Proving Data Race Freedom in Task Parallel Programs Using a Weaker Partial Order

Benjamin Ogles[1], Peter Aldous[1], and Eric Mercer[1]

[1]Brigham Young University, Provo, UT, 84601, USA

Abstract—**Task parallel programming models such as Habanero Java help developers write idiomatic parallel programs and avoid common errors. Data race freedom is a desirable property for task parallel programs but is difficult to prove because every possible execution of the program must be considered. A partial order over events of an observed program execution induces an equivalence class of executions that the program may also produce. The Does-not-Commute (DC) relation is an efficiently computable partial order used for data race detection. As a relatively weak partial order, the DC relation can represent relatively large equivalence classes of program executions. However, some of these executions may be infeasible, thus leading to false data race reports. The contribution of this paper is a mechanized proof that the DC relation is actually sound for commonly used task parallel programming models. Sound means that the first data race identified by the DC relation is guaranteed to be a real data race. A prototype analysis in the Java Pathfinder model checker shows that the DC relation can significantly reduce the number of explored states required to prove data race freedom in Habanero Java programs. In this application, the search for data race using the DC relation is both sound and complete.**

I. Introduction

A parallel program execution is said to *witness* a data race when two distinct threads access the same memory location consecutively and one of the accesses is a write. A parallel program is said to contain a data race when it can produce an execution that witnesses a data race.

Dynamic analysis attempts to detect data races by observing a program execution and computing a partial order over the observed events, as in Serebryany et al. [27]. A pair of events that are unrelated by the partial order are considered to be logically concurrent, i.e. they can occur adjacent to one another in some execution of the program. Therefore, even if the observed program execution does not *witness* a data race, the partial order may still *detect* a data race if two conflicting events are logically concurrent.

Weaker partial orders can identify more pairs of concurrent events and therefore have a higher chance of detecting a data race if one exists. The challenge is to define a partial order that is as weak as possible while remaining *sound*. Sound partial orders guarantee that the first pair of unrelated conflicting events in the observed execution actually represents a feasible data race [20]. A data race is *feasible* if a program execution exists that witnesses the reported data race.

The most commonly used sound partial order is Lamport's Happens-Before (HB) relation [18]. Events that access the same lock variable are totally ordered by the HB relation. The Weak-Causally-Precedes (WCP) relation weakens the HB relation by relaxing the ordering on locking events [17]. The WCP relation has also been proven sound.

The Does-not-Commute (DC) relation weakens the WCP relation to detect data races that the HB and WCP relations miss [26]. However, the DC relation is unsound; it can report data races that are infeasible. To compensate, the Vindicator analysis in [26] attempts to construct the witness execution for reported data races at runtime. If it cannot construct a witness execution for a data race, then it is discarded as a false positive. This ensures that the analysis is sound as a whole, but can also create a significant runtime overhead.

This work makes the observation that although the DC relation is unsound in general, there may be classes of program executions that it can analyze in a sound manner. The main contribution of this paper is a mechanized proof in Coq that the first data race detected by the DC relation is always a feasible data race if the locking events in the observed program execution all share the same lock variable. Furthermore, this class of program executions includes the executions that are produced by certain task parallel programming models, making the result practically useful. The proof script exports theorems that are parametric to a relation over events in a task parallel program execution, allowing them to be reused in verifying other partial orders.

Task parallel programming models such as Habanero Java [3] aim to reduce the complexity of parallel programs by restricting how threads can interact with each other. These languages remain useful despite their restrictions because they include high level constructs such as parallel loops, futures and mutual exclusion that help developers write idiomatic parallel programs and avoid common errors. They can also help gain portable performance across different computer architectures because parallel computations are expressed as tasks, a hardware agnostic abstraction. More importantly for this work, the DC relation is sound on task parallel semantics.

There are multiple definitions for the semantics of Habanero Java's **isolated** construct; for example, the official webpage for Habanero Java [1] states that all **isolated** blocks of code execute in mutual exclusion, while Cavé et al. [3] use the weaker property that mutual exclusion is only necessary for **isolated** blocks containing conflicting accesses. The former semantics can be implemented as mutual exclusion with a single global lock. This approach is performant when critical sections execute infrequently [3] and enables a guarantee of deadlock freedom. With only one lock, the DC relation is sound. This work also argues informally that the mechanized proof can be extended to program executions produced by other **isolated** semantics, such as atomic variables.

Although the DC analysis is sound, it is not complete. It cannot always detect data races that are knowable from an observed program execution. Furthermore, the program under analysis may also exhibit different behaviors depending on the observed thread schedule. A model checker can overcome this problem by enumerating all of the relevant thread schedules for the program. Therefore, the DC analysis is sound and incomplete with respect to one observed execution but the model checking analysis is sound and complete with respect to the whole program and a given input.

This work also presents a prototype implementation of the DC data race detection analysis in the Java Pathfinder model checker [31], which already has implementations of HB and WCP. The partial order used for data race detection is also used as the dependency relation in a dynamic partial order reduction algorithm [9]. This order directly influences the number of thread schedules the model checker must explore to prove the absence of data race in Habanero Java programs for a given input. Experimental results show that using the DC relation can significantly reduce the number of thread schedules explored by the model checker. Because the DC relation is sound for the Habanero Java programming model, detected data races are guaranteed to be real errors and can be immediately reported to the user, avoiding the overhead of the Vindicator analysis.

In summary, the contributions of this paper are as follows.

- Utilities for verifying partial orders over task parallel program executions in Coq.
- A mechanized proof that the DC relation is sound for task parallel programs that use a single global lock.
- An implementation of the dynamic partial order reduction algorithm in the Java Pathfinder model checker and an empirical comparison of the HB, WCP and DC relations.

