Transitive Joins: A Sound and Efficient Online Deadlock-Avoidance Policy

Caleb Voss
Georgia Institute of Technology
cvoss@gatech.edu

Tiago Cogumbreiro
University of Massachusetts Boston
tiago.cogumbreiro@umb.edu

Vivek Sarkar
Georgia Institute of Technology
vsarkar@gatech.edu

Abstract

We introduce a new online deadlock-avoidance policy, Transitive Joins (TJ), that targets programs with dynamic task parallelism and arbitrary join operations. In this model, a computation task can asynchronously spawn new tasks and selectively join (block) on any task for which it has a handle. We prove that TJ soundly guarantees the absence of deadlock cycles among the blocking join operations. We present an algorithm for dynamically verifying TJ and show that TJ results in fewer false positives than the state-of-the-art policy, Known Joins (KJ). We evaluate an implementation of our verifier in comparison to prior work. The evaluation results show that instrumenting a program with a TJ verifier incurs geometric mean overheads of only 1.06× in execution time and 1.09× in memory usage, which is better overall than existing KJ verifiers. TJ is a practical online deadlock-avoidance policy that is applicable to a wide range of parallel programming models.

CCS Concepts • **Software and its engineering → Deadlocks**; **Parallel programming languages**.

Keywords deadlock avoidance, task parallelism, futures

1 Introduction

In parallel programs, a *deadlock* is a bug that arises when a cycle of blocking dependencies forms among a collection of concurrent resources [22]. Deadlocks can be difficult to reason about as the circumstances which cause them involve the ordering of computation steps across multiple concurrently executing tasks, and deadlocks can often arise nondeterministically due to variations in task scheduling. We study the deadlock problem as it applies to the dynamic task parallelism programming model, and we propose a *dynamically verifiable policy* that guarantees *deadlock freedom.*

PPoPP '19, February 16–20, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295724

In this paper, we focus on deadlocks that can arise from join operations in task-parallel programs, that is, operations that wait for task termination. For concreteness, and without loss of generality, we discuss programs in which the eventual result value of each asynchronously executing task is represented by a copyable *Future* object. Any task can perform a blocking get operation on a Future to retrieve the return value once it is available [17]. It is standard practice to refer to this blocking wait as a *join* operation. Accordingly, the creation of a new asynchronously executing task is referred to as a *fork*.

Our modeling of the deadlock-avoidance problem using Futures is applicable to a wide range of parallel programming models. For example, the forking and joining of system-level threads, as in Java and C++, can be modeled abstractly with Futures. Moreover, the concurrency library of Java [15] and, more recently, the standard library of C++ [23] directly include Futures as a high-level abstraction. Cilk includes 'spawn' and 'sync' commands; Cilk's model is more limited than Futures in general because a Cilk function is compelled to join with all the tasks it has spawned [12]. Cilk programs can only exhibit *fully strict* computation graphs [2]. X10, the Habanero Java language (HJ), and the Habanero Java Library (HJlib) include an async-finish model, which, although more general than Cilk, is also more limited than Futures; rather than join with arbitrary tasks, a task can join all at once with the collection of tasks it created (transitively) within a given computation [7]. Programs based on async-finish exhibit *terminally strict* computation graphs [16]. X10, HJ, and HJlib also support arbitrary joins on asynchronously spawned tasks in the form of Futures [6, 7, 21].

Approaches to the deadlock problem include solutions which statically detect the possibility of deadlocks or verify deadlock freedom [4, 27, 34], solutions which detect deadlocked tasks at runtime [20, 25], and solutions which avoid deadlocks by intercepting blocking operations that might cause a deadlock if allowed to proceed [9, 10, 28]. Our work is a deadlock-avoidance policy that gives target programs the ability to handle illegal joins as an exception.

Specifically, this paper makes the following contributions:

1. We formulate a policy called Transitive Joins (TJ) as a simple set of rules restricting which joins are permissible, and we prove that TJ guarantees a program's joins will not deadlock (TJ is sound).

2. We prove that TJ extends an existing policy due to Cogumbreiro et al. [10], called Known Joins (KJ), by admitting a superset of the programs that are valid under KJ. We argue for the utility of the large class of the newly admitted TJ programs.
3. TJ can be used as an online verification algorithm that aborts potentially unsafe joins, raising an exception in the program. A simple algorithm for TJ verification takes $O(h)$ time to check each join and $O(n)$ space in total, where n is the number of tasks spawned, and h is the height of the task fork tree.
4. We implemented a TJ verifier; its evaluation shows a geometric mean overhead of 1.06× execution time and 1.09× memory usage, justifying the use of TJ in practice as an always-on runtime safety check. TJ's overheads improve on the best implementation of KJ, whose time and memory overheads are 1.09× and 1.30×, respectively.

Our Transitive Joins verifier implements one of three proposed algorithms for online policy verification. We evaluate the TJ verifier against two available KJ verifiers on five benchmarks which both TJ and KJ admit as valid. We also include a deadlock-free benchmark which satisfies TJ but not KJ, thereby requiring the KJ verifiers to fall back to a more expensive cycle-detection algorithm. The results show that TJ incurs comparable or better execution time and memory overheads to the KJ implementations, so that it is practical and desirable to use TJ in favor over the KJ verifier and classical cycle-detection.

Outline. This paper opens in Section 2 with an informal description of TJ and explores its utility through example programs. In Section 3 we give formal definitions and theorems on the correctness of TJ, and in Section 4 we formally relate TJ to KJ. Section 5 describes an online TJ verification algorithm, which is evaluated in Section 6. Section 7 covers related work on deadlocks, and Section 8 concludes.

2 Overview

We informally describe our novel Transitive Joins policy and frame it as the transitive closure of two intuitive rules. Through two examples representing practical, realizable program patterns, we argue for the utility of TJ in admitting an additional class of deadlock-free programs over prior work.

2.1 TJ Principles

Because precise cycle detection is slow, we accept the reality of false positives in online deadlock avoidance. So in designing a sound policy (one that forbids all deadlocking runs), it is important to carefully control *which* runs raise false positives. The join operations that are permitted by a sound deadlock-avoidance policy ought to be those which naturally emerge from program structure, for two reasons. 1) It is inconvenient for the programmer if the policy excludes a program whose deadlock freedom is plainly evident. 2) By excluding programs whose deadlock freedom is obscure and hard to understand, we encourage sanitary coding practices.

Guided by program structure, we devise a few principles which determine the join permissions that a given task is granted. First, we observe that, in the Futures model of task parallelism, the continuation of a forking task receives a Future object which refers to the forked task; that is, the parent has a joinable handle to the child. However, the child task does not receive a handle to the parent. Therefore, it is natural to (I) permit the parent to join (block) on the child, but not to permit the child to join on the parent. Second, a forked task receives parameters or captured data from the forking task, which can readily include previously created Futures. Therefore, it is natural to (II) permit a child to join on the tasks for which the parent held join permission at the time of the fork.

Prior work takes rules (I,II) and adds a third rule to create a sound deadlock-avoidance policy called Known Joins (KJ) [10]. A corollary to the soundness of KJ is that rules (I,II) alone are also sound (though very imprecise).

The key observation which leads to our novel policy is the following: Under a sound policy, if task a performs a permitted join on task b, and if task b performs a permitted join on task c, then task a has effectively blocked on task c and yet is not in danger of deadlocking. Therefore, it is natural to declare that (III) permission-to-join should be a transitive relation. We define the Transitive Joins policy (TJ) as the rules (I,II,III).

Figure 1 illustrates two programs and their TJ permissions. First consider the diagram on the left. Task a forks task b, then task d. Task b forks task c. There is no guarantee about whether c or d is created first. Under rule (I), it is always valid for a parent to join on its own child. Therefore, every fork edge is also a join permission edge. By rule (II), task d inherits from a its permission to join on b, since a held this permission at the time d was created. Readers familiar with KJ will see that, after d executes a join on b, d then *learns* KJ permission to join on c. But under rule (III) of TJ, d has permission to join on c by transitivity through b, whether or not d actually joins on b.

