallelism pattern of the programs. Strassen, sparselu and poisson, with relatively high overheads, use promises without restrictions. The computation graphs created by the three are much more complex than the others, which is reflected by the number of non-tree joins generated.

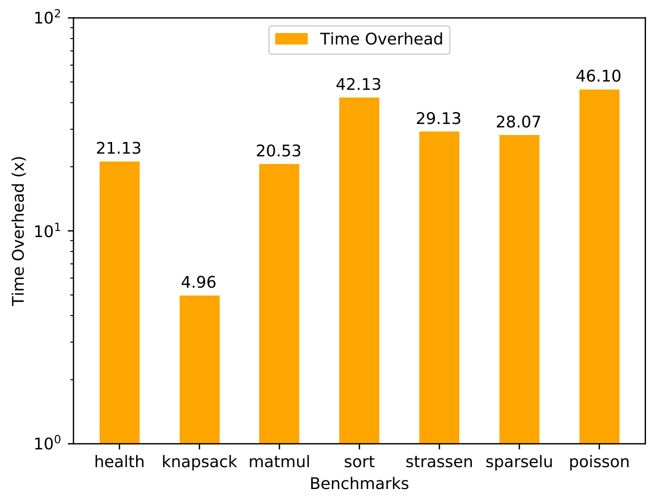


Figure 11 DRDP time overhead.

5.5 Performance Optimization: Graph Traversal Order

The motivation for selection between depth-first search (dfs) and breadth-first search (bfs) is summarized in Table 3, which shows traversal counts by different combinations of graph order (dfs vs. bfs) and iterating order through non-tree joins (back-to-front vs. front-to-back). Our optimization is extremely helpful: we always get the smallest values for columns "Max Task Visited" and "Average Task Visited" when doing bfs on the computation graph and iterating non-tree joins back-to-front. The rows with these two choices are marked in yellow in Table 3. The column "Max Task Visited" records the max number of tasks traversed in any run of the reachability check. The column "Average Task Visited" is the average number of tasks visited for each reachability check that does the graph traversal part. We picked bfs and back-to-front approach in our race detector and evaluated all benchmarks on them.

5.6 Performance Optimization: Redundant Check Elimination

Another crucial observation is that duplicate read and write checks in the same step are the bottlenecks for four benchmarks (matmul, strassen sparselu and poisson). The results in Section 5.4 included the optimized performance for these four benchmarks. In this section, we study the performance impact of the Redundant Check Elimination optimization for these four benchmarks; currently this optimization had no impact on the remaining three benchmarks evaluated in Section 5.4 because they would need interprocedural analysis across recursive calls, which is currently not performed by our optimizing compiler.

Table 3 Graph traversal order comparison.

Bench	Graph Order	Input	NT Order	Min Visited Task	Max Visited Task	Average Visited Task
strassen	dfs	512 16	back to front	1	2	1.43
strassen	dfs	512 16	front to back	1	337	11.04
strassen	bfs	512 16	back to front	1	1	1.00
strassen	bfs	512 16	front to back	1	4	1.13
sparselu	dfs	32 8	back to front	1	544	8.85
sparselu	dfs	32 8	front to back	1	544	65.72
sparselu	bfs	32 8	back to front	1	2	1.07
sparselu	bfs	32 8	front to back	1	17	6.08
poisson	dfs	2048 128 3	back to front	1	32	9.54
poisson	dfs	2048 128 3	front to back	1	32	9.54
poisson	bfs	2048 128 3	back to front	1	6	2.84
poisson	bfs	2048 128 3	front to back	1	6	2.84

We record runtime statistics for these programs with or without reducing redundant checks. To measure the time overhead and memory usage, we have to re-evaluate these benchmarks on a different machine with enough memory. We present the outcomes in Table 4. The parentheses after some data show the increase/decrease percentage.

Table 4 Performance comparison for reducing redundant checks.