II. Preliminaries

The set M denotes a finite set of memory locations. A concurrent program is a finite set of threads T that are each able to to *read* (rd), *write* (wrt), *acquire* (acq), or *release* (rel) memory locations throughout their execution. These are generally referred to as thread *actions*: $A = \{\mathrm{rd}, \mathrm{wrt}, \mathrm{acq}, \mathrm{rel}\}$. Reading and writing are considered *access* actions. The rest are *locking* actions.

The meaning of a concurrent program execution is given by its observed *history* of *events*. An event is a tuple, $e \in T \times A \times M$, identifying a thread, action, and memory location. A history, $\sigma = e_0 \ldots$ is a sequence of zero or more events.

A history defines a total order $\prec^{\sigma}$ over its events. The projection of a history to a thread, $\sigma|_t$, is a sub-history that preserves the ordering in $\prec^{\sigma}$ and includes all events found in σ that belong to t (and no others). Two events are *thread ordered* in a history, $e_i \prec^{\sigma}_{TO} e_j$, if and only if they occur in order on the same thread: $\exists t \in T\ (e_i \in \sigma|_t \wedge e_j \in \sigma|_t \wedge e_i \prec^{\sigma} e_j)$. Set membership on a sequence, although a slight abuse of notation, is used for simplicity in the presentation.

An acquire event $e_i = \langle t, \mathrm{acq}, m\rangle$ in a history has a *match* if and only if a release event follows in the same thread and if no threads acquire or release m in events occurring between the acquire and the release: $\exists e_j \in \sigma\ (e_j = \langle t, \mathrm{rel}, m\rangle \wedge e_i \prec^{\sigma} e_j \wedge \forall e_k \in \sigma\ (e_i \prec^{\sigma} e_k \prec^{\sigma} e_j \implies \forall t' \in T\ (e_k \neq \langle t', \mathrm{acq}, m\rangle \wedge e_k \neq \langle t', \mathrm{rel}, m\rangle)))$. The match for a release event is defined conversely. In this definition, acquire and release must be matched on the same thread, and no intervening locking actions can exist on the same memory location by any thread.

Definition 1 (well-formed history)**.** *A history is well-formed if and only if all release events are matched, all acquire and release events apply to the same location, and that memory location is not found in any access events in the history.*

A well-formed history dedicates a single memory location as a global lock. The DC relation is proven sound only on well-formed histories. In general, any non-deadlocking execution of a concurrent program that uses a single global lock generates a well-formed history.

Two access events $e_i = \langle t, a, m\rangle$ and $e_j = \langle t', a', m\rangle$ on the same memory location are said to conflict, $e_i \asymp e_j$, if they originate from different threads and either is a write event: $t \neq t' \wedge (a = \mathrm{wrt} \vee a' = \mathrm{wrt})$. A history σ is said to *witness* a data-race if two events are adjacent in σ and conflict: $\sigma = e_0 \ldots e_i e_j e_k \ldots \wedge e_i \asymp e_j$. Being adjacent in the history guarantees that there are no intervening lock actions.

It is not easy to determine if there is a data race in a history if it is not directly witnessed. Approaches to detect data race based on lock sets track the locking actions in each thread and report a race on conflicting accesses not protected by a common lock [6]. Partial order approaches predict from the observed history other histories that witness the data race if any exist.

The HB relation is sound in regards to the histories it predicts meaning that all the predictions are valid histories that can be generated by the program. As such, a data race witnessed by a predicted history is real when using the HB relation. The goal of this work is to show that the DC relation maintains this important property in limited but useful contexts while predicting even more histories from the observed one.

The set of events in the *critical section* in a history for an acquire event, $e = \langle t, \mathrm{acq}, m\rangle$, is $CS(e) = \{e_i \mid e_i \in \sigma|_t \wedge e \prec^{\sigma} e_i \prec^{\sigma} match(e)\}$. The critical section for a release event is defined similarly.

Definition 2 (DC Relation)**.** *The does-not-commute relation for a history, $\prec^{\sigma}_{DC}$, is the smallest partial order over events in σ that includes $\prec^{\sigma}_{TO}$ and satisfies the following: if there exists a release event e, an acquire event e' and two access events e_i and e_j such that*

$$e_i \in CS(e) \wedge e_j \in CS(e') \wedge e \prec^{\sigma} e_j \wedge e_i \asymp e_j$$

then $e \prec^{\sigma}_{DC} e_j$.

The relation only includes, beyond the per-thread ordering of events, the orders between events in critical sections and later conflicting events also in critical sections. This definition is adapted from [26] and simplified for task parallel programs. The simplification falls out of the history being well-formed so there is never more than a single global lock. With a single lock, it is not necessary to add an order from a release event to an acquire event when their separate critical sections have DC ordered events.

The DC relation *predicts* a data race from an execution if and only if $\exists e_i, e_j \in \sigma\ (e_i \asymp e_j \wedge e_i \prec^\sigma e_j \wedge e_i \not\prec^\sigma_{DC} e_j)$. Any conflicting accesses that are unordered by DC are data races. The proof shows that the predicted data race that happens the earliest in the history is feasible. Later races may or may not be feasible, since the first data race could influence events downstream.