The diagram on the right of Figure 1 begins with the same scenario of forks. However, d then forks task e, which joins on c. Task e inherits from its parent the permission to join on b. Under KJ it is not legal for e to directly join on c.[1] By contrast, join permission is transitive in TJ. Since there is a (non-empty) path of join permissions from e to c, e is permitted by TJ to join on e *without* first joining on the other nodes along that path (namely b).

[1]KJ was designed to prove deadlock freedom from data race freedom in the absence of additional synchronization mechanisms. If e obtains a handle to c without synchronization, there must be a data race, so KJ is not interested in admitting this execution.

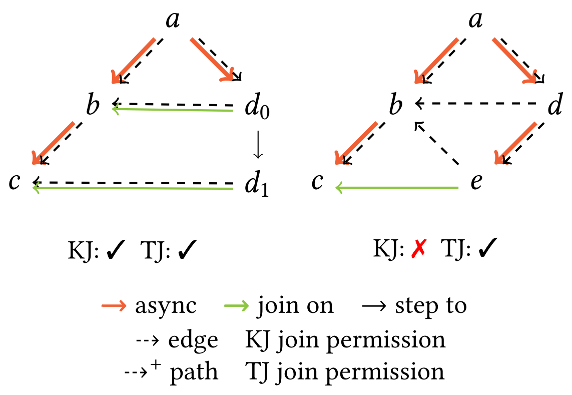

Figure 1. The actions of two example programs, showing the fork tree (orange), the joins (green), and the join permissions (dotted). Children of each node are drawn in fork order from left to right. The run on the right is accepted only by TJ.

Compared to the state of the art, Transitive Joins reduces the gap between what is deadlock-free and what is accepted by a policy. We will show that KJ-valid executions are a strict subset of TJ-valid executions (Theorem 4.3). TJ admits new executions which rely on the transitivity of join permission but which skip joins or perform joins out of order with respect to KJ. Stated altogether as a single principle, TJ allows *any task* in a subtree T_1 of the fork tree to join on *any task* in another subtree T_2, provided the root of T_2 is an older sibling of (same parent, but forked earlier than) the root of T_1. This principle will be formalized as Theorem 3.15.

2.2 Program Model

As introduced in X10 [7] (and more recently in C++ [23]), we use the keyword async to asynchronously execute a block of code (capturing local values, but sharing heap-allocated objects and arrays); async immediately returns a handle to the task, called a Future. Futures have a blocking method, join(), which waits for the associated task to terminate and then returns that task's result value (or null if there is no result). Our examples make use of some standard Java classes and methods. In particular, ConcurrentLinkedQueue is a queue for which concurrent accesses are sequentially ordered, and AtomicReferenceArray is an array whose entries exhibit volatile access.

2.3 Unordered Join on All Descendants

To see the utility of a transitive permission-to-join relation, consider the program given in Listing 1. It consists of a divide-and-conquer routine, f, which we have parallelized by enclosing each recursive call in its own async task on lines 5 and 6. We keep the Future for every task in a shared queue, and the main task awaits the completion of the entire algorithm by joining on each Future in the queue (lines 14–15).

Listing 1. Divide-and-conquer algorithm with no guarantee on the relative order of joins.

```
 1 void f (Queue<Future> tasks) {
 2   if (done()) return 1;
 3   // Child launches before being pushed.
 4   // No parent-child order guarantees.
 5   tasks.add(async { f(tasks); });
 6   tasks.add(async { f(tasks); });
 7 }
 8 void main () {
 9   Queue<Future> tasks =
10       new ConcurrentLinkedQueue<Future>();
11   f(tasks);
12   // May join with any descendant.
13   int result = 0;
14   while (!tasks.isEmpty())
15     result += tasks.poll().join();
16 }
```

The join pattern of this program behaves like an implementation of the finish construct, where main joins on all tasks transitively spawned during its computation.[2] Because a join does not unblock until the joinee terminates, it is guaranteed that once the queue of tasks is found to be empty, no more asynchronous tasks are running.

However, when each new task is created, it may begin executing before or after its Future is placed onto the queue. Therefore, the queue does not respect any ordering between parent and child tasks. Many possible runs of this program violate the Known Joins policy because main may try to join on a task, *a*, before obtaining permission to do so via joining on the parent of *a*. Such a scenario does not, however, violate the Transitive Joins policy since TJ employs a transitive join permission. If main is permitted to join on the parent of *a*, since the parent is permitted, in turn, to join on *a*, then it follows that main is permitted to join on *a* directly. Therefore, main is permitted to join on an arbitrary element of the queue at any time.

2.4 Critical Path Reduction

To further show the utility of the permissions granted by Transitive Joins, we present a program implementing a map-reduce framework. Map-reduce is a common pattern of concurrency in which a large collection of parallel work is distributed among several mapper tasks, later to be accumulated into one result by one or more reducer tasks. We give an example map-reduce program in Listing 2. Lines 5–6 fork N asynchronous mappers, but since these lines of code are themselves asynchronous to main, the program does not wait for all the mappers to be created before continuing. Lines 9–20 fork C asynchronous reducers. Each reducer accumulates

[2]More specifically, the join pattern is a natural way to implement the 'finish accumulator' construct [30], which joins on all tasks that were forked within some scope and collects their results.

Listing 2. Map-reduce program showing another use of Transitive Joins.

```
 1 void main () {
 2   AtomicReferenceArray<Future> mappers =
 3       new AtomicReferenceArray<Future>(N);
 4   async { // Async mapper spawning
 5     for (i = 0; i < N; i++)
 6       mappers.set(i, async { return work(); });
 7   };
 8   // Chunked reduce phase
 9   Future[] reducers = new Future[C];
10   for (c = 0; c < C; c++) {
11       reducers[c] = async {
12           acc = 0;
13           for (i = c*N/C; i < (c+1)*N/C; i++) {
14               while (mappers.get(i) == null)
15                   Thread.yield();
16               acc += mappers.get(i).join();
17           }
18           return acc;
19       };
20   }
21   acc = 0;
22   for (c = 0; c < C; c++)
23       acc += reducers[c].join();
24 }
```

the results of a chunk of N/C mappers in lines 12–18.[3] The spin loop on lines 14–15 is a low-level way for the reducer to wait for each mapper to be placed into the synchronized array; although it is beyond the scope of this work to consider primitives other than Futures, a high-level event-driven primitive could be used instead. Finally, lines 21–23 accumulate the partial results from all reducers.

Listing 2 is deadlock-free, and it is valid under TJ but not under KJ. Observe that the mapper tasks are grandchildren of main, and the reducers inherit main's join-permission relationship to the mappers. According to KJ it is illegal to execute the join() on line 16, unless 1) main joins with the line 4 async task *prior* to forking the reducers, or else 2) each reducer must itself join with the line 4 async. Unlike Listing 1, Listing 2 *always* violates KJ, rather than violating it nondeterministically.

However, under TJ it is legal for the reducers to join with the mappers without any additional requirements. The main task is transitively permitted to join with its grandchildren (the mappers), and the reducers inherit that permission. In this way, the program can begin reducing the results as soon as they arrive, even if they arrive before all the mapper tasks are forked. That is, a KJ-compliant variant of Listing 2 would have a longer critical path than the present code since a join on the line 4 async task would have to be inserted on the critical path.

[3]This pattern is more complex than that of Listing 1 and cannot be written using finish constructs because each reducer selects a subset of mappers to wait on, rather than all of the tasks that were forked within a given scope.

3 Formalism

We formally define our novel Transitive Joins policy and prove two main results: 1) TJ join permission is a total order, so all TJ-valid program runs are deadlock-free (Theorems 3.10 and 3.11). 2) TJ join permission is induced by the preorder traversal of the fork tree and has a natural decision procedure based on lowest common ancestors (Theorems 3.15 and 3.17). Only selected proofs are presented in this work, due to space limitations.