Bench	Reduce Checks	RD Time	RD Mem	Input	Reachability Check	Write Check	Read Check
matmul	No	118.95 (+1038%)	74.77 (+3805%)	1024	1.57E+07 (+0%)	1.68E+07 (+0%)	2.15E+09 (+6299%)
matmul	Yes	10.45	1.91	1024	1.57E+07	1.68E+07	3.36E+07
strassen	No	19.73 (+233%)	12.88 (+584%)	512 16	1.94E+08 (+496%)	4.10E+06 (-47%)	7.74E+07 (+408%)
strassen	Yes	5.93	1.88	512 16	3.26E+07	7.77E+06	1.52E+07
poisson	No	214.85 (+135%)	37.87 (+129%)	7424 128 3	1.93E+09 (+84%)	2.07E+08 (-0.02%)	1.36E+09 (+154%)
poisson	Yes	91.49	16.56	7424 128 3	1.05E+09	2.07E+08	5.38E+08
sparselu	No	1137.68 (+918%)	382.37 (+2189%)	128 32	9.94E+08 (+84%)	5.86E+09 (+3029%)	1.20E+10 (+3219%)
sparselu	Yes	111.74	16.70	128 32	5.41E+08	1.87E+08	3.62E+08

All four benchmarks (matmul, strassen, poisson and sparselu) are matrix-based. As explained in Section 4.4.2, a considerable amount of redundant read checks can be eliminated in such cases. After the transformation, as shown in Table 4, the running time increases by 135% to 1038% when comparing the optimized version with the unoptimized version.

5.7 Comparison with ThreadSanitizer

Directly evaluating other tools on our benchmarks will typically generate false positives because they do not consider synchronization constraints imposed by promises. Nevertheless, given the widespread use of ThreadSanitizer in practice, we decided to evaluate it on one of our benchmarks (matmul), which is the only one for which ThreadSanitizer was able to complete successfully and that too with a smaller input. We set "report_bugs=0, ignore_uninstrumented_modules=1" for ThreadSanitizer to ensure that it ignores uninstrumented code and does not print a race report.

The issue we encountered is that ThreadSanitizer crashes on large inputs for our promise-based applications. The error messages from these crashes report a stack overflow. Most likely, it was caused by the fixed-size stack ThreadSanitizer set[5], especially since all of our benchmarks use recursion. We use a smaller input size (128 x 128 for the matmul benchmark) instead to perform the evaluation.

[5] `https://github.com/llvm/llvm-project/blob/2e999b7dd1934a44d38c3a753460f1e5a217e9a5/compiler-rt/lib/tsan/rtl/tsan_platform_posix.cpp#L53`

Table 5 Performance comparison between DRDP and ThreadSanitizer.

Tool	Threads	Time Overhead	Memory Overhead	Time (ms)	Memory (MB)
ThreadSanitizer	4	20.26×	34.91×	45.8	180.49
Baseline	4	1.00×	1.00×	2.3	5.17
DRDP	1	6.10×	33.65×	17.6	166.31
Baseline	1	1.00×	1.00×	2.9	4.94

This benchmark generates 7 tasks at runtime for the given input size, so we reasonably chose to execute it with 4 worker threads for the ThreadSanitizer case. From the results in Table 5, we can see that when the input size is small DRDP has a better time performance and a similar memory performance compared with ThreadSanitizer. This is even though the ThreadSanitizer execution uses 4 threads, and the DRDP execution uses 1 thread. (The Baseline measurements were obtained on the same original version of the benchmark, but with 4 threads and 1 thread respectively.)

6 Related Work

The concept of determinacy was introduced by Karp and Miller in the late 1960s [23]. More recently, Dennis et al. reviewed this past work and introduced the concepts of structural and functional determinism [13]. Related work on the Habanero-Java programming model [7] classified task-parallel programs into seven categories where each category satisfies or does not satisfy certain properties, such as data race freedom, deadlock-freedom or determinism. The paper observed that for some parallel constructs, data race freedom implies determinacy but no proof was given for that claim. Concurrent Collection (CnC) is a dataflow-based coordination language, which was proved to be determinate [6]. Data-race free GPU programs that use barriers for synchronization have been proven to be determinate [10], though the programming model is data parallel rather than task parallel. Similarly, programs designed for heterogeneous systems can achieve portability (same input, same result regardless of the specific backend used) if data-race free [22].