Definition 3 (first data race)**.** *The DC predicted data race on events e_x and e_y (with e_x occurring first: $e_x \prec^\sigma e_y$) is the first predicted data race for the history if and only if e_x and e_y race and no other races occur before e_y and if e_x is the* last *event before e_y that races with e_y: $e_x \not\prec^\sigma_{DC} e_y \wedge e_x \asymp e_y \wedge \forall e_i, e_j \in \sigma\ (e_i \prec^\sigma e_j \prec^\sigma e_y \wedge e_i \asymp e_j \implies e_i \prec^\sigma_{DC} e_j) \wedge \forall e_z \in \sigma\ (e_z \asymp e_y \wedge e_z \prec^\sigma e_y \implies e_z \prec^\sigma e_x)$*

The proof in the next section shows that for any well-formed history with some earliest DC predicted data race, a sub-history exists that witnesses the data race and can be constructed from the events of the observed history. This sub-history is a valid program execution that can be inferred from the observed execution. Such a sub-history is formalized as a *valid reordering*.

A read event, $e_j = \langle t', \mathrm{rd}, m\rangle$, *observes* a write event $e_i = \langle t, \mathrm{wrt}, m\rangle$, given as $e_i = \mathit{observes}(e_j, \sigma)$, in a history if and only if $e_i \prec^\sigma e_j \wedge \forall e_k \in \sigma\ (e_i \prec^\sigma e_k \prec^\sigma e_j \implies \forall t_k \in T\ (e_k \neq \langle t_k, \mathrm{wrt}, m\rangle))$. The observed write is the value read. This definition naturally assumes a sequentially consistent memory model. Weaker memory models are not considered in this work.

Definition 4 (valid reordering)**.** *A sub-history σ' is a valid reordering of σ if and only if σ' is well-formed, $\forall t \in T$ ($\sigma'|_t$ is prefix of $\sigma|_t$), and $\forall e \in \sigma', \forall t \in T, \forall m \in M\ (e = \langle t, \mathrm{rd}, m\rangle \implies \mathit{observes}(e, \sigma') = \mathit{observes}(e, \sigma))$.*

The definition preserves locking semantics because the sub-history must be well-formed. It also maintains the sequential execution in each thread and the observed write for each read. Any program that produces the history σ, can also produce valid reorderings of σ.

A valid reordering of a history σ consists of a subset of the events in σ. In order for the valid reordering to witness the first DC predicted race in σ this subset of events is strategically selected. The strategy employed by the proof is based on *causal events* [24], [4]. If e_x and e_y are the first DC predicted data race in the original history, then the set of causal events for the data race is $CE(\sigma, e_x, e_y) = \{e \mid e \in \sigma \setminus \{e_x, e_y\} \wedge (e \prec^\sigma_{DC} e_x \vee e \prec^\sigma_{DC} e_y)\}$. Any causal event must be DC ordered before one of the events in the data race.

In order for causal events to be arranged into a valid reordering, certain events must come before others. For example, read events must come after their observed write events and release events must come after their matching acquire events. These requirements are encoded as a relation over causal events, called the *Witness-Order*.

Definition 5 (witness-order)**.** *Witness-Order, denoted by $\prec$, is the smallest reflexive and transitive relation over $CE(\sigma, e_x, e_y)$ that satisfies these rules*

- *if $e_i \prec^\sigma_{DC} e_j$, then $e_i \prec e_j$*
- *if e_i is a release event and e_j is an acquire event with $e_i \prec^\sigma e_j$ then $e_i \prec e_j$*
- *if e_j is an acquire event and e_i is a release event with $e_j \prec^\sigma e_i$ and $\mathit{match}(e_j) \notin CE(\sigma, e_x, e_y)$, then $e_i \prec e_j$*

The first two rules intuitively enforce that read events observe the same write events as in the original history and that critical sections do not overlap. The last rule handles the case where one of the events that races occurs in a critical section. In this case, the acquire event is a causal event but the matching release event is not. For the causal events to be arranged into a well-formed history, this acquire event must be the last event in the history that accesses the global lock. Therefore, the third rule of Witness-Order ensures that this acquire event is ordered after every other release event in the reordering.

III. Soundness of DC

This section gives a sketch of the proof that the first DC predicted race in a well-formed task parallel program history is always a feasible data race. The Coq source code is available for download at *https://bitbucket.org/byu-vv/traces-coq/src/master*.

Let the events $e_x = \langle t_x, a_x, m\rangle$ and $e_y = \langle t_y, a_y, m\rangle$ form the first DC predicted race in the history σ. Also, without loss of generality, let e_y be the last event in σ, as it can always be sliced to this point during an analysis. By the definition of feasible data race, it is sufficient to prove that there exists a valid reordering of σ that witnesses the data race of e_x and e_y. This is done constructively by selecting a subset of events from σ, reordering them into a new sub-history and appending e_x and e_y to this reordering. It is then shown that the sub-history with e_x and e_y last is a valid reordering of σ. Because e_x and e_y are the last events in the sub-history, it also witnesses the data race, and it is concluded that e_x and e_y must form a real race. A few supporting lemmas simplify the proof.

Lemma 1. *If for two events $e = \langle t, a, n\rangle$ and $e' = \langle t', a', n'\rangle$ in σ, $e \prec^\sigma_{DC} e' \wedge t \neq t'$, then $\exists e_r = \langle t_r, \mathrm{rel}, l\rangle$ such that $e \prec^\sigma_{TO} e_r \wedge e_r \prec^\sigma_{DC} e'$*

This follows directly from the definition of the DC relation.

Lemma 2. *If for an acquire event $e \in CE(\sigma, e_x, e_y)$, $\mathit{match}(e) \notin CE(\sigma, e_x, e_y)$, then $e \prec^\sigma_{TO} e_x$ or $e \prec^\sigma_{TO} e_y$*

Let $e = \langle t, \text{acq}, l \rangle$. Because e is a causal event it must be DC ordered before e_x or e_y. Assume that $t \neq t_x \wedge t \neq t_y$. Then by Lemma 1, there must exist a release event thread ordered after e that is DC ordered before e_x or e_y. But this is a contradiction because σ is well-formed and $match(e)$ is not a causal event, so e must be the last causal event on its thread to access the global lock.