3.1 Policy Definition

The Transitive Joins policy is defined over a trace language and recognizes a conservative subset of deadlock-free traces.

Definition 3.1. Let symbols $a, b, \ldots$ denote *tasks*. An *action*, α, is one of $\mathit{init}(a)$ (a is the root task), $\mathit{fork}(a, b)$ (a forks b), or $\mathit{join}(a, b)$ (a awaits the termination of b). A *trace*, t, is a sequence of actions; we will use $t_1; t_2$ for trace concatenation.

A trace is valid if it satisfies reasonable constraints on the tasks used in the forks and joins. For example, a fork should always introduce a new task, and a join should only occur between certain pairs of tasks, according to the policy. We keep the formalism general so that other policies (in particular, Known Joins) can fit into the same framework.

Definition 3.2. Let R be some family of relations indexed by traces so that R_t is the relation corresponding to trace t. With respect to R, let $t : A$ denote that t is a valid trace consisting of the tasks A.

$$\frac{}{\mathit{init}(a) : \{a\}}\ \text{valid-init}$$

$$\frac{t : A \qquad a \in A \qquad b \notin A}{t; \mathit{fork}(a, b) : A \cup \{b\}}\ \text{valid-fork}$$

$$\frac{t : A \qquad R_t(a, b)}{t; \mathit{join}(a, b) : A}\ \text{valid-join-}R$$

The valid-* rules state that a trace must begin with an *init* action, that each *fork* action must connect an existing task with a fresh task, and that each *join* action must connect two tasks which are related by R_t, where t is the trace so far.

Definition 3.3. The Transitive Joins judgment $t \vdash a < b$ is given by the following rules. Let $t \vdash a \leq b$ be shorthand for $a = b \lor (t \vdash a < b)$.

$$\frac{t \vdash c \leq a}{t; \mathit{fork}(a, b) \vdash c < b}\ \text{TJ-left}$$

$$\frac{t \vdash a < c}{t; \mathit{fork}(a, b) \vdash b < c}\ \text{TJ-right}$$

$$\frac{t_1 \vdash a < b}{t_1; t_2 \vdash a < b}\ \text{TJ-mono}$$

The $<$ relation should be regarded as the permission-to-join relation for the Transitive Joins policy. (We will show that $<$ is a total order over all the tasks in a trace, justifying the choice of notation.)

Definition 3.4. The *Transitive Joins policy* (TJ) accepts those traces $t : A$ that are derivable when the relation family R is instantiated as $R_t(a, b) \triangleq t \vdash a < b$.

3.2 TJ is Deadlock-free

Lemma 3.5 (Irreflexivity)**.** *For any TJ-valid trace $t : A$, $t \vdash a < a$ cannot be derived for any $a \in A$.*

Lemma 3.6. *If $t : A$ (w.r.t. any relation family R) and $a \in A$, there is a unique action $\alpha := fork(p, a)$ occurring in t, for some task p.*

Definition 3.7. In the statement of Lemma 3.6, let p be called the *parent* of a, and a a *child* of p. Let the (irreflexive) transitive closure of the parent relation be the *ancestor* relation. The reverse of the ancestor relation is the *descendant* relation.

Lemma 3.8 (Transitivity of $<$)**.** *For each TJ-valid task $t : A$ and $a, b, c \in A$, if $t \vdash a < b$ and $t \vdash b < c$, then $t \vdash a < c$.*

Proof. By induction on t.

a) $t := init(a) : \{a\}$: Vacuous by Lemma 3.5.
b) $t := t'; fork(a, b) : A$ and $t' : A'$: By the inductive hypothesis and TJ-mono we have that $t \vdash \cdot < \cdot$ is transitive over A'. It remains to include b in the transitivity. Where $x, y \in A'$, we find that we can only derive
   - $t \vdash x < b$ from TJ-left and $t' \vdash x \le a$
   - $t \vdash b < x$ from TJ-right and $t' \vdash a < x$
   - $t \vdash x < y$ from TJ-mono and $t' \vdash x < y$.

   There are three roles b can play in the transitivity property; the other two roles (call them a_1, a_2) are distinct from b (and thus in A') by Lemma 3.5.
   i) Suppose $t \vdash a_1 < a_2 < b$. Then $t' \vdash a_1 < a_2 \le a$, which yields $t' \vdash a_1 \le a$ by the existing transitivity. With TJ-left we derive $t \vdash a_1 < b$, as desired.
   ii) Suppose $t \vdash b < a_1 < a_2$. Then $t' \vdash a < a_1 < a_2$, which yields $t' \vdash a < a_2$ by the existing transitivity. With TJ-right we derive $t \vdash b < a_2$, as desired.
   iii) Suppose $t \vdash a_1 < b < a_2$. Then $t' \vdash a_1 \le a < a_2$, which yields $t' \vdash a_1 \le a_2$ by the existing transitivity. With TJ-mono we derive $t \vdash a < a_2$, as desired.
c) $t := t'; join(a, b) : A$: Trivial by the inductive hypothesis and TJ-mono.

□

We now show that TJ is *sound* in the sense that it does not admit any deadlocking traces.

Definition 3.9. A trace t contains a *deadlock* if there is a sequence of tasks $a_0, a_1, \ldots, a_n$ ($n \ge 0$) such that t contains $join(a_n, a_0)$ and $join(a_i, a_{i+1})$ for all $i < n$.

Theorem 3.10 (Total order)**.** *For any TJ trace $t : A$, the relation $t \vdash \cdot < \cdot$ defines a (strict) total order over A. That is, $<$ is transitive and trichotomous (for all $a, b \in A$, exactly one of $a < b$, $a = b$, $b < a$ holds).*

Proof. We already have transitivity (Lemma 3.8). It remains to show trichotomy. First, $a = b$ precludes $t \vdash a < b$ and $t \vdash b < a$ by Lemma 3.5. Second, if $a \ne b$, we cannot have both $t \vdash a < b$ and $t \vdash b < a$, since transitivity would then yield $t \vdash a < a$. It remains to show that if $a \ne b$, then at least one of $t \vdash a < b$ or $t \vdash b < a$ holds. We prove the property by induction on t.

a) $t := init(a) : \{a\}$: Vacuous.
b) $t := t'; fork(a, b) : A$: Since only b is new, it suffices to show that for all $a' \in A$ $a' \ne b$ implies $t \vdash a' < b$ or $t \vdash b < a'$. By the inductive hypothesis, we have $t' \vdash a' \le a$ or $t' \vdash a < a'$. In the first case, apply TJ-left; in the second, apply TJ-right.
c) $t := t'; join(a, b) : A$: Trivial by the inductive hypothesis and TJ-mono.

□

Theorem 3.11 (Deadlock-freedom of TJ)**.** *If t is a TJ trace, then t does not contain a deadlock.*

3.3 TJ Order as a Tree Traversal

The preceding formalism has shown that TJ guarantees deadlock-freedom; however, it is not immediately clear how to implement the policy. We now characterize TJ in a way that suggests a natural decision procedure, and we prove its equivalence to the original definitions.

Definition 3.12. Let a trace $t : A$ be given, and let $E := \{(a, b) \mid fork(a, b) \in t\}$ serve as the (directed) edge relation of a tree, T, over the vertices A. Further, let T be equipped with a local child indexing function, $I : A \to \mathbb{Z}$, behaving as follows. For children, b_1, b_2 of a, $fork(a, b_1)$ precedes $fork(a, b_2)$ in t if and only if $I(b_1) < I(b_2)$. T is called the *fork tree* of t.

In the remainder of the section, let T be the fork tree of a TJ-valid trace $t : A$.