There has been a long history of dynamic determinacy race detection algorithms and tools based on vector clocks [16, 21, 33]. A major advantage of the vector clock approach is that it can be applied to parallel programs with arbitrary parallel constructs, including locks and transactions (say). However, its major limitation when applied to task-parallel programs is that it can only provide guarantees on a per-schedule (rather than per-input) basis since it is not practical for vector clock sizes to be proportional to the number of active tasks.

The serial projection property has been used in past work to perform per-input race detection via sequential execution for restricted classes of task-parallel programs [4, 30]. These algorithms take advantage of the structural property of computation graphs for fork-join programs, but they do not support the arbitrary computation graphs that can be generated by task-parallel programs with promises. Surendran and Sarkar also leveraged the serial projection property to devise the first per-input dynamic race detector for task-parallel programs with futures [35]. The futureRD race detector from Utterback et al. [37] supports a restricted class of futures: it does not allow multiple `get` operations on a future handle. In contrast to previous algorithms, DRDP can support general blocking operations in task-parallel programs with promises. It also illustrates how dynamic race detection can be performed via sequential execution for task-parallel programs that do not satisfy the serial projection property.

Labeling techniques have also been used in past work on race detection for task parallelism. This approach enables reachability checking between two nodes by comparing two labels. Mellor-Crummey introduced the Offset-Span [26] algorithm as one such approach, in which the length of the label attached to each task can grow as large as the depth of nested fork structures. The SP-Bags [15] structure devised by Feng and Leiserson, and the ESP-Bags [30] introduced by Raman et al. are also examples of using labeling to record happens-before relationships.

More recently, there have been some new results on performing per-input dynamic race detection when executing programs in parallel instead of sequentially. In 2012, Raman et al. introduced the DPST data structure [31], which runs in parallel and efficiently tracks happens-before relationships of async-finish constructs. An application of DPST targets task parallelism in OpenMP has been proposed [45]. Utterback et al. [38] proposed an asymptotically optimized parallel race detection algorithm for fork-join programs. More recently, Xu et al. [43] introduced the first known parallel dynamic race detector for task-parallel programs with futures. However, none of these prior works support per-input determinacy race detection of task-parallel programs with promises.

7 Conclusions

In this paper, we addressed the problem of dynamic detection of data-races and determinacy-races in Task-Parallel Programs with Promises (TP3), which support more flexible synchronization patterns than fork-join constructs and futures. We first introduced a featherweight programming language that captures the semantics of TP3 and provides a basis for formally defining determinacy using our semantics. This definition subsumes functional determinacy (same output for same input) and structural determinacy (same computation graph for same input). We also introduced DRDP, the first-known per-input dynamic determinacy race detector algorithm for TP3, and demonstrated that it is practical to implement. To the best of our knowledge, DRDP is the first race detector that executes a task-parallel program sequentially without requiring the serial-projection property, which is a critical requirement for TP3 in general. The execution time slowdowns are all under 50×, which is comparable to overheads incurred by other dynamic race detection and debugging tools used in practice. The results also highlighted the impact of two important optimizations, traversal order and redundant check elimination, in obtaining these results. Opportunities for future work include exploring static and dynamic optimizations to further reduce the overheads in our implementation of the DRDP, as well as extensions to support determinacy race detection for promise-like constructs used in heterogeneous parallelism (e.g., CUDA graph) and in distributed-memory parallelism (e.g., MPI_Request).

References

1 Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H Ahn, Ignacio Laguna, Martin Schulz, Gregory L Lee, Joachim Protze, and Matthias S Müller. Archer: effectively spotting data races in large openmp applications. In *2016 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 53–62. IEEE, 2016.

2 Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, USA, 2004. IEEE Computer Society.

3 Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In Rajiv Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2010. URL: `http://dblp.uni-trier.de/db/conf/cc/cc2010.html#BenabderrahmanePCB10`.