Lemma 3. *There is at most one acquire event e such that $e \in CE(\sigma, e_x, e_y)$ and $match(e) \notin CE(\sigma, e_x, e_y)$*

As mentioned in Sec. II, this case only appears when one of e_x and e_y occur in a critical section. If there were more than one causal acquire event without a matching causal release event, then either both e_x and e_y occur in critical sections or one of their threads performed two acquire events without performing a release event. The first case is a contradiction because the DC relation always orders conflicting events in critical sections, so e_x and e_y would not be a data race. The second case is a contradiction because σ is a well-formed history. As defined in this work, well-formed histories do not permit reentrant locks.

Lemma 4. *$\prec$ is antisymmetric.*

By Lemma 3 there are two cases to consider. In the first case, for every acquire event $e \in CE(\sigma, e_x, e_y)$, $match(e)$ is also a causal event: $match(e) \in CE(\sigma, e_x, e_y)$. Therefore the last rule of $\prec$ will never apply and $\prec \subseteq \prec^{\sigma}$. Because $\prec^{\sigma}$ is antisymmetric, $\prec$ is also antisymmetric in this case.

In the second case, there is one causal acquire event $e = \langle t, \text{acq}, l \rangle$ such that $match(e)$ is not a causal event. If there is no release event e' such that $e \prec^{\sigma} e'$ then the third rule of $\prec$ will still never apply. Therefore $\prec \subseteq \prec^{\sigma}$ still holds and $\prec$ is antisymmetric.

Otherwise, some $e' = \langle t', \text{rel}, l \rangle$ exists such that $e \prec^{\sigma} e'$ and $e' \prec e$. If a cycle exists in $\prec$, then it must be through such a path for some e', because all other edges in $\prec$ align with the total order of the original history. However, $\prec$ only orders e before causal events on its same thread because there are no release events thread ordered after e that are also causal events. Therefore, if a cycle exists through the path $e \prec e'$, then $t = t'$ which contradicts the previous statement.

Let σ' be any linearization of $\prec$.

Lemma 5. *For any thread t, $\sigma'|_t$ is a prefix of $\sigma|_t$.*

Because $\prec^{\sigma}_{TO} \subseteq \prec^{\sigma}_{DC}$, for any causal event $e \in CE(\sigma, e_x, e_y)$, any event $e' \in \sigma$ such that $e' \prec^{\sigma}_{TO} e$ is also a causal event. In addition, the first rule of $\prec$ ensures that all events on the same thread will be ordered in σ' as they were in σ.

Lemma 6. *σ' is well-formed.*

Because of Lemma 3, there are two cases to consider. In the first case, for every causal acquire event, its matching release is also a causal event. Then the first and second rule of $\prec$ ensure that all locking events are ordered in σ' as they were in σ. Because σ is consistent with lock semantics, so is σ'.

In the second case, there is one causal acquire event such that its matching release is not a causal event. Then the rules of $\prec$ ensure that this acquire event is the last locking event in σ' and the rest of the locking events are ordered as in σ.

Lemma 7. *For every read event $e \in CE(\sigma, e_x, e_y)$, $observes(e, \sigma) = observes(e, \sigma')$*

Because e_x and e_y are the first race in σ and $observes(e, \sigma)$ is either an event on the same thread as e or $observes(e, \sigma) \asymp e$, it must be the case that $observes(e, \sigma) \prec^{\sigma}_{DC} e$. By the same logic, for any two causal events $e_i, e_j \in CE(\sigma, e_x, e_y)$, if $e_i \asymp e_j$, then $e_i \prec^{\sigma}_{DC} e_j \vee e_j \prec^{\sigma}_{DC} e_i$. Therefore the write events on the variable accessed by $observes(e, \sigma)$ and e are totally ordered in σ'

Theorem 1. *$\prec$ is a partial order over $CE(\sigma, e_x, e_y)$ and the history $\sigma' = e_0 \ldots e_x e_y$ is a valid reordering of σ.*

The definition of causal events guarantees that e_x and e_y are enabled at the end of σ'. The rest follows from Lemma 4, Lemma 5, Lemma 6 and Lemma 7.

A. Other Implementations of Mutual Exclusion

The **isolated** keyword in Habanero Java is used to mark regions of code that may need to be executed in a mutually exclusive manner. The semantics of **isolated** ensures that code in isolated regions will be executed mutually exclusive from other interfering isolated regions, where interfering means that the isolated regions contain conflicting access events. The simplest implementation of **isolated** that preserves these semantics is a single global lock, which is used in this work. However, other implementations are possible in specific cases such as atomic Java variables and software transactional memory.

These implementations can all be considered specific instances of a more general programming model where multiple lock variables are accessed but any given thread can only hold one lock at a time. The mechanized proof has not yet been generalized to this programming model.

For such an effort, Lemma 3 would need to be modified to state that there is at most one causal acquire event for each lock variable whose matching release is not also a causal event. Lemma 6 would then hold by an argument very similar to the one presented for the simpler programming model. Lemma 1, Lemma 2, Lemma 5 and Lemma 7 would still hold as well. It is left to show that $\prec$ is still always antisymmetric. A cycle can only exist in $\prec$ if a causal acquire event's matching release is not a causal event and there is some access event thread ordered after the acquire event that is somehow ordered before a different causal release event on the same lock variable. Such a situation would still be impossible in the more general programming model because of Lemma 1 and Lemma 2.