Definition 3.13. Define a *preorder tree-traversal* $<_T$ to be any total order over A satisfying the following rules:

1) If $fork(a, b)$ is in t, then $a <_T b$.
2) If $fork(a, c)$ precedes $fork(a, b')$ in t, and b' is an ancestor of b, then $b <_T c$.

Definition 3.14. Let $lca^+(a, b)$ denote the *extended lowest common ancestor* of a and b, defined as 1) $\textsc{anc}^+$ when a is a (proper) ancestor of b, 2) $\textsc{dec}^*$ when a is a descendant of or equal to b, or 3) $\textsc{sib}(a', b')$, where a', b' are the unique nodes such that a', b' are siblings, a' is an ancestor of a, and b' an ancestor of b.

Note that the traditional lowest common ancestor of a and b is either 1) a, 2) b, or 3) the parent of a' and b', corresponding to each of the three cases of lca^+. The extended version has the benefit of telling us how each of a and b are connected to their lowest common ancestor.

Theorem 3.15 (Decision procedure for $<_T$). *Let $a, b \in A$ be given. Proceeding case-wise on $lca^+(a, b)$, we have*

a) $\textsc{anc}^+$ *implies* $a <_T b$.
b) $\textsc{dec}^*$ *implies* $a \not<_T b$.
c) $\textsc{sib}(a', b')$ *implies* $(a <_T b \iff I(a') > I(b'))$.

Corollary 3.16. *There is at most one* $<_T$.

Theorem 3.17 (Preorder). $t \vdash \cdot < \cdot$ *is the unique* $<_T$.

Therefore, Theorem 3.15 suggests an algorithm for TJ verification based on lowest common ancestors in the fork tree. One can dynamically construct T and I as a monotonically growing data structure during the execution of a trace. Upon executing $fork(a, b)$, b is added to T as a new child of a. The index map, I, represents the order in which the children are added.

4 TJ Subsumes KJ

We can recapitulate the definition of the Known Joins policy [10] within the framework of Section 3.1. The common setting then allows us to prove that Transitive Joins subsumes Known Joins (Theorem 4.3), meaning that TJ accepts at least all of the same executions as KJ.

Definition 4.1. The judgment $t \vdash a \prec b$, which denotes that task a knows task b after the execution of trace t, is given by the following rules.

$$\frac{}{t; fork(a,b) \vdash a \prec b}\ \text{KJ-child}$$

$$\frac{t \vdash a \prec c}{t; fork(a,b) \vdash b \prec c}\ \text{KJ-inherit}$$

$$\frac{t \vdash b \prec c}{t; join(a,b) \vdash a \prec c}\ \text{KJ-learn}$$

$$\frac{t_1 \vdash a \prec b}{t_1; t_2 \vdash a \prec b}\ \text{KJ-mono}$$

Definition 4.2. The *Known Joins policy* (KJ) instantiates the relation family R as $R_t(a, b) \triangleq t \vdash a \prec b$, and accepts each trace t for which we can derive some $t : A$.

The $\prec$ relation is the permission-to-join, or "knowledge", relation of Known Joins. The above definition is modified from its original form in Cogumbreiro et al. [10]. The original definition used a map K from tasks to knowledge sets. We have $a \prec b \iff b \in K(a)$. The rules KJ-child and KJ-inherit show, respectively, that a parent task knows its child, and the child inherits all the knowledge of the parent at the time of the fork (cf. T-ASYNC, [10]). The rule KJ-learn shows that, in a join, the waiting task acquires all the knowledge of the terminating task (cf. T-GET, [10]). From KJ-mono, we have that $\prec$ grows monotonically during a trace.

Note the parallels between the $<$ rules in Section 3.1 and the $\prec$ rules here. In particular, TJ-left subsumes KJ-child; TJ-right is essentially KJ-inherit; and we have monotonicity in both cases by the *-mono rules. The differences are that TJ-left obtains much more information than KJ-child (completing transitivity in combination with TJ-right) and that $<$ has no join rule.

We say that TJ *subsumes* KJ by the following result. Note that the superset relation of Corollary 4.4 is strict (c.f. Section 2.3).

Theorem 4.3. *If t is KJ-valid, then* $t \vdash a \prec b \implies t \vdash a < b$.

Corollary 4.4. *If t is KJ-valid, then t is TJ-valid. That is, the TJ policy accepts a superset of the traces of the KJ policy.*

Proof of Theorem 4.3.

Proof. (By induction on the proofs of $t \vdash a \prec b$ and of the KJ-validity of t). If $t \vdash a \prec b$ is derived by KJ-child, KJ-inherit, or KJ-mono, replace the given rule with TJ-left, TJ-right, or TJ-mono, respectively, and recurse on the hypothesis of the rule. The remaining case is that $t \vdash a \prec b$ is derived by KJ-learn. We, therefore, must have a proof of that rule's hypothesis, $t' \vdash c \prec b$, where $t := t'; join(a, c)$. Recurse on $t' \vdash c \prec b$ to obtain $t' \vdash c < b$. Since t is KJ-valid and ends with a join, the KJ validity of t is derived by valid-join-R (for $R := \prec$), which has, as a hypothesis, $t' \vdash a \prec c$. Recurse on $t' \vdash a \prec c$ to obtain $t' \vdash a < c$. Finally, apply Lemma 3.8 (transitivity of $<$) to obtain $t' \vdash a < b$, which yields $t \vdash a < b$ by TJ-mono. □

We have now seen that TJ subsumes KJ and that TJ is a total order over the created tasks. TJ is in this sense a maximally permissive sound policy: TJ permits a superset of the KJ-valid traces, but if even a single additional task pair were added to TJ's total order join permission relation, TJ would then admit some deadlocking traces. Therefore any sound policy which strictly subsumes TJ must necessarily not impose a total order *a priori* from the fork tree; that is, it must respond dynamically to other events besides task forking. That is to say, by observing only forks, TJ improves upon KJ (which observes both forks and joins), but to improve upon TJ one must again observe more than forks.

5 TJ Verifier Specification

We describe an online verifier of TJ validity, including fork and join routines and some possible algorithms for the $<_T$ decision procedure of Theorem 3.15. There is a modular separation between the fork and join routines of the verifier and the underlying implementation of Theorem 3.15.

5.1 Verifier Interface

We assume the ability to create a task record by invoking

$$a = \{\mathsf{node} : u, \mathsf{code} : f\},$$

Algorithm 1 Verifier Interface

```
procedure FORK($a$, $f$)
    if $a$ = null then
        $v \leftarrow$ ADDCHILD(null)
    else
        $v \leftarrow$ ADDCHILD($a$.node)
    $b \leftarrow$ {node : $v$, code : $f$}
    start($b$)
    return $b$

procedure JOIN($a$, $b$)
    if LESS($a$.node, $b$.node) then
        return wait($b$)                    ▷ join is legal
    else
        fault
```

where u is an arbitrary piece of data and f is the function that the task should execute. We assume that execution of a.code begins asynchronously once the method *start*(a) is invoked. We assume that the method *wait*(a) blocks until the termination of a.code and returns the return value of a.code.

We supply two procedures in Algorithm 1:

- FORK(a, f) implements the action of async f by task a; it returns the new child task. If a is null, this represents the initialization of a root task to execute f.
- JOIN(a, b) implements the action of b.*join* by task a; it returns the return value of b.code upon termination of b.code if the join is TJ-valid, and faults (without blocking) otherwise.

Algorithm 1 makes calls to two procedures which maintain a (real or virtual) tree data structure, T:

- ADDCHILD(u) should create and return a new child of vertex u in T, or a new root vertex if u is null.
- LESS(v_1, v_2) should return whether $v_1 <_T v_2$ in the tree T that has been constructed by all the preceding calls to ADDCHILD.

The requirements imposed on ADDCHILD and LESS are

1. Every call to ADDCHILD must return a unique value.
2. LESS and ADDCHILD may be called concurrently (with themselves and with each other).