4 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1007912.1007933.

5 Utpal Bora, Santanu Das, Pankaj Kukreja, Saurabh Joshi, Ramakrishna Upadrasta, and Sanjay Rajopadhye. Llov: A fast static data-race checker for openmp programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26, 2020.

6 Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. Concurrent collections. *Scientific Programming*, 18:203–217, 2010. 3-4. doi:10.3233/SPR-2011-0305.

7 Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61, 2011.

8 Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with mpi. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 712–725, 2013. doi:10.1109/IPDPS.2013.78.

9 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.

10 Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 397–409, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535882.

11 LLVM Community. Tsan predefined constants for vector clocks, May 2022. URL: https://github.com/lechenyu/llvm-project/blob/main/compiler-rt/lib/tsan/rtl/tsan_defs.h.

12 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

13 Jack B Dennis, Guang R Gao, and Vivek Sarkar. Determinacy and repeatability of parallel program schemata. In *2012 Data-Flow Execution Models for Extreme Scale Computing*, pages 1–9. IEEE, 2012.

14 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, USA, 2009. IEEE Computer Society. doi:10.1109/ICPP.2009.64.

15 Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258492.258493.

16 Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. *ACM Sigplan Notices*, 44(6):121–133, 2009.

17 Standard C++ Foundation. C++11 Standard Library Extensions — Concurrency, May 2021. URL: https://isocpp.org/wiki/faq/cpp11-library-concurrency.

18 Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 351–360, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2751205.2751248.

19 Yizi Gu and John Mellor-Crummey. Dynamic data race detection for openmp programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 767–778. IEEE, 2018.

20 Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985. `doi:10.1145/4472.4478`.

21 Intel. Intel inspector, May 2021. URL: `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html#gs.1wvmbu`.

22 Feiyang Jin, John Jacobson, Samuel D. Pollard, and Vivek Sarkar. Minikokkos: A calculus of portable parallelism. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 37–44, 2022. `doi:10.1109/Correctness56720.2022.00010`.

23 Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. `doi:10.1016/S0022-0000(69)80011-5`.

24 Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3126908.3126958`.

25 LLNL. C cpp details oct 2021, October 2021. URL: `https://github.com/LLNL/dataracebench/wiki/Tool-Evaluation-Dashboard`.

26 John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 24–33, New York, NY, USA, 1991. Association for Computing Machinery. `doi:10.1145/125826.125861`.

27 Mozilla Developer Network. Javascript reference – Promise, July 2021. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`.

28 Oracle. Java® Platform, Standard Edition and Java Development Kit Version 17 API Specification - CompletableFuture, October 2021. URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CompletableFuture.html`.

29 Joachim Protze, Martin Schulz, Dong H Ahn, and Matthias S Müller. Thread-local concurrency: a technique to handle data race detection at programming model abstraction. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 144–155, 2018.

30 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design*, 41(3):321–347, December 2012. `doi:10.1007/s10703-012-0143-7`.

31 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2254064.2254127`.

32 Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation. *ACM Trans. Parallel Comput.*, 6(4), December 2019. `doi:10.1145/3365655`.

33 Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

34 Jun Shirako and Vivek Sarkar. Integrating Data Layout Transformations with the Polyhedral Model. In *Proc. of IMPACT 2019*, 2019.

35 Rishi Surendran and Vivek Sarkar. Dynamic determinacy race detection for task parallelism with futures. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 368–385, Cham, 2016. Springer International Publishing.

36 Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. Ompracer: A scalable and precise static race detector for openmp programs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

37 Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. Efficient race detection with futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 340–354, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3293883.3295732`.

38 Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 83–94, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2935764.2935801`.

39 Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software – ICMS 2010*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010.

40 Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, January 2012.

41 Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of openmp dependent tasks with the kastors benchmark suite. In Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, pages 16–29, Cham, 2014. Springer International Publishing.

42 Caleb Voss and Vivek Sarkar. An ownership policy and deadlock detector for promises. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 348–361, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3437801.3441616`.

43 Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. Parallel determinacy race detection for futures. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 217–231, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3332466.3374536`.

