Non-Preemptive Semantics for Data-Race-Free Programs*

Siyang Xiao[1], Hanru Jiang[1], Hongjin Liang[2], and Xinyu Feng[2]

[1] University of Science and Technology of China, Hefei, 230027, China
{yutio888, hanru219}@mail.ustc.edu.cn
[2] State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, 210023, China
{hongjin, xyfeng}@nju.edu.cn

Abstract. It is challenging to reason about the behaviors of concurrent programs because of the non-deterministic interleaving execution of threads. To simplify the reasoning, we propose a non-preemptive semantics for data-race-free (DRF) concurrent programs, where a thread yields the control of the CPU only at certain carefully-chosen program points. We formally prove that DRF concurrent programs behave the same in the standard interleaving semantics and in our non-preemptive semantics. We also propose a novel formulation of data-race-freedom in our non-preemptive semantics, called NPDRF, which is proved equivalent to the standard DRF notion in the interleaving semantics.

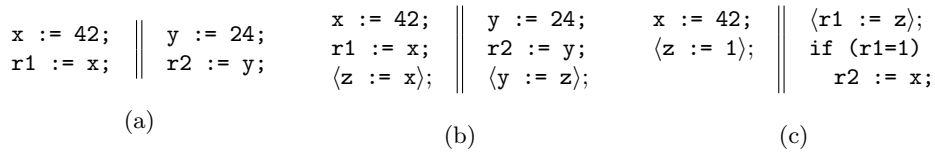
Keywords: data-race-freedom · interleaving semantics · non-preemptive semantics.

1 Introduction

Interleaving semantics has been widely used as standard operational semantics for concurrent programs, where the execution of a thread can be preempted at any program point and the control is switched to a different thread. Reasoning about multi-threaded concurrent programs in this semantics is challenging because the number of possible interleaving executions can be exponential (with respect to the length of the program).

On the other hand, for a large class of programs, it is a waste of effort to enumerate and verify all the possible interleavings, because many of them actually lead to the same result. For instance, the simple program in Fig. 1(a) has six possible interleavings, but all of these interleavings result in the same final state ($\texttt{x} = \texttt{r1} = 42$, $\texttt{y} = \texttt{r2} = 24$). Thus, to analyze the final result of the program in Fig. 1(a), we can reason about only one interleaving instead of all the six, which dramatically reduces the verification effort. The question is, can we *systematically* reduce the number of interleavings without reducing the possible behaviors of a concurrent program?

* This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61502442 and 61632005.



(a) (b) (c)

Fig. 1. Data-race-free programs

Actually it is well-known that, for data-race-free (DRF) programs, we only need to consider interleavings at synchronization points. Informally a data race occurs when multiple threads access the same memory location concurrently and at least one of the accesses is a write. All the three programs in Fig. 1 are data-race-free. Here we use the atomic statement $\langle S \rangle$ to mean that the execution of S is atomic, i.e. it cannot be interrupted by other threads. Thus the two accesses of `z` in Fig. 1(b) are well synchronized, and the program is data-race-free. In Fig. 1(c), although both threads access the shared variable `x` outside of the atomic statement, they cannot be accessed *concurrently* (assuming the initial value of `z` is 0) because the read of `x` in the second thread can only be executed (if executed at all) after the write of `z`, which then happens after the write of `x`.

Although it is a folklore theorem that the behaviors of DRF programs under the standard interleaving semantics should be equivalent to those in some non-preemptive semantics where a thread yields the control of the CPU at will at certain program points, it is not obvious where these program points should be, especially if we want to minimize such program points to reduce as many possible interleavings as possible. And what if the language allows other effects other than memory accesses, such as I/O operations? Would non-termination of programs affect the choice of these program points?

For instance, a straightforward approach is to treat both the entry and the exit of atomic blocks $\langle S \rangle$ as program points to allow interleaving (i.e.switching between threads). But can we just use only one of them instead of both? If yes, are the entry and the exit points equally good? It is quite interesting to see that we can pick the exit of the block as the only switching point, and the behaviors under preemptive semantics are still preserved. However, the entry point does not work, which may lead to strictly less behaviors than preemptive semantics if the program following the atomic block does not terminate.