The guarantees provided to ADDCHILD and LESS by Algorithm 1 are

3. No two instances of ADDCHILD shall be called concurrently on the same parameter (since that parameter uniquely determines the calling task).
4. Each parameter to LESS shall previously have been returned by ADDCHILD.

Algorithm 2 TJ-GT implementation

```
1:  procedure ADDCHILD($u$)
2:      $v \leftarrow$ {parent : $u$, ix : null, depth : 0, children : 0}
3:      if $u$ = null then
4:          return $v$
5:      $v$.depth $\leftarrow u$.depth + 1
6:      $v$.ix $\leftarrow u$.children
7:      $u$.children $\leftarrow u$.children + 1
8:      return $v$

9:  procedure LESS($v_1$, $v_2$)
10:     if $v_1 = v_2$ then
11:         return false
12:     else if $v_1$.depth < $v_2$.depth then
13:         return ¬LESS($v_2$, $v_1$)
14:     $i_1, i_2 \leftarrow$ null          ▷ child indices we arrive by
15:     while $v_2$.depth < $v_1$.depth do
16:         $i_2 \leftarrow v_2$.ix
17:         $v_2 \leftarrow v_2$.parent
18:     while $v_1 \neq v_2$ do
19:         $i_1 \leftarrow v_1$.ix
20:         $i_2 \leftarrow v_2$.ix
21:         $v_1 \leftarrow v_1$.parent
22:         $v_2 \leftarrow v_2$.parent
23:     if $i_1$ = null then                ▷ $i_2$ is never null
24:         return true                     ▷ ANC$^+$ case
25:     return $i_1 > i_2$                  ▷ $v_1 <_T v_2$ iff $i_1 > i_2$
```

5.2 Possible LCA Algorithms

According to Theorem 3.15, the join validation step in an online verifier is essentially a lowest common ancestors computation in the fork tree, T. Recall that we defined an extended LCA function, $lca^+(a, b)$, which provides specific information about how a and b are connected to their LCA. We describe three possible algorithms, TJ-GT, TJ-JP, and TJ-SP. The complexity bounds for each algorithm, and for prior work, are recorded in Table 1.

5.2.1 TJ-GT

The most basic algorithm for maintaining the fork tree, T, and answering lca^+ queries is given in Algorithm 2. This algorithm, TJ-GT, is characterized by having a shared global tree. Each vertex v stores a pointer to its parent, u, an index indicating how many siblings have preceded v, the depth of v in T, and the number of children v has forked so far.

Per its specification, ADDCHILD(u) creates and installs a new child for vertex u. The first step is to allocate a unique vertex, setting u as its parent (line 2). In the special case that we are creating the root node (u is null), we leave v with an initial depth of 0 and no children (lines 3–4). Otherwise, the depth of v is calculated from its parent's depth (line 5).

Table 1. Algorithmic complexities for policy verification; n is the number of tasks, and h is the height of the fork tree. In the worst case, $h = n$.

	KJ-VC	KJ-SS	TJ-GT	TJ-JP	TJ-SP
Fork time	$O(n)$	$O(1)$	$O(1)$	$O(\log h)$	$O(h)$
Join time	$O(n)$	$O(n)$	$O(h)$	$O(\log h)$	$O(h)$
Space	$O(n^2)$	$O(n)$	$O(n)$	$O(n \log h)$	$O(nh)$

The number of preceding children is taken from the parent's child count, which is then incremented (lines 6–7).

LESS(v_1, v_2) decides $v_1 <_T v_2$. The procedure first eliminates the case that v_1 is at least as deep as v_2 using the equivalence $v_1 <_T v_2 \iff v_1 \neq v_2 \land v_2 \not<_T v_1$ (lines 10–13). The procedure then lifts v_2 to an ancestor of the same depth as v_1 by following the parent pointers (lines 15–17). Afterward, both v_1 and v_2 are lifted together until their LCA is found (lines 18–22). Throughout the process, every time a parent pointer is traversed from v to v', the index of v as a child of v' is stored (lines 14–16, 19–20). If the path traversed by v_1 to the LCA was trivial, since the path traversed by v_2 always takes at least one step, the initial v_1 was a proper ancestor of the initial v_2 (lines 23–24). Finally, i_1, i_2 give us the relative order of the two sibling vertices that were traversed immediately prior to reaching the LCA (line 25).

ADDCHILD satisfies requirement 1) of Section 5.1 because it returns a freshly allocated vertex, v. Observe that no concurrent reads can be performed on the fields of v until after ADDCHILD returns, thanks to guarantee 4) of Section 5.1. Moreover, the only data that is modified by ADDCHILD and is at the same time visible to other threads is u.nchildren. However, LESS does not access nchildren, and guarantee 3) ensures that no concurrent ADDCHILD will access the same u.nchildren. Furthermore, LESS is a read-only procedure. Therefore, we justify the correctness of ADDCHILD and LESS in satisfying requirement 2) *without* making use of any synchronization on the data structure.

Each vertex (hence each task) requires a constant amount of storage. The space complexity of TJ-GT is then $O(|T|)$, linear in the number of tasks created. In order to update the data structure upon a fork, $O(1)$ operations are required. No update is required upon joining, but the join verification may require scanning the tree along two paths no longer than the height of the tree. Therefore the time complexity per join is $O(\mathit{height}(T))$.

5.2.2 TJ-JP

Alternative algorithms for lowest common ancestors, and hence lca^+, can be formulated based on various solutions to the level-ancestor problem [18]. For the sake of exploring different time and space complexity trade-offs, we consider another possible algorithm, TJ-JP, based on *jump pointers* [1]. At a cost of extra pointers in the tree, we can dramatically improve the join verification complexity. Let each node maintain not a single parent pointer but an array of pointers to each of its 2^ith ancestors. Moreover, let each of these pointers be paired with the index of the child that the pointer arrives through; these will serve the same purpose as the index field in Algorithm 2. The space per node is no longer constant but $O(\log \mathit{height}(T))$, since a node at depth d will need an array of size $\log_2 d$. Setting up these jump pointers requires $O(\log d)$ time per fork at a node of depth d. However, the jump pointers make traversing the tree much more efficient than in TJ-GT. Instead of scanning linearly across two paths to find their meeting point, one can perform a binary search using the jump pointers. Thus, the join verification time is only $O(\log \mathit{height}(T))$.

Algorithm 3 TJ-SP implementation

```
1:  procedure AddChild($u$)
2:      if $u$ = null then
3:          return {path : [], children : 0}
4:      $p \leftarrow append(copy(u.\text{path}), u.\text{children})$
5:      $u.\text{children} \leftarrow u.\text{children} + 1$
6:      return {path : $p$, children : 0}

7:  procedure Less($v_1$, $v_2$)
8:      $i \leftarrow 0$
9:      while $i < min(length(v_1.\text{path}), length(v_2.\text{path}))$ do
10:         if $v_1.\text{path}[i] \neq v_2.\text{path}[i]$ then
11:             return $v_1.\text{path}[i] > v_2.\text{path}[i]$
12:         $i \leftarrow i + 1$
13:     return $length(v_1.\text{path}) < length(v_2.\text{path})$
```

5.2.3 TJ-SP

Finally, we propose a task-local version of the $<_T$ decision algorithm, TJ-SP (Algorithm 3), in which an explicit shared tree is replaced by a per-task array recording the task's path from the root (its spawn path). Upon each fork, the new task copies its parent's array, appending its own index among its siblings (line 4). An $O(\log \mathit{height}(T))$ scan for the longest common prefix of two tasks' spawn paths yields lca^+ information (lines 9–12). Where the paths diverge, we compare the sibling indices as usual (lines 10–11). If the paths match up to the length of the shorter one, then one of the tasks is an ancestor of the other, so we use the relative lengths of the paths to discriminate the ANC$^+$ and DEC* cases (line 13).