44 Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. Parallel data race detection for task parallel programs with locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 833–845, 2016.

45 Lechen Yu, Feiyang Jin, Joachim Protze, and Vivek Sarkar. Leveraging the dynamic program structure tree to detect data races in openmp programs. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 54–62, 2022. `doi:10.1109/Correctness56720.2022.00012`.

46 Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proceedings of the 2010 international symposium on Memory management*, pages 93–102, 2010.

A Proof

▶ **Lemma 14.** *Non-tree joins (nt) and least-significant ancestor (lsa) maintain happens-before relationships correctly, given the routines in Figure 8.*

Proof. For a task T in the disjoint-set D, there are two conditions we may update D's nt according to Figure 8.

- Lines 19 to 25: when task T gets a promise, we add the empty task created by the promise setter to D's nt. In this way, we create dependency from step nodes (those before the set operation) in the setter task to the remaining step nodes in T.
- Line 26 to 38: when finish F ends in task T, we merge all non-tree joins from merged tasks. The merged tasks are essentially descendants of T and we keep those non-tree joins in D's nt.

For a task T in the disjoint-set D, there are two conditions we may update D's lsa according to Figure 8.

- Lines 1 to 12: when task T is created, if the parent has non-tree joins, we set D's lsa to be the parent. Otherwise, we set D's lsa to be the lsa of parent. This obeys the definition of lsa (the lowest ancestor that has non-tree joins).
- Line 26 to 38: when finish F ends in task T, we keep T's lsa and ignore merged tasks' lsa. This is valid because merged tasks are essentially descendants of T, so for any merged task, its lsa is either T, T's lsa or one of T's descendants. In all cases, the nt and lsa of D already cover the information. ◀

▶ **Theorem 15.** *If DRDP does not report any determinacy race during an execution of program P with input ψ, then no execution of P with ψ will have a write-write race on any memory location r.*

Proof. Consider an execution δ of a program P with input ψ in which DRDP is enabled and does not report any determinacy race.

Suppose that a write-write race, χ, occurs on a memory location r in some execution δ' of P with ψ. Let W_1 and W_2 denote the two steps that write to r resulting in the race in δ'. Note that the execution δ' does not have any race until χ occurs. Without loss of generality, assume W_1 writes to r first and W_2 writes to r later in δ. We will prove by contradiction that DRDP must report the race in δ. There are two cases:

1. There are no writes to r between W_1 and W_2 in δ.
   When W_1 occurs in δ', Figure 8 lines 45 - 55 check all readers in reader list and last writer of r to see if any can execute in parallel with W_1. Because χ is the first determinacy race in δ', no race will be reported when W_1 occurs. We then save W_1 as the last writer to r in δ'.
   When W_1 occurs in δ, because no determinacy race occurs yet, we then save W_1 as the last writer to r in δ.
   When W_2 occurs in δ, we will run $PRECEDE(W_1, W_2)$ to check if $W_1 \rightsquigarrow W_2$. In δ', $PRECEDE(W_1, W_2)$ returns false. In δ, $PRECEDE(W_1, W_2)$ returns true. We are going to show it is impossible in δ that true was returned.
   In Algorithm 9, we may return true in four places: lines 4, 8, 26, and 51.
   Line 4 cannot be executed in δ because DPST is the same across all executions without determinacy race. When W_2 happens, the related DPST parts that were previously generated are the same across δ, δ'. As a result, in δ, DRDP cannot return true at line 4.
   Line 8 cannot be executed in δ. This line only returns true if the task W_1 in is still active in δ at this point. This means either W_2 is W_1's descendant or W_1's task has set a promise that W_2's task needs. Both conditions cannot be true because in δ' we have $W_1 \parallel W_2$.
   Lines 26 and 51 cannot be executed in δ. We may return in these two lines if we find a path from W_1 to W_2 in a graph traversal over the computation graph. The validity is proved in Lemma 14.
2. There are writes to r by steps W_i ... W_j between W_1 and W_2 in δ.
   In δ', the happens-before relationship must be $W_1 \rightsquigarrow W_i \rightsquigarrow W_{i+1} \rightsquigarrow \dots W_{j-1} \rightsquigarrow W_j$ because χ is the first race in δ'. As $W_1 \parallel W_2$, we have $W_j \parallel W_2$, so W_j and W_2 is also a pair of write that leads to a race. Because the related computation graph parts are the same across data-race free execution, the same happens-before relationship exists in δ. When W_2 occurs in δ, the shadow memory has W_j as the last writer. We will run $PRECEDE(W_j, W_2)$ to check if $W_j \rightsquigarrow W_2$. In δ, $PRECEDE(W_j, W_2)$ cannot return true, which can be proved similarly as part 1. This is a contradiction with the statement DRDP does not report any race in δ. ◀