B. Weaker Partial Order

The mechanized proof is written such that the main theorem can be reused to verify other partial orders over well-formed

Thread 1	Thread 2
acq(l)	
wrt(x)	
rel(l)	
	acq(l)
	wrt(x)
	rel(l)
	wrt(x)

Fig. 1. An execution with a predictable race but no DC race.

```
procedure InsertBacktrackingPoints
    $s \leftarrow \text{peek}(S)$
    $\{t\} \leftarrow \text{backtrack}(s)$
    $p \leftarrow s$
    repeat
        $p \leftarrow \text{pre}(S, p)$
        if $\text{TO}(p) \not\sqsubseteq \text{TO}(s) \wedge \text{DC}(p) \sqsubseteq \text{DC}(s)$ then
            if $\text{tid}(s) \in \text{enabled}(p)$ then
                $\text{backtrack}(p) \leftarrow \text{backtrack}(p) \cup \{t\}$
            else
                $\text{backtrack}(p) \leftarrow \text{enabled}(p)$
            end if
        end if
    until $p = \text{null}$
end procedure
```

Fig. 2. The DPOR algorithm.

histories. Fig. 1 shows a history with a predictable data race that the DC relation fails to detect. The data race is predictable because the events in the observed execution can be rearranged into a valid reordering that witnesses it. However, the DC relation conservatively orders the write events in critical sections even when there may be no dependent read events on that same memory location later in the history, causing it to miss the error.

The Coq model introduced by this work has been used to define another partial order, weaker than the DC relation, that addresses this issue. This partial order only orders conflicting events across critical sections if one is a read event and one is a write event. In order to prove the soundness of this partial order, the definition of data race was modified to exclude accesses that are protected by the same lock. Although this new relation can detect data races missed by the DC relation and has been proven sound on well-formed histories, it has not been proven complete and no algorithm currently exists to compute it. For these reasons, the DC analysis remains the focus of this paper.

IV. Implementation

This section presents a prototype implementation of a sound and complete data race detection analysis, developed as a tool in the Java Pathfinder (JPF) model checker [31]. The source code is available for download at *https://bitbucket.org/byu-vv/jpf-hj/src/FMCAD2019/.*

JPF can execute a program compiled to Java bytecode and expose the program events to tools that consume them. The tools in this work are vector clock algorithms that compute the DC, WCP and HB relations respectively [26], [17], [8]. The

```
procedure OnRead($t, x, isolated$)
    if $isolated$ then
        $C_t \leftarrow C_t \sqcup L^w_x$
        $R_l \leftarrow R_l \cup \{x\}$
    end if
    if $W_x \not\sqsubseteq C_t$ then
        report DC race
    end if
    $R_x(t) \leftarrow C_t(t)$
end procedure
procedure OnWrite($t, x, isolated$)
    if $isolated$ then
        $C_t \leftarrow C_t \sqcup L^w_x \sqcup L^r_x$
        $W_l \leftarrow W_l \cup \{x\}$
    end if
    if $W_x \not\sqsubseteq C_t \vee R_x \not\sqsubseteq C_t$ then
        report DC race
    end if
    $W_x(t) \leftarrow C_t(t)$
end procedure
procedure OnFork($p, c$)
    $C_c \leftarrow C_p$
    $C_c(c) \leftarrow C_c(c) + 1$
    $C_p(p) \leftarrow C_p(p) + 1$
    $P_c \leftarrow P_p$
    $P_c(c) \leftarrow P_c(c) + 1$
    $P_p(p) \leftarrow P_p(p) + 1$
end procedure
procedure OnJoin($p, c$)
    $C_p \leftarrow C_p \sqcup C_c$
    $P_p \leftarrow P_p \sqcup P_c$
end procedure
procedure OnAcquire($t$)
    $s \leftarrow \langle \{t\}, \text{DoneSetRef}() \rangle$
    $\text{push}(S, s)$
end procedure
procedure OnRelease($t$)
    $\text{TO}(\text{peek}(S)) \leftarrow P_t$
    $\text{DC}(\text{peek}(S)) \leftarrow C_t$
    for $x \in R_l$ do
        $L^r_x \leftarrow L^r_x \sqcup C_t$
    end for
    for $x \in W_l$ do
        $L^w_x \leftarrow L^w_x \sqcup C_t$
    end for
    $R_l \leftarrow W_l \leftarrow \emptyset$
    $C_t(t) \leftarrow C_t(t) + 1$
    InsertBacktrackingPoints()
end procedure
```

Fig. 3. The DC data race detection algorithm.

vector clock algorithm implementations are taken directly from related work with almost no modifications other than those specific to JPF. Fig. 3 shows the analysis pseudo code with the DC tool operations inlined for ease of presentation.

A vector clock is a function $VC \in T \rightarrow N$ that maps threads to counter values. Vector clocks also support point-wise comparison ($\sqsubseteq$) and point-wise maximum ($\sqcup$) operations. They are updated in the algorithm such that their comparison is equivalent to set membership in the DC partial order or thread total order.

For each thread t, vector clocks C_t and P_t are maintained to compute the DC relation and thread order respectively. Let $e_i, e_j \in \sigma$ be the last events processed by the analysis on threads t_i and t_j respectively. Then $C_i \sqsubseteq C_j \implies e_i \prec^{\sigma}_{DC} e_j$. Similarly, $P_i \sqsubseteq P_j \implies e_i \prec^{\sigma}_{TO} e_j$.

For each shared memory location x, vector clocks R_x, W_x, L^r_x and L^w_x are also maintained to store the timestamps of threads that access x. R_x and W_x are updated on accesses to x to store the current counter value for the accessing thread so they can be used for checking for data race. For example, in order for a thread t to safely read x, it must have a counter value for every other thread in its own vector clock C_t greater than or equal to the corresponding counter value in W_x. L^r_x and L^w_x are updated on release events to store the maximum counter values for all threads that have accessed x in a critical section. This ensures that the next thread to access x inside of a critical section can update itself with the same max counter values, thus synchronizing itself with the other threads.