6 Evaluation

We empirically compare the performance of Transitive Joins with that of Known Joins. The two measurements we are interested in are the execution time overhead and the memory usage overhead incurred when using each policy. The three policy implementations we test are Known Joins using vector clocks (KJ-VC), Known Joins using snapshot sets

(KJ-SS), and Transitive Joins using the spawn path algorithm (TJ-SP). The complexity bounds of Table 1 suggest that the jump-pointer algorithm, TJ-JP, may only pay off if the fork tree is very deep. None of our benchmarks exhibit fork trees deeper than 8 tasks, so we did not pursue an evaluation of TJ-JP. We chose to implement TJ-SP over TJ-GT despite the extra memory requirements because using task-local arrays instead of a shared tree of pointers can benefit from cache locality.

All policies were implemented with Armus [9] as a fallback deadlock detector. That is, if the given policy flags a join as invalid, general cycle detection is invoked to determine if the join would truly create a deadlock or if it is just a false positive. Therefore, both the KJ and TJ verifiers are sound *and precise* as implemented. None of our benchmarks can deadlock, but because the fallback cycle detection is slow, the performance of each verifier can be impacted if the policy frequently triggers false positives.

6.1 Benchmark Programs

To make the comparison as fair as possible, we include the same five benchmark programs as in the KJ evaluation [10], and we have implemented our TJ verifier within the same framework as the available KJ implementation, namely, the Habanero Java language [6]. For completeness, we added a benchmark, NQueens, that is invalid under KJ but valid under TJ.

Jacobi A central finite difference stencil is iteratively computed for an 8192×8192 matrix. On each of 30 iterations, a 16×16 array of tasks is forked to compute the stencil in blocks. Each block depends on its own values from the preceding iteration, as well as values at the block boundaries for up to four neighboring blocks. Therefore, each task awaits the completion of five tasks from the previous iteration before proceeding.

Smith-Waterman Two DNA sequences, each of length 21,726, are aligned using the Smith-Waterman dynamic programming algorithm. The score array is divided into 40×40 chunks, with each chunk being computed by a task. Each task must await the completion of three neighboring tasks.

Crypt This Java Grande Forum [31] benchmark has been adapted to the Habanero Java language. The program encrypts and then decrypts 50 MB of data. Each of these two phases consists of embarrassingly parallel work divided among 8192 tasks that are forked and then joined by the root.

Strassen Two $n \times n$ matrices can be multiplied block-wise using seven (not eight) $(n/2) \times (n/2)$ multiplications in a divide-and-conquer strategy. At every level of recursion, the current task spawns the seven recursive multiplications and four subsequent matrix addition tasks. The recursion is cut off at blocks of size 128×128, which are multiplied directly. To multiply the top-level 4096×4096 matrices, the benchmark must spawn 30,811 tasks in a tree of depth 5. A Strassen benchmark also appears in the Cilk and Barcelona OpenMP Task Suite benchmark sets [11, 12].

Table 2. Runtime and memory overheads for verification. Bold-face indicates best factor in each row.

Benchmark	Time (s)/ Mem. (GB)	KJ-VC	KJ-SS	TJ-SP
		Policy Overheads		
Jacobi	12.15	**1.10×**	1.12×	**1.10×**
	3.210	1.33×	1.01×	**1.00×**
Smith-Waterman	5.445	1.03×	1.09×	**0.98×**
	3.726	**1.00×**	**1.00×**	**1.00×**
Crypt	0.4250	9.15×	1.08×	**0.99×**
	0.3286	6.55×	1.02×	**1.00×**
Strassen	7.816	**0.99×**	1.00×	1.00×
	9.005	1.03×	1.22×	**1.02×**
Series	81.10	**1.00×**	1.04×	1.01×
	0.9680	1.95×	2.61×	**1.46×**
NQueens[4]	23.55	1.48×	**1.24×**	1.32×
	6.156	1.53×	1.49×	**1.10×**
Geom. Mean Overhead	Time	1.58×	1.09×	**1.06×**
	Mem.	1.73×	1.30×	**1.09×**

Series Another Java Grande Forum [31] benchmark has been adapted to Habanero Java by Cogumbreiro et al. [10]. One million independent tasks are spawned by the root to calculate the coefficients of the Fourier series for a simple polynomial. The computation completes once the root has joined with all tasks.

NQueens Like Strassen, NQueens employs a divide-and-conquer algorithm to allocate work among tasks. Unlike Strassen, in which each task joins on its own children or siblings, the root task of NQueens joins on all tasks in any order to collect the result. The recursion is 14 levels deep. Almost 3.4 million tasks are spawned in a tree of height 8; the 6 remaining levels of recursion proceed sequentially. An example program with the same high-level structure as NQueens was discussed in Section 2.3. The sequence of joins by the root task of NQueens potentially violates Known Joins, but not Transitive Joins. A divide-and-conquer NQueens also appears in the Cilk and Barcelona OpenMP Task Suite benchmark sets [11, 12].

6.2 Execution Time and Memory Results

We ran each benchmark program under each policy implementation on a 16-core AMD Opteron 3.2 GHz machine. The operating system was Debian 8.10; the Java version was OpenJDK 1.7, running in JDK 1.5 compatibility; the Habanero Java runtime was invoked with 16 hardware threads.

The execution time and memory overheads are presented in Table 2. For the baseline (no policy enabled), we give the absolute time in seconds and memory usage in gigabytes. The execution time is reported using the steady-state methodology [13]; we take the mean of 30 runs, after discarding one warmup run. The memory usage is the average amount of memory in use throughout the 30 post-warmup runs, sampled once every 100 ms. For each of the three verifiers, we report the overhead factors for execution time and memory. Finally, we give the geometric mean overhead for each verifier across all benchmarks. The absolute execution times for the baseline and all three policies are shown with 95% confidence intervals in Figure 2.

The TJ-SP time and memory overheads are no greater than a factor of 1.10× in all but two cases (memory for Series and time for NQueens). The KJ-VC and KJ-SS overheads for all the pre-existing benchmarks (the verifiers were not previously evaluated on NQueens) satisfactorily replicate most of the results of Cogumbreiro et al. [10], with the exception of KJ-VC's memory usage on Jacobi, which we found to be non-competitive with the other verifiers. TJ-SP is comparable in time overhead (within 10 percentage points) to at least one of the KJ verifiers on every benchmark. On memory overhead, TJ-SP again performs comparably well to at least one of the KJ verifiers in all cases except Series and NQueens, for which TJ-SP performs significantly *better* than both KJ verifiers.

On Series, all three verifiers incurred significant memory overheads—at or above 1.5×. TJ-SP is the superior verifier on Series, taking three quarters as much memory as the second best, KJ-VC. Still, Series is an outlier for the memory overhead of TJ-SP, which does not exceed 1.1× on any other benchmark. The excessive memory usage for all three verifiers on Series may be due to the fact that the baseline memory footprint consists of very little data, dominated by the one million spawned tasks, and the space complexity is dependent on the number of tasks.

NQueens is the only potentially KJ-invalid program and was not previously tested with KJ verifiers. We find that the NQueens execution time is significantly impacted by all three verifiers, with KJ-SS performing the best and KJ-VC the worst. Figure 2 shows a large variance in execution time for NQueens over the other benchmarks, suggesting a higher degree of nondeterminism. Recall that NQueens nondeterministically violates KJ and triggers cycle detection; however it never violates TJ. On memory usage for NQueens, both KJ verifiers incur a 1.5× overhead, but TJ-SP performs with only a 1.1× overhead, despite the large number of tasks.

In taking the geometric mean of execution time overheads for each verifier, we find that TJ-SP and KJ-SS have low and comparable impacts on execution time (less than 1.1×). However, the geometric mean of memory overheads reveals that the TJ verifier is the only one of the three verifiers to have a consistently low impact on memory usage.

[4]NQueens was compiled to use an alternative cooperative work-sharing runtime, in contrast to the blocking work-sharing runtime used in all the other benchmarks. This is due to KJ-SS anomalously timing out after two hours under the blocking runtime.