In this paper we formally study the non-preemptive semantics of DRF programs, and discuss the possible variations of the semantics and how they affect the equivalence with the preemptive semantics. The paper makes the following contributions:

- We define the notion of DRF operationally based on the preemptive operational semantics.
- We propose non-preemptive semantics for DRF programs. In addition to memory accesses, our language allows externally observable operations such as I/O operations. Threads in the semantics can be preempted only at the

end of each atomic block or after the I/O operations. We discuss how non-termination could affect the choice of switching points.
- We define the semantics equivalence based on the set of execution traces in each semantics. Then we formally prove that our non-preemptive semantics is equivalent to the preemptive one for DRF programs.
- We also give a new operational notion of DRF in the non-preemptive semantics, which is called NPDRF. We prove that NPDRF is equivalent to the original definition of DRF in the preemptive semantics. This allows one to study DRF programs fully within the non-preemptive semantics.

Related Work. Non-preemptive (or cooperative) semantics has been studied in various settings, for example in high scalability thread libraries [4, 11, 14], as alternative models for structuring or reasoning about concurrency [6, 1, 15, 12], and in program analysis [16].

Beringer *et el.* [5] made a proposal of proving compilation correctness under a cooperative setting in order to reuse the compiler correctness proofs for sequential settings. Their proposal is based on the conjecture that when the source program is proved to be data-race-free, its behavior would be the same under cooperative semantics. We prove this conjecture in this paper.

Ferreira *et el.* [8] proposed a grainless semantics relating concurrent separation logic and relaxed memory models. Their grainless semantics execute non-atomic instructions in big steps, and cannot be interrupted by other threads, which is similar to cooperative semantics where program sections between atomic blocks are not interfered by other threads. They also proved DRF programs behave the same under interleaving semantics and grainless semantics in their setting. Our semantics and formulation of DRF are sufficiently different from theirs. Moreover, there is no in-depth discussion on how non-termination of programs affects the choice of the switching points to minimize the interleaving.

Collingbourne *et el.* [7] and Kojima *et el.* [10] studied the equivalence between interleaving semantics and the lock-step semantics usually used in graphics processing units (GPUs). They found that race-free programs executed in the standard interleaving semantics and the lock-step semantics should get the same result. However, the lock-step semantics require that each thread execute exactly the same set of operations in parallel. It is designed specially for GPU and is sufficiently different from the non-preemptive semantics we study here.

There has been much work formalizing DRF, e.g., DRF-0 [2] and DRF-1 [3]. Marino *et el.* [13] proposed a memory model called DRFx. They proposed a new concept called region conflict freedom, which requires no conflicting memory accesses between program sections instead of instructions, therefore enables efficient SC violation detecting. Their notion of region conflict freedom is similar to our NPDRF: regions are code snippets no larger than code snippets between critical sections, while in our NPDRF, we compare memory accesses of executions between atomic steps. They did not have a formal operational formulation as we do. Hower et al. [9] proposed the notion of heterogeneous-race-free (SC-HRF) for

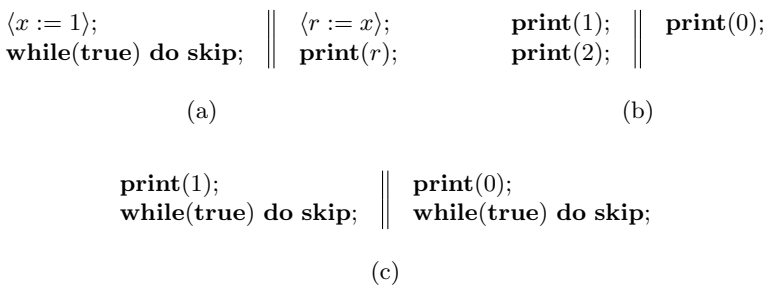


Fig. 2. More examples about non-preemptive executions

scoped synchronization in heterogeneous systems, which is for a different setting from ours.