Due to the simplified definition of the DC relation for task parallel programming languages, the vector clock queues used in [26] are not necessary. Otherwise, the algorithm for computing the DC relation and reporting data race on access events remains unchanged and the reader should refer to that work for more information. Thread order is extended to accommodate dynamic threads that can be forked and joined. The procedures *OnFork* and *OnJoin* ensure that child threads initially inherit the timestamps of their parents and communicate back any synchronization when they are joined.

JPF runs the input program until completion, passing every thread action and memory access to the tool, allowing the DC vector clock algorithm to analyze the entire history. Because the DC relation is a sound partial order, any detected data race is a real error and is reported to the user. However, the DC relation is not complete, it cannot guarantee that a data race is always detected if one exists in the program. This is due to some programs exhibiting different behavior depending on how the scheduler resolves mutual exclusion. In order to always detect a data race if one exists, the analysis in Fig. 3 uses JPF to rewind the program, reset the analysis state and explore different thread schedules.

JPF executes programs in transitions which are created by the tool. A transition marks a program state and is a tuple $s \in \mathcal{P}(T) \times \mathcal{P}(T)$, identifying a backtrack set of threads and a set of threads already explored from this state, called a done set. The transitions are stored in a stack S and JPF explores the transitions in a depth first manner, always scheduling a thread from the backtrack set of the transition at the top of the stack if it is not also in the transition's done set. The analysis uses a dynamic partial order reduction algorithm (DPOR) [9] to populate the bactrack sets of transitions.

DPOR is a well known result in software model checking that explores a persistent set of transitions from every state by only backtracking on dependent transitions. This results in exploring a reduced state space that is sufficient for verifying safety properties in parallel programs. Dependent means that program behavior may depend on the relative execution order of two transitions. In general, dependent transitions correspond

Thread 1	Thread 2	Thread 3
wrt(x)		
acq(l)		
wrt(x)		
rel(l)		
	acq(l)	
	wrt(y)	
	rel(l)	
		acq(l)
		rd(y)
		wrt(x)
		rel(l)

Fig. 4. An execution with conditional dependence.

to shared memory accesses. However, in the context of data race detection, the only shared memory accesses that are allowed to be enabled in the same state are accesses inside critical sections. Therefore, the DPOR algorithm will cause the model checker to backtrack when two critical sections are dependent.

Dependent critical sections are identified by checking if the release event of the critical section earlier in the history is DC ordered before the release event of the critical section later in the history. If the two release events are not also thread ordered, then their critical sections must contain conflicting events and the model checker must explore an execution where they are performed in the opposite order. Therefore a backtracking point is added to explore the execution where the critical sections are executed in another order. In this way, the partial order used for data race detection directly effects the number of backtracking points added by the DPOR algorithm. Because the DC relation identifies more independent critical sections, it can add fewer backtracking points and is the most effective at reducing the explored state space.

The procedure *OnRelease* associates the current transition with the present values of C_t and P_t to use when searching for dependent transitions. This is done via the utility functions DC and TO which map transitions to the DC and thread order vector clocks associated with their release events.

Transitions are initialized in *OnAcquire* with a backtrack set holding only the current thread and a done set shared between transitions at the same level in the state graph explored by JPF (the *DoneSetRef* function handles this plumbing). The procedure *InsertBacktrackingPoints* implements DPOR in JPF and is invoked at each release event. The procedure traverses the transition stack to find transitions that are dependent with the current transition. If a dependent transition is found, the presently executing thread is added to the backtrack set of the dependent transition. If the desired thread is not enabled in the dependent state, then all enabled threads in that state are conservatively added to the backtrack set in an effort to eventually enable the desired thread.

A. *Correctness of the Analysis*

The analysis uses the DC relation to check each execution for data race while simultaneously identifying dependent critical sections that must be interleaved by the model checker to generate more executions. This work assumes the correctness of the vector clock algorithm that models the DC relation and thread order as they are taken directly from other works. The whole analysis is precise, i.e. it does not report false data races and a data race is always reported if one exists in the program. It does not report false data races because the DC data race detection analysis is sound with respect to each explored execution as shown by the proof in Sec. III.

The analysis always reports a data race if one exists because the DPOR algorithm explores all non-deterministic thread interleavings that could reveal a data race. This property is contingent on using a valid method for detecting dependent transitions. A dependency relation is valid if independent transitions cannot enable or disable one another and if exploring two independent transitions consecutively from the same state always results in the same unique successor state.

Transitions in this paper are created at acquire events and so are not ended until after the matching release event is executed. This means that no transition can disable another since the lock will always be available at the beginning of transitions. Transitions cannot enable one another either unless they are on the same thread, in which case they would not be independent.

Exploring two independent transitions from the same state will always lead to the same unique successor state because independent critical sections access disjoint memory locations. The exception to this rule occurs when a data race exists in the program, in which case it will be detected by the analysis because independent critical sections will not hide the data race by imposing a DC ordering across threads.

The DPOR algorithm in this paper differs from the algorithm in [9] in two significant ways. First, because there is no way to inspect the next transition of every thread before executing it, the algorithm in this paper inserts backtracking points one thread at a time. In contrast, the original DPOR algorithm inserts backtracking points for every thread at every transition. Second, because the DC relation is a conditional dependency relation [16], the algorithm in this paper does not stop inserting backtracking points at the first dependent transition that is found.

As an example, consider the history in Fig. 4. The last write event to the memory location x in thread 3 may depend on the value read by the immediately previous read event on y. In other words, it may be the case that thread 3 only writes to x when its critical section executes after the critical section in thread 2.