7 Related Work

7.1 General Approaches

Broadly, there are three categories of solutions to the deadlock problem, as outlined by Coffman et al. [8]: 1) static prevention, 2) detection at runtime, and 3) avoidance at runtime. In the first category, approaches seek to statically identify potential deadlocks or prove their absence using, for example, static analyses or type systems [4, 27, 34]. Runtime detection consists of identifying cycles of deadlocked tasks after the deadlock has already occurred [19, 20, 24–26, 32]. Finally, runtime deadlock avoidance strategies, to which this work belongs, intercept attempted join operations that will or may cause a deadlock [3, 5, 9, 10, 14, 28, 33]. The advantage of avoidance over detection is that a target program has the opportunity to recover from aborted joins; however, avoidance can be more challenging and expensive than detection [9].

A general algorithm to avoid deadlocks on task termination and barrier synchronization is to perform cycle detection in the *waits-for graph* to determine if each attempted join would create a deadlock cycle [29]. The advantage of cycle detection is that it is sound and precise; that is, it exactly determines whether a deadlock would arise. However, the time to verify each join with this method is quadratic in the number of tasks [29], resulting in prohibitive runtime overheads in practice [10].

7.2 Relationship to Known Joins

Recent work on deadlock avoidance has proposed Known Joins (KJ), a conservative policy that precludes deadlocks using a set of rules that are efficient to check at runtime [10]. KJ's advantage is that online policy verification is a low-overhead operation in practice. However, the concern with a conservative policy is that it may have limited utility by rejecting many reasonable deadlock-free programs. The implementation of a Known Joins verifier therefore uses Armus [9], a cycle-detection algorithm for deadlock avoidance, as a fallback mechanism. When a program violates KJ, Armus is invoked to precisely filter out false positives. A previous evaluation has demonstrated prohibitively expensive runtime overheads in using Armus alone, but the hybrid implementation of KJ with Armus achieves both the low overhead of a conservative policy and the precision of cycle-detection [10]. However, prior work has not established how KJ performs when a deadlock-free target program frequently triggers the fallback mechanism.

Known Joins was originally designed for the purpose of proving deadlock-freedom from data-race-freedom in the Futures model. For this reason, the KJ policy is not necessarily

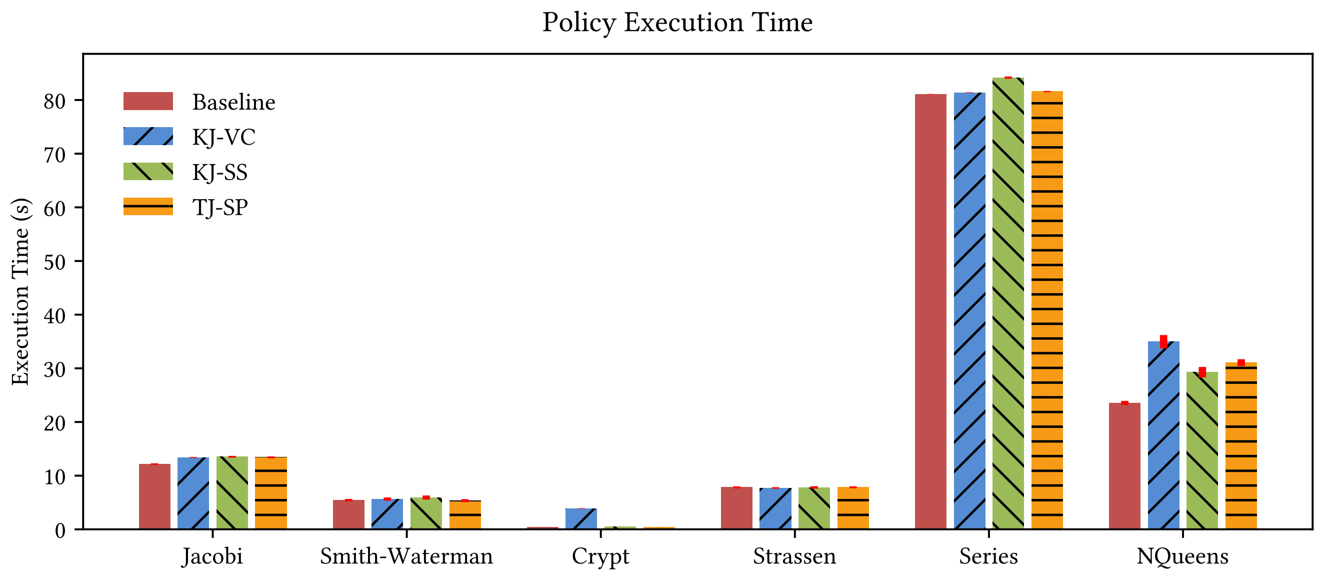

Figure 2. Execution times for each evaluated policy verifier, showing the mean with a 95% confidence interval.

suited as a widely applicable deadlock-avoidance strategy. In contrast, we set out to create a new policy specifically designed to admit a large class of deadlock free programs in the Futures model. To understand this point, we ask what interesting programs are deadlock-free but rejected by KJ?

Consider the tree formed by the forks in a program, where each newly forked task is a child, and where the forking task is the parent. KJ permits a task to join on an immediate child. But if a task joins on a descendant other than one of its immediate children, KJ requires the task to have *first* joined with all the intervening nodes on the path to the descendant. The class of behaviors we have chosen to admit in TJ is the *arbitrary descendant join*: A task should be permitted to join on any descendant regardless of what other joins have already occurred, because weakening KJ to admit arbitrary descendant joins does not compromise deadlock-freedom. This behavior arises in a natural implementation of the 'finish' construct of X10 and Habanero Java. When a finish block ends, it joins with the collection of tasks spawned transitively within the block [7]. A finish implementation may trigger a false positive under KJ unless the join order carefully respects the fork order, limiting the usefulness of KJ. We discuss this example at length in Section 2.3. In order to admit arbitrary descendant joins, we define a new Transitive Joins (TJ) policy for deadlock-freedom, which exhibits transitive join permissions.

Transitivity of join permissions also has a practical advantage for the implementation of an online policy verifier. TJ retains the KJ concept of permission inheritance upon forking. But strikingly, the KJ concept that a joining (waiting) task should learn new permissions from the joinee (terminating) task becomes obsolete, for the joining task will already have these permissions by transitivity. As a consequence, a join operation does not require a TJ verifier to update any permission state, thereby simplifying a possible implementation in comparison to a KJ verifier.

8 Conclusion

We have presented a novel deadlock-avoidance policy, Transitive Joins, for dynamic task parallelism with arbitrary join operations, which is applicable to a range of parallel programming models such as Futures and threads. The policy is a set of rules for task joins that admits only deadlock-free runs. TJ admits a strictly larger class of practical deadlock-free programs than the prior Known Joins policy.

TJ can be implemented as an online verifier using one of several possible algorithms, with cycle-detection as a fallback to catch false positives. The implementation of our TJ verifier for the Habanero Java language competes with and in some cases surpasses existing implementations of two KJ verifiers in minimizing both time and memory overhead. We demonstrated that TJ can efficiently verify benchmarks with a large number of tasks, incurring geometric mean time and memory overheads of 1.06× and 1.09×, respectively. This result improves over the state of the art, KJ-SS, whose time and memory overheads are 1.09× and 1.30×, respectively. Therefore, we propose that our TJ verifier is a practical choice for an always-on deadlock safety checker.

Acknowledgments

This work is supported by the National Science Foundation under Collaborative Grant No. 1629564 and Graduate Research Fellowship Grant No. 1650044.

References

[1] Michael A. Bender and Martín Farach-Colton. 2004. The Level Ancestor Problem Simplified. *Theor. Comput. Sci.* 321, 1 (2004), 5–12.

[2] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 720–748.