▶ **Theorem 16.** *If DRDP does not report any data race during an execution of program P with input ψ, then no execution of P with ψ will have a read-write determinacy race on any memory location r.*

Proof. Consider an execution δ of a program P with input ψ in which DRDP is enabled and does not report any determinacy race.

Suppose that a read-write race, χ, occurs on a memory location r in some execution δ' of P with ψ. Let R_1 and W_1 denote the two steps that read and write r resulting in the race in δ'. Because this is a read-write data race, R_1 occurs before W_1 in δ'. Note that the execution δ' does not have any race until χ occurs. We will prove by contradiction that DRDP must report the race in δ. There are two cases:

1. R_1 executes before W_1 in δ.
   a. There are no writes of r between R_1 and W_1 in δ.
      When R_1 occurs in δ, we check race with the last write. We then save R_1 to the reader list of r in δ.
      When W_1 occurs in δ, we will run $PRECEDE(R_1, W_1)$ to check if $R_1 \rightsquigarrow W_1$. The call returns returns true in δ. This is impossible. The reasoning is similar to Theorem 15 part 1.
   b. There are writes of r by steps $W_i \dots W_j$ between R_1 and W_1 in δ.
      Theorem 15 states that if there is a write-write race in the program P, DRDP will always report it. This means if there exists a race in any pair of writers in $W_i \dots W_j, W_1$, DRDP must find it. Because in δ execution, DRDP does not report any write-write race, we must have $W_i \rightsquigarrow W_{i+1} \rightsquigarrow \dots \rightsquigarrow W_{j-1} \rightsquigarrow W_j \rightsquigarrow W_1$. This relationship is the same in δ'.
      As a result, we can conclude in δ', we have $R_1 \parallel W_i$, otherwise there cannot be a race between R_1 and W_1 in δ'. In δ, when W_i occurs, DRDP must report the race. The reasoning is similar to part a. This is a contradiction with the statement that DRDP does not report any race in δ.
2. W_1 executes before R_1 in δ.
   a. There are no writes of r between W_1 and R_1 in δ.
      The proof is similar to Theorem 15 part 1. We omit the details.
   b. There are writes of r by steps $W_i \dots W_j$ between W_1 and R_1 in δ.
      The proof is similar to Theorem 15 part 2. We omit the details. ◀

▶ **Theorem 17.** *If DRDP does not report any determinacy race during an execution of program P with input ψ, then no execution of P with ψ will have a write-read determinacy race on any memory location r.*

Proof. We omit the proof because it is similar to proof for Theorem 16. ◀

▶ **Theorem 18.** *If DRDP reports a determinacy race on r during an execution of program P with input ψ, then at least one execution of P with ψ will have this determinacy race on r.*

Proof. In Algorithm 9, we may return true in four places: lines 4, 8, 26, and 51. If DRDP reports a write-write race or a read-write race or a write-read race, we know none of the lines is executed. The validity of the check is explained in Theorem 15 part 1.

From the definition of determinacy race and the fact DRDP reports a race, we know at least one execution of program P will show the race. ◀

▶ **Theorem 19.** *The race detection algorithm described in Figure 8 and 9 is sound and precise.*

Proof. Theorem 15,16,17 show that the algorithm is sound for a given input. Theorem 18 proves that the algorithm is precise for a given input. ◀