Because of this, the DPOR algorithm in this paper cannot stop at the first dependent transition found, as is done in [9]. If this were the case, then in Fig. 4, thread 3 would only be added as a backtracking point in the transition of thread 2 which is the first dependent transition found by the algorithm. But then in the new execution where thread 1 executes and then thread 3 and 2, thread 3 may not write to x and a backtracking point for thread 3 would not be added to the transition of thread 1. In this case, it would actually cause the analysis to miss the data race between thread 1 and thread 3 on x that is detected when the critical sections are executed in the order thread 2, 3 and then 1. Further engineering could relax the requirement of traversing the entire transition stack by keeping track of which shared memory locations have contributed to each dependency and stopping when all memory locations have been accounted for.

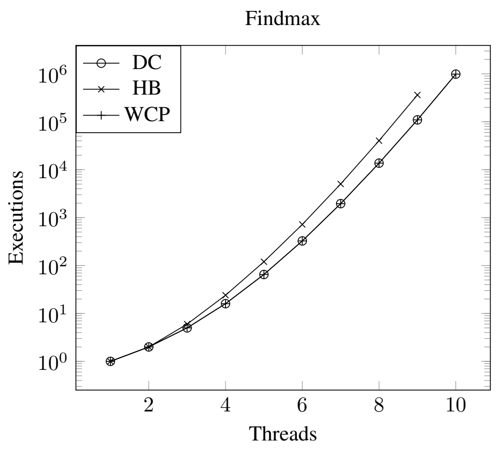

Fig. 5. The Findmax results: explored executions by number of threads.

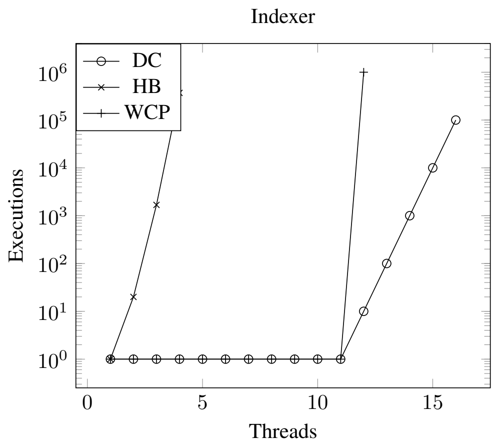

Fig. 6. The Indexer results: explored executions by number of threads.

V. Results

Two benchmarks were used to test how using the DC relation can reduce the number of explored executions required to prove data race freedom in Habanero Java programs. Each benchmark takes the number of threads to use as input and contains no data races. Vector clock analyses for the HB and WCP relation were also implemented and tested.

The first benchmark is a program called *findmax*, which was taken as a snippet of a function in the DualSPHysics project at *https://github.com/DualSPHysics/DualSPHysics*. The code was written in OpenMP and translated to Habanero Java.

The *findmax* benchmark takes a large array of integers as input. Each thread takes a equal size chunk of the array and finds the maximum of the sub-array. An **isolated** region is then used to update the shared value *max* which is the result. The number of executions required to verify the implementation for an increasing number of threads is shown in Fig. 5. The figure shows execution numbers for DC with circles, for HB with $\times$ marks, and for WCP with + marks.

The graph shows that the WCP relation and the DC relation perform identically in this case. They both perform slightly better than the HB relation because they avoid reordering critical sections that only read *max* and do not write to it.

The second benchmark is called *indexer* which is translated into Habanero Java from [9]. In this program, each thread generates integers that are then placed in a shared hash table. The hash table is accessed inside an **isolated** region in case the hash codes conflict. Because most hash codes do not conflict, many of the critical sections are independent. The results are shown for *indexer* and an increasing number of threads in Fig. 6. This figure uses the same convention for execution numbers as does Fig. 5.

The HB relation performs very poorly in this case and cannot tractably analyze a program with more than 4 threads. The composition of the WCP relation with the HB relation has an effect on this benchmark and the WCP relation results in an intractable analysis as soon as the first hash codes conflict. No hash codes conflict until 12 threads are used, so the WCP and DC relation perform identically up to this point, correctly identifying all critical sections as independent. The DC analysis is able to scale further than the other analyses and analyze the program even when it uses 16 threads.

VI. Related Work

As mentioned in Sec. I, common partial orders used for data race detection include the Happens-Before (HB) [18], the Weak-Causally-Precedes (WCP) [17] and the Does-not-Commute (DC) [26] relation. The Schedulable-Happens-Before (SHB) relation [20] strengthens the HB relation in order to guarantee that every data race detected by the relation (not just the first) is a real error. Rather than compute a fixed partial order, the data race detection analysis presented in [24] tries to construct a witness execution for every pair of conflicting events.

Dynamic analyses have been developed specifically for task parallel programs because the simpler programming model allows for algorithmic optimizations. However, many of these analyses cannot reason about the arbitrary synchronization created by critical sections without reporting false data races [7], [25], [15], [29], [6], [21], [30], [28], [33]. Other analyses can reason precisely about mutual exclusion but only use the HB relation to detect data race [22], [34].

Dynamic partial order reduction was originally presented in [9] as a way to exploit information available at run time to reduce the state space explored by software model checkers. That work only considered fixed dependency relations for simplicity. This work and others [16] consider conditional dependency relations, where two transitions may be independent in some contexts but not in others. Dynamic partial order reduction has been implemented in JPF before in [23] but the analysis used a fixed dependency relation and targeted more general concurrent programs and therefore lacked some of the optimizations made possible when analyzing task parallel programs. For example, the task parallel programming model allows JPF to completely implement the DPOR algorithm while only creating transition objects at the beginning of critical sections, instead of detecting shared objects dynamically and creating backtracking objects at memory access events.