[3] Gérard Boudol. 2009. A Deadlock-Free Semantics for Shared Memory Concurrency. In *Proc. 6th Int'l. Coll. on Theoretical Aspects of Computing (ICTAC '09)*. Springer-Verlag, Berlin, Germany, 140–154.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proc. 17th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, 211–230.

[5] Jeremy D. Buhler, Kunal Agrawal, Peng Li, and Roger D. Chamberlain. 2012. Efficient Deadlock Avoidance for Streaming Computation with Filtering. In *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, 235–246.

[6] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proc. 9th Int'l. Conf. on Principles and Practice of Programming in Java (PPPJ '11)*. ACM, New York, NY, 51–61.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proc. 20th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, 519–538.

[8] E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System Deadlocks. *ACM Comput. Surv.* 3, 2 (1971), 67–78.

[9] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2015. Dynamic Deadlock Verification for General Barrier Synchronisation. In *Proc. 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '15)*. ACM, New York, NY, 150–160.

[10] Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco T. Vasconcelos, and Max Grossman. 2017. Deadlock Avoidance in Parallel Programs with Futures: Why Parallel Tasks Should Not Wait for Strangers. *Proc. ACM Program. Lang.* OOPSLA, Article 103 (2017), 26 pages.

[11] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proc. 2009 Int'l. Conf. on Parallel Processing (ICPP '09)*. IEEE Computer Society, Washington, DC, 124–131.

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. ACM SIGPLAN 1998 Conf. on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, 212–223.

[13] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proc. 22nd ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '07)*. ACM, New York, NY, 57–76.

[14] Prodromos Gerakios, Nikolaos Papaspyrou, Konstantinos Sagonas, and Panagiotis Vekris. 2011. Dynamic Deadlock Avoidance in Systems Code Using Statically Inferred Effects. In *Proc. 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. ACM, New York, NY, Article 5, 5 pages.

[15] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. 2006. *Java Concurrency in Practice*. Pearson Education, London, England.

[16] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism. In *Proc. 2009 IEEE Int'l. Symp. on Parallel & Distributed Processing (IPDPS '09)*. IEEE Computer Society, Washington, DC, 1–12.

[17] Robert H. Halstead, Jr. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (1985), 501–538.

[18] Dov Harel and Robert Endre Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.

[19] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. 2009. A Graph Based Approach for MPI Deadlock Detection. In *Proc. 23rd Int'l. Conf. on Supercomputing (ICS '09)*. ACM, New York, NY, 296–305.

[20] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *Proc. Int'l. Conf. on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society, Los Alamitos, CA, Article 30, 11 pages.

[21] Shams Imam and Vivek Sarkar. 2014. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *Proc. 2014 Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, 75–86.

[22] S. Sreekaanth Isloor and T. Anthony Marsland. 1980. The Deadlock Problem: An Overview. *Computer* 13, 9 (1980), 58–78.

[23] ISO. 2011. *ISO/IEC 14882:2011: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland.

[24] Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. Müller. 2008. MPI Correctness Checking with Marmot. In *Tools for High Performance Computing*. Springer, Berlin, Germany, 61–78.

[25] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. 2004. MPI Application Development Using the Analysis Tool MARMOT. In *Proc. Int'l. Conf. on Computational Science (ICCS '04)*. Springer-Verlag, Berlin, Germany, 464–471.

[26] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. 2003. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience* 15, 2 (2003), 93–100.

[27] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective Static Deadlock Detection. In *Proc. 31st Int'l. Conf. on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, 386–396.

[28] Armand Navabi, Xiangyu Zhang, and Suresh Jagannathan. 2008. Quasi-static Scheduling for Safe Futures. In *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, 23–32.

[29] Spiridon A. Reveliotis, Mark A. Lawley, and Placid M. Ferreira. 1997. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *IEEE Trans. Automat. Control* 42, 10 (1997), 1344–1357.

[30] Jun Shirako, Vincent Cavé, Jisheng Zhao, and Vivek Sarkar. 2013. Finish Accumulators: An Efficient Reduction Construct for Dynamic Task Parallelism. In *Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Germany, 264–265.

[31] L. A. Smith, J. M. Bull, and J. Obdrzálek. 2001. A Parallel Java Grande Benchmark Suite. In *Proc. 2001 ACM/IEEE Conf. on Supercomputing (SC '01)*. ACM, New York, NY, Article 8, 10 pages.

[32] Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2011. Large Scale Verification of MPI Programs Using Lamport Clocks with Lazy Update. In *Proc. 2011 Int't. Conf. on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, Washington, DC, 330–339.

[33] Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe Futures for Java. In *Proc. 20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, 439–453.

[34] Amy Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *Proc. 19th European Conf. on*

Object-Oriented Programming (ECOOP ’05). Springer-Verlag, Berlin, Germany, 602–629.

A Artifact Appendix

A.1 Abstract

The Transitive Joins deadlock avoidance policy is provided as a plugin for the Habanero-Java compiler and runtime. The software archive contains source code for the compiler, TJ-SP, KJ-VC, KJ-SS, and the benchmark suite. Automated scripts set up the testing environment and run the benchmarks. A multicore machine with a Docker installation is required. The average execution time and memory usage overheads should be compared with the results in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** deadlock avoidance
- **Program:** Habanero-Java benchmarks included
- **Compilation:** Habanero-Java compiler included
- **Run-time environment:** Docker
- **Hardware:** recommend 16-core CPU
- **Metrics:** execution time, memory usage
- **Output:** metrics; execution time and memory usage plots
- **Experiment workflow:** extract archive; build Docker image; run benchmark and result processing scripts

A.3 Description

A.3.1 How delivered

The source files and scripts were made available to the evaluators as a Docker image that includes all software dependencies.

A.3.2 Hardware dependencies

Our evaluation was conducted on a 16-core CPU 3.2 GHz AMD Opteron. The benchmark script launches the JVM with 60 GB of RAM. Any comparable machine will suffice. We reported execution time overheads factors relative to a baseline; however even these figures will vary from machine to machine due to nondeterminism.

A.3.3 Software dependencies

The Docker image includes Java 8 and Python 3 installations, the source tree for Habanero-Java, the source code for the policy plugins, and the six benchmark programs.

A.4 Installation

The Docker image is built by a Makefile, which includes the automated setup of the Habanero-Java compiler, runtime, and policy plugins.

A.5 Experiment workflow

Each benchmark is a Habanero-Java program using Futures. Within the Docker container, a script runs the registered benchmarks in turn. For a given benchmark, the program is first compiled and run under Habanero-Java with no modifications so that join operations are unchecked (the baseline). Then for each policy (KJ-VC, KJ-SS, TJ-SP) the program is re-compiled so that the join operations are checked by that policy first, and, upon violation, by Armus cycle detection. Each benchmark program is configured to run 31 iterations within the same VM, printing the execution time for each iteration and average memory usage across 100-ms samples.

A second script parses the log file from all the benchmarks, computes the mean of 30 execution times (dropping the first sample), the overhead factors for time and for memory usage of each policy over the baseline, and the geometric mean of overheads across all benchmarks. This script generates data for Table 2 and a plot like Figure 2.

A.6 Evaluation and expected result

The TJ-SP algorithm presented in this paper is implemented as a Habanero-Java extension, alongside existing implementations of KJ-VC and KJ-SS. The code which checks each join operation for validity is After installation, a single script runs all or a selected subset of the benchmarks, generating a log file with raw data. A second script parses the data to produce aggregate results and plots.

The full suite takes around 5 hours to run on the suggested hardware. Independent of absolute execution time and memory usage, the computed time and memory overhead factors should reproduce results comparable to Table 2. Factors such as machine architecture, scheduling, and nondeterminism will contribute to differences in the results.

A.7 Experiment customization

Any Habanero-Java program can be added as a benchmark. The ‘Harness’ class records and prints timing and memory statistics in the same way as the existing benchmarks. The benchmark script can be adapted to include new benchmarks, or the Habanero-Java compiler and runtime can be invoked directly on the the program.