Organizations. In the rest of this paper, we make some informal discussion about non-preemptive semantics in Sec. 2. Then we introduce the basic technical settings about the language and the preemptive semantics in Sec. 3. In Sec. 4, we give the definition of our non-preemptive semantics and discuss the equivalence of preemptive semantics and non-preemptive semantics. In Sec. 5, we discuss the notion of data-race-freedom in our non-preemptive semantics (called NPDRF) and the equivalence of NPDRF and DRF. Sec. 6 concludes this paper.

2 Informal discussions of the non-preemptive semantics

In this section we informally compare the program behaviors in preemptive semantics and non-preemptive semantics with different choices of switching points, based on which we give a principle to ensure the semantics equivalence.

In Fig. 2(a), assuming the initial value of `x` is 0, the program may either print out 1 or print out 0 in the preemptive semantics, or generate no output at all if the scheduling is unfair. In non-preemptive semantics, if we allow switching at the exit of atomic blocks, we can get the same set of possible behaviors. However, if we choose the entry as the only switching point, it is impossible to print out 1. This is because the left thread does not terminate after the write of `x`, so there is no chance for the second thread to run.

Our language has the **print**(e) command as a representative I/O command. When we observe program behaviors, we observe the sequences of externally observable I/O events. To allow the non-preemptive semantics to generate the same set of event sequences as in the preemptive semantics, we must allow switching at each I/O command. In Fig. 2(b), it would be impossible to observe the sequence "102" if we disallow switching at each print command.

This is easy to understand if we view each print command as a write to a shared tape[3]. However, it would be interesting to see what would happen if we project the whole output sequence to each thread and observe the resulting set of subsequences instead of the whole sequence. That is, we assume each thread has its own tape for outputs. For the program in Fig. 2(b), we can observe the subsequence "12" for the left thread and "0" for the right, no matter we allow switching at the print command or not.

However, the next example in Fig. 2(c) shows that, even if we only observe thread-local output sequences, we still need to treat the print command as an switching point, otherwise it would be impossible to see both outputs in non-preemptive semantics.

Fig. 2(a) and (c) show that non-termination of the code segment between synchronization (or I/O) points plays an important role when we choose the switching points. Essentially this is because, in non-preemptive semantics, non-termination of these code segments prevents the switching from the current thread to another. Although such code segment never accesses shared resources in DRF programs, they should be viewed similarly as the accesses of shared resources — they all generate external effects affecting other threads.

Based on the above discussion, we follow the principle below when deciding switching points in our non-preemptive semantics to ensure its equivalence to the preemptive semantics:

There must be at least one switching point between any two consecutive externally observable effects generated at runtime in the same thread.

Here *externally observable effects* are those that either affect behaviors of other threads or generate externally observable events.

It is interesting to note that if a non-terminating code segment comes before an atomic block, there is no switching point in between if we do not treat the entry of the atomic block as an switching point. This actually does *not* violate the above principle because the atomic block never gets executed if it follows a non-terminating code segment. Therefore there are NO two consecutive externally observable effects generated at runtime.

In the following sections we formalize the ideas in our definition of non-preemptive semantics and prove that it preserves the behaviors of DRF programs in preemptive semantics.

3 The Language and Preemptive Semantics

The syntax of the language is shown in Fig. 3. In the language we distinguish variables from memory cells. The whole program P consists of n sequential threads, each with its own local variables (like thread-local registers). They communicate through a shared heap (i.e. memory).

The arithmetic expressions e and boolean expressions b are pure in that they do not access the heap. The commands $x := [e]$ and $[e] := e'$ reads and

[3] In this case, we don't view two concurrent print commands as a data race. Instead, we assume there is an implicit enclosing atomic block for each $\mathbf{print}(e)$.

$$
\begin{array}{ll}
(\mathit{Expr}) & e ::= x \mid \mathbf{n} \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots \\
(\mathit{Bexp}) & b ::= \mathbf{true} \mid \mathbf{false} \mid e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \ldots \\
(\mathit{Prim}) & c ::= x := e \mid x := [e] \mid [e] := e' \mid \mathbf{print}(e) \\
(\mathit{Stmt}) & S ::= c \mid \mathbf{skip} \mid S; S \mid \langle S \rangle \mid \mathbf{if}(b)\ S_1\ \mathbf{else}\ S_2 \mid \mathbf{while}(b)\ \mathbf{do}\ S \\
(\mathit{Prog}) & P ::= S_1 \parallel \ldots \parallel S_n
\end{array}
$$

Fig. 3. Syntax of the language

writes heap cells at the location e, respectively. The $\mathbf{print}(e)$ command generates externally observable output of the value of e.

The atomic statement $\langle S \rangle$ executes S sequentially, which cannot be interrupted by other threads. It can be viewed as a convenient abstraction for hardware supported atomic operations (such as the `cas` instruction). Here S cannot contain other atomic blocks or print commands. This is enforced in our operational semantics rules presented below.

The state model is defined in Fig. 4. The world W contains the program P, the id of the current thread `t` and the program state σ. The state σ contains a heap h, a mapping ls that maps each thread id to its local store s, and a binary flag `d` indicating whether the current thread is executing inside an atomic block.

$$
\begin{array}{rlll}
(\mathit{World}) & W & ::= & (P, \mathtt{t}, \sigma) \\
(\mathit{ThrdId}) & \mathtt{t} & \in & \mathbb{N} \\
(\mathit{Addr}) & a & \in & \mathbb{N} \\
(\mathit{Store}) & s & \in & \mathit{Var} \to \mathit{Int} \\
(\mathit{StoreList}) & ls & \in & \mathit{ThrdID} \to \mathit{Store} \\
(\mathit{Heap}) & h & \in & \mathit{Addr} \rightharpoonup \mathit{Int} \\
(\mathit{Bit}) & \mathtt{d} & ::= & 0 \mid 1 \\
(\mathit{State}) & \sigma & ::= & (h, ls, \mathtt{d}) \\
(\mathit{LocSet}) & rs, ws & \in & \mathcal{P}(\mathit{Addr}) \\
(\mathit{FootPrint}) & \delta & ::= & (rs, ws) \qquad \mathtt{emp} \stackrel{\text{def}}{=} (\emptyset, \emptyset) \\
(\mathit{Label}) & \iota & ::= & \gamma \mid \mathbf{out}\ n \\
(\mathit{iLabel}) & \gamma & ::= & \tau \mid \mathbf{sw} \mid \mathbf{atm}
\end{array}
$$

Fig. 4. Runtime constructs and footprints

Footprint-based Semantics. Thread execution is defined as labeled transition in the form of $(S, (h, s)) \xmapsto[\delta]{\iota} (S', (h', s'))$. Figure 5 shows selected rules for

thread-local transitions. Each step is associated with a label ι and a footprint δ. The label contains the information about this step. There are two class of labels, the internal labels γ and the externally observable output event (**out** n). An internal label γ can record a step inside an atomic block (**atm**), a context switch (**sw**, which is used only in the whole program transitions in Fig. 6), or a regular silent step (τ). Note the ATOM rule only allows τ-steps inside the atomic block, which are converted to **atm**-steps looking from outside of the block. Therefore the block $\langle S \rangle$ cannot contain other atomic blocks or print commands in S.

The footprint δ is defined as a pair (rs, ws), where rs and ws are the sets of memory locations read or written during the transition. The record of footprint allows us to define data-races below. When a step makes no memory accesses, the footprint is defined as `emp`, where rs and ws are both empty set.

$$\frac{[\![e]\!]_s = n \quad s' = s[x \rightsquigarrow n]}{(x := e, (h, s)) \xmapsto[\texttt{emp}]{\tau} (\mathbf{skip}, (h, s'))}\ (\textsc{assn}) \qquad \frac{[\![e]\!]_s = l \quad h[l] = n \quad s' = s[x \rightsquigarrow n]}{(x := [e], (h, s)) \xmapsto[(\{l\},\emptyset)]{\tau} (\mathbf{skip}, (h, s'))}\ (\textsc{ld})$$

$$\frac{[\![e]\!]_s = l \quad [\![e']\!]_s = n \quad l \in \mathsf{dom}(h) \quad h' = h[l \rightsquigarrow n]}{([e] := e', (h, s)) \xmapsto[(\emptyset,\{l\})]{\tau} (\mathbf{skip}, (h', s))}\ (\textsc{st}) \qquad \frac{[\![e]\!]_s = l \quad l \notin \mathsf{dom}(h)}{([e] := e', (h, s)) \xmapsto[\texttt{emp}]{\tau} \mathbf{abort}}\ (\textsc{st-abt})$$

$$\frac{[\![e]\!]_s = n}{(\mathbf{print}(e), (h, s)) \xmapsto[\texttt{emp}]{\mathbf{out}\ n} (\mathbf{skip}, (h, s))}\ (\textsc{prt}) \qquad \frac{(S, (h, s)) \xmapsto[\delta]{\tau} (S', (h', s'))}{(\langle S \rangle, (h, s)) \xmapsto[\delta]{\mathbf{atm}} (\langle S' \rangle, (h', s'))}\ (\textsc{atom})$$

$$\frac{}{(\langle \mathbf{skip} \rangle, (h, s)) \xmapsto[\texttt{emp}]{\mathbf{atm}} (\mathbf{skip}, (h, s))}\ (\textsc{atom-end})$$

Fig. 5. Selected rules for thread-local transitions

Transitions of the global configuration W are defined in the form of $W \overset{\iota}{\underset{\delta}{\Rightarrow}} W'$ in Fig. 6. The THRD rule shows a step of execution outside of atomic blocks. The flag `d` must be 0 in this case. It is set to 1 when executing inside the atomic block (the ATOMIC rule), and reset to 0 at the end (the ATOMIC-END rule). The SWITCH rule says the execution of the current thread can be switched to a different one at any time as long as the current thread is not executing an atomic block (i.e. the bit `d` must be 0). Here we use the label **sw** to indicate this is a switch step. Its use will be explained below.

W may also lead to **abort** if the execution of the current thread aborts (the ABT rule). It leads to a special configuration **done** if every individual thread terminates (the DONE rule).

Multi-step Transitions. We use $(S, (h, s)) \xmapsto[\delta]{\iota}{}^{+} (S', (h', s'))$ to represent multi-step transitions that the label of each transition is ι. Here δ is the accumulation

$$\frac{P(\mathtt{t}) = S \quad ls(\mathtt{t}) = s \quad \iota \neq \mathbf{atm} \quad (S,(h,s)) \xmapsto[\delta]{\iota} (S',(h',s')) \quad ls' = ls[\mathtt{t} \rightsquigarrow s']}{(P,\mathtt{t},(h,ls,0)) \overset{\iota}{\underset{\delta}{\Rightarrow}} (P[\mathtt{t} \rightsquigarrow S'],\mathtt{t},(h',ls',0))} \text{ (THRD)}$$

$$\frac{\begin{array}{c} P(\mathtt{t}) = S \quad S = \langle S_2 \rangle \vee S = \langle S_2 \rangle; S_3 \quad ls(\mathtt{t}) = s \\ S_2 \neq \mathbf{skip} \quad (S,(h,s)) \overset{\mathbf{atm}}{\underset{\delta}{\longmapsto}} (S',(h',s')) \quad ls' = ls[\mathtt{t} \rightsquigarrow s'] \end{array}}{(P,\mathtt{t},(h,ls,\mathtt{d})) \overset{\mathbf{atm}}{\underset{\delta}{\Longrightarrow}} (P[\mathtt{t} \rightsquigarrow S'],\mathtt{t},(h',ls',1))} \text{ (ATOMIC)}$$

$$\frac{P(\mathtt{t}) = S \quad S = \langle \mathbf{skip} \rangle \vee S = \langle \mathbf{skip} \rangle; S_2 \quad ls(\mathtt{t}) = s \quad (S,(h,s)) \overset{\mathbf{atm}}{\underset{\mathtt{emp}}{\longmapsto}} (S',(h,s))}{(P,\mathtt{t},(h,ls,\mathtt{d})) \overset{\mathbf{atm}}{\underset{\mathtt{emp}}{\Longrightarrow}} (P[\mathtt{t} \rightsquigarrow S'],\mathtt{t},(h,ls,0))} \text{ (ATOMIC-END)}$$

$$\frac{P(\mathtt{t}) = S \qquad ls(\mathtt{t}) = s \qquad (S,(h,s)) \xmapsto[\delta]{\iota} \mathbf{abort}}{(P,\mathtt{t},(h,ls,\mathtt{d})) \overset{\iota}{\underset{\delta}{\Rightarrow}} \mathbf{abort}} \text{ (ABT)}$$

$$\frac{P(\mathtt{t}') \neq \mathbf{skip}}{(P,\mathtt{t},(h,s,0)) \overset{\mathbf{sw}}{\underset{\mathtt{emp}}{\Longrightarrow}} (P,\mathtt{t}',(h,s,0))} \text{ (SWITCH)} \qquad \frac{P = \mathbf{skip} \parallel \cdots \parallel \mathbf{skip}}{(P,\mathtt{t},\sigma) \overset{\tau}{\underset{\mathtt{emp}}{\Rightarrow}} \mathbf{done}} \text{ (DONE)}$$

Fig. 6. Selected rules for global transitions

of the footprints generated. We may ommit the label or the footprint when they are irrelevant in the context. Similarly $\xmapsto[\delta]{\iota}{}^*$ represents transitions of zero or multiple steps. In the case of zero step, δ is $\mathtt{emp}$. $W \overset{\iota}{\underset{\delta}{\Rightarrow}}{}^+ W'$ and $W \overset{\iota}{\underset{\delta}{\Rightarrow}}{}^* W'$ are similarly defined in the global semantics. In particular, $W \overset{\gamma}{\underset{\delta}{\Rightarrow}}{}^* W'$ means only internal labels γ are generated during the transitions. The labels in these steps can be τ, **atm** or **sw**. Labels in different steps do not have to be the same in this case. We also use a natural number k as the superscript to indicate a k-step transition.

Event Traces and Program Behaviors. The behavior of a concurrent program is defined as an externally observable event trace $\mathcal{B}$, which is a finite or infinite sequence of output values generated in the output event (**out** n), with possible ending events **done** or **abort**. An empty trace is represented as ϵ. Traces are co-inductively defined in Fig. 7.

The trace ends with **done** or **abort** if the execution terminates normally or aborts, respectively. When the program generates observable events (by the **print** command), the output value is put on the trace. Co-inductively defined, a trace can be infinite, which represents a diverging execution generating infinite number of outputs.

$$\frac{W \overset{\gamma}{\Rightarrow}^{+} \textbf{abort}}{Etr(W, \textbf{abort})} \qquad \frac{W \overset{\gamma}{\Rightarrow}^{*} W' \quad W' \overset{\textbf{out}\ n}{\Longrightarrow} W'' \quad Etr(W'', \mathcal{B})}{Etr(W, n :: \mathcal{B})}$$

$$\frac{W \overset{\gamma}{\Rightarrow}^{+} \textbf{done}}{Etr(W, \textbf{done})} \qquad \frac{W \overset{\gamma}{\Rightarrow}^{*} W' \quad W' \overset{\tau/\textbf{atm}}{\Longrightarrow} W'' \quad Etr(W'', \epsilon)}{Etr(W, \epsilon)}$$

$$ProgEtr((P, \sigma), \mathcal{B}) \text{ iff } \exists \mathtt{t}. Etr((P, \mathtt{t}, \sigma), \mathcal{B})$$

Fig. 7. Definition of event trace in preemptive semantics

We allow a trace to be finite but not end with **done** or **abort**. In this case, after generating the last output recorded on the trace, the program runs forever but does not generate any more observable events (called silent divergence). Note our definition of silent divergence in the last rule in Fig. 7 requires there must always be non-switch steps (which can be τ steps or **atm** steps) executing. This prevents the execution that keeps switching between threads but does not execute any code.

Definition of Data Races. Below we first define conflict of footprints in Def. 1, which indicates conflicting accesses of shared memory. Two footprints δ and δ' are conflicting, i.e. $\delta \frown \delta'$, if there exists a memory location in one of them also shows up in the *write* set of the other.

Definition 1 (Conflicting footprints). $\delta \frown \delta'$ iff $((\delta.rs \cap \delta'.ws \neq \emptyset) \vee (\delta.ws \cap \delta'.rs \neq \emptyset) \vee (\delta.ws \cap \delta'.ws \neq \emptyset))$

In Fig. 8, we define data races operationally in the preemptive semantics. The key idea is that, during the program execution, we predict the footprints of any two threads and see if they are conflicting. Note that we only do the prediction at switching points. That is, footprints of threads are predicted only when the threads can indeed be switched to run. We cannot do the prediction when executing inside an atomic block, where switching is disallowed.

The first two rules inductively define the predicate $predict(W, \mathtt{t}, \delta, \mathtt{d})$. Suppose we execute thread $\mathtt{t}$ in W (which may or may not be the current thread $\mathtt{t}'$) zero or multiple steps. The accumulated footstep is δ. We let $\mathtt{d}$ be 1 if the predicted steps are in an atomic block, and 0 otherwise.

Then $W \Longmapsto \mathtt{Race}$ if there exist two threads whose predicted footprints are conflicting, and at least one of them is not executing an atomic block. Since we assume that atomic blocks cannot be executed at the same time, atomic blocks that generate conflicting footprints are not considered as a data race.

We define $(P, \sigma) \Longmapsto \mathtt{Race}$ if it predicts a data race after zero or multiple steps of execution starting from a certain thread. $\mathsf{DRF}(P, \sigma)$ holds if (P, σ) never reaches a race.

In both PREDICT-0 and PREDICRT-1 rules, the flag in W must be 0, indicating the current thread $\mathtt{t}'$ is not inside an atomic block (so the execution is

$$\frac{W = (P, \mathtt{t}', (h, ls, 0)) \quad P(\mathtt{t}) = S \quad ls(\mathtt{t}) = s \quad (S, (h, s)) \xmapsto[\delta]{\tau}{}^{*} (S', (h', s'))}{predict(W, \mathtt{t}, \delta, 0)} \text{ (PREDICT-0)}$$

$$\frac{W = (P, \mathtt{t}', (h, ls, 0)) \quad P(\mathtt{t}) = S \quad ls(\mathtt{t}) = s \quad (S, (h, s)) \xmapsto[\delta]{\mathbf{atm}}{}^{*} (S', (h', s'))}{predict(W, \mathtt{t}, \delta, 1)} \text{ (PREDICT-1)}$$

$$\frac{\begin{array}{c}\mathtt{t}_1 \neq \mathtt{t}_2 \quad predict(W, \mathtt{t}_1, \delta_1, \mathtt{d}_1) \quad predict(W, \mathtt{t}_2, \delta_2, \mathtt{d}_2) \\ \delta_1 \frown \delta_2 \quad (\mathtt{d}_1 = 0 \vee \mathtt{d}_2 = 0)\end{array}}{W \Longmapsto \mathtt{Race}} \text{ (RACE)}$$

$$\frac{(P, \mathtt{t}, \sigma) \Rightarrow^{*} W \qquad W \Longmapsto \mathtt{Race}}{(P, \sigma) \Longmapsto \mathtt{Race}} \qquad \frac{\neg (P, \sigma) \Longmapsto \mathtt{Race}}{\mathsf{DRF}(P, \sigma)}$$

Fig. 8. Definition of races and data-race-freedom

```
⟨[0] := 42;  ‖  ⟨x := [0];                          ‖  [1] := 42;
 [0] := 0;⟩  ‖   if(x = 42) [1] := 42 else skip ⟩   ‖
```

Fig. 9. Example of an DRF program

at a switching point). Otherwise the predict rule is able to make use of the intermediate state during the execution of an atomic block that is invisible to other threads, and predict conflicting footprints that is not possible during execution. We can see this problem from the example program in Fig. 9. Assuming the heap cells are initialized with 0, it is easy to see the program is race-free since the second thread has no chance to write to the memory location 1. However, if we permit prediction inside an atomic block, we are able to make prediction at the program point right after the statement (`[0] := 42;`) in the first thread. Then in the predicted execution the second thread can reach the first branch of the conditional statement and write to location 1 since location 0 now contains 42. We can also predict that the third thread writes to location 1 as well. This kind of conflicting footprints would never be generated during the actual execution of the program and should not be considered as data race.

4 Non-Preemptive Semantics

Below we define our non-preemptive semantics, and prove its equivalence with preemptive semantics for DRF programs. As explained before, the key point of non-preemptive semantics is to reduce the potential interleaving in concurrency. It is done by limiting thread-switching to certain program points (called switching points). Thus the code fragment between switching points can be reasoned about as sequential code, and interleaving is only considered at those switching points.

4.1 Semantics

$$\frac{P(\mathtt{t}) = S \quad ls(\mathtt{t}) = s \quad ls' = ls[\mathtt{t} \rightsquigarrow s'] \quad (S,(h,s)) \xmapsto[\delta]{\tau} (S',(h',s'))}{(P,\mathtt{t},(h,ls,0)) :\underset{\delta}{\overset{\tau}{\Rightarrow}} (P[\mathtt{t} \rightsquigarrow S'],\mathtt{t},(h',ls',0))} \text{(NP-THRD)}$$

$$\frac{\begin{array}{c} P(\mathtt{t}) = S \qquad S = \langle S_a \rangle \vee S = \langle S_a \rangle; S_b \qquad S_a \neq \mathbf{skip} \\ ls(\mathtt{t}) = s \qquad ls' = ls[\mathtt{t} \rightsquigarrow s'] \qquad (S,(h,s)) \xmapsto[\delta]{\mathbf{atm}} (S',(h',s')) \end{array}}{(P,\mathtt{t},(h,ls,\mathtt{d})) :\underset{\delta}{\overset{\mathbf{atm}}{\Longrightarrow}} (P[\mathtt{t} \rightsquigarrow S'],\mathtt{t},(h',ls',1))} \text{(NP-ATOM)}$$

$$\frac{\begin{array}{c} P(\mathtt{t}) = S \qquad S = \langle \mathbf{skip} \rangle \vee S = \langle \mathbf{skip} \rangle; S'' \qquad ls(\mathtt{t}) = s \\ (S,(h,s)) \xmapsto[\mathtt{emp}]{\mathbf{atm}} (S',(h,s)) \qquad \mathtt{t}' = \mathtt{t} \vee P(\mathtt{t}') \neq \mathbf{skip} \end{array}}{(P,\mathtt{t},(h,ls,\mathtt{d})) :\underset{\mathtt{emp}}{\overset{\mathbf{sw}}{\Longrightarrow}} (P[\mathtt{t} \rightsquigarrow S'],\mathtt{t}',(h,ls,0))} \text{(NP-ATOM-SW)}$$

$$\frac{P(\mathtt{t}) = S \qquad (S,(h,ls)) \xmapsto[\mathtt{emp}]{\mathbf{out}\ n} (S',(h,ls)) \qquad \mathtt{t}' = \mathtt{t} \vee P(\mathtt{t}') \neq \mathbf{skip}}{(P,\mathtt{t},(h,ls,0)) :\underset{\mathtt{emp}}{\overset{\mathbf{out}\ n}{\Longrightarrow}} (P[\mathtt{t} \rightsquigarrow S'],\mathtt{t}',(h,ls,0))} \text{(OUT-SW)}$$

$$\frac{P(\mathtt{t}) = \mathbf{skip} \qquad \sigma.\mathtt{d} = 0 \qquad P(\mathtt{t}') \neq \mathbf{skip}}{(P,\mathtt{t},\sigma) :\underset{\mathtt{emp}}{\overset{\mathbf{sw}}{\Longrightarrow}} (P,\mathtt{t}',\sigma)} \text{(END-SW)}$$

$$\frac{P = \mathbf{skip} \parallel \cdots \parallel \mathbf{skip}}{(P,\mathtt{t},\sigma) :\underset{\mathtt{emp}}{\overset{\tau}{\Rightarrow}} \mathbf{done}} \text{(NP-DONE)}$$

$$\frac{P(\mathtt{t}) = S \qquad ls(\mathtt{t}) = s \qquad (S,(h,s)) \xmapsto[\mathtt{emp}]{\tau} \mathbf{abort}}{(P,\mathtt{t},(h,ls,\mathtt{d})) :\underset{\mathtt{emp}}{\overset{\tau}{\Rightarrow}} \mathbf{abort}} \text{(NP-ABT)}$$

Fig. 10. Non-preemptive Semantics

The non-preemptive semantics is defined in