A number of enhancements to DPOR have been proposed that are complementary to this work. For example, an analysis may record the context in which backtracking points were created in order to focus the search after backtracking [2]. Other improvements use dynamic information such as the values written to variables, to determine whether transitions may actually be independent [4]. The model checking analysis presented in [10] uses static analysis to reduce the number of memory locations tracked for shared accesses.

As an alternative to dynamic partial order reduction, some analyses use an SMT solver to compute the entire set of other program executions that can be inferred from the observed execution [12], [11], [13]. This approach has the benefit of guaranteeing that the set of executions explored is optimal. And although the analysis can be parallelized effectively, the SMT queries can become huge for even moderately sized programs. This paper has assumed a sequentially consistent memory model while other model checking analyses have been extended to weaker models [14].

Chen et al. propose a parametric framework for proving that a partial order is sound [5]. The framework will yield a soundness proof for a given partial order as a corollary of its main theorem. The main theorem is a statement about traces and says that any property that holds over the operation of swapping an adjacent pair of events in a trace that are unrelated by the partial order will hold for every linearization of the partial order. However, this definition of sound is too strong for the goal of this work as the desired property is that of valid reordering and the DC relation yields linearizations that are not valid traces. Therefore the proof presented in this work only proves that a valid reordering exists for every data race, not that every linearization of the partial order is a valid reordering.

Many claims with accompanying theorems and proofs are made in the literature of data race detection. As they get more difficult to reason about and implement, mechanized proofs have helped confirm that the work is correct. FastTrack implementations have been verified in Coq to ensure that the instrumentation and synchronization used in the data race detection do not introduce error [19], [32].

VII. Conclusion

This work presented a proof that the DC relation is sound for a class of concurrent programs that includes commonly used task parallel programming models. A prototype analysis in the JPF model checker demonstrated the benefit of using a weaker partial order for data race detection. The mechanized proof provides a foundation for finding more insights and justifying stronger claims in the future.

References

[1] Habanero-Java - Habanero - Rice University campus wiki. https://wiki.rice.edu/confluence/display/HABANERO/Habanero-Java.

[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *ACM SIGPLAN Notices*, volume 49, pages 373–384. ACM, 2014.

[3] Vincent Cav, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. 08 2011.

[4] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 2(POPL):31, 2017.

[5] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *International Conference on Computer Aided Verification*, pages 240–253. Springer, 2007.

[6] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[7] Mingdong Feng and Charles E Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[8] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[9] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, January 2005.

[10] M. Gligoric, P. C. Mehlitz, and D. Marinov. X10x: Model checking a new programming language with an "old" model checker. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 11–20, April 2012.

[11] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *ACM SIGPLAN Notices*, volume 50, pages 165–174. ACM, 2015.

[12] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. *SIGPLAN Not.*, 49(6):337–348, June 2014.

[13] Jeff Huang and Arun K. Rajagopalan. Precise and maximal race detection from incomplete traces. *SIGPLAN Not.*, 51(10):462–476, October 2016.

[14] Shiyou Huang and Jeff Huang. Maximal causality reduction for tso and pso. In *ACM SIGPLAN Notices*, volume 51, pages 447–461. ACM, 2016.

[15] Weixing Ji, Li Lu, and Michael L Scott. Tardis: Task-level access race detection by intersecting sets. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet), Houston, TX*, 2013.

[16] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.

[17] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. *SIGPLAN Not.*, 52(6):157–170, June 2017.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[19] William Mansky, Yuanfeng Peng, Steve Zdancewic, and Joseph Devietti. Verifying dynamic race detection. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 151–163. ACM, 2017.

[20] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):145:1–145:29, October 2018.

[21] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33. IEEE, 1991.

[22] Radha Nakade, Eric Mercer, Peter Aldous, and Jay McCarthy. Model-checking task parallel programs for data-race. In Aaron Dutle, César Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods*, pages 367–382, Cham, 2018. Springer International Publishing.

[23] Eric Noonan, Eric Mercer, and Neha Rungta. Vector-clock based partial order reduction for jpf. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.

[24] Andreas Pavlogiannis. Fast, sound and effectively complete dynamic race detection. *CoRR*, abs/1901.08857, 2019.

[25] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *International Conference on Runtime Verification*, pages 368–383. Springer, 2010.

[26] Jake Roemer, Kaan Genç, and Michael D Bond. High-coverage, unbounded sound predictive race detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 374–389. ACM, 2018.

[27] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler. In *International Conference on Runtime Verification*, pages 110–114. Springer, 2011.

[28] Rishi Surendran and Vivek Sarkar. Dynamic determinacy race detection for task parallelism with futures. In *International Conference on Runtime Verification*, pages 368–385. Springer, 2016.

[29] Robert Utterback, Kunal Agrawal, Jeremy Fineman, I Lee, and Ting Angelina. Efficient race detection with futures. *arXiv preprint arXiv:1901.00622*, 2019.

[30] Robert Utterback, Kunal Agrawal, Jeremy T Fineman, I Lee, et al. 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*, pages 83–94. ACM, 2016.

[31] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated software engineering*, 10(2):203–232, 2003.

[32] James R Wilcox, Cormac Flanagan, and Stephen N Freund. Verified ft: a verified, high-performance precise dynamic race detector. In *ACM SIGPLAN Notices*, volume 53, pages 354–367. ACM, 2018.

[33] 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. ACM, 2016.

[34] Lechen Yu and Vivek Sarkar. Gt-race: graph traversal based data race detection for asynchronous many-task parallelism. In *European Conference on Parallel Processing*, pages 59–73. Springer, 2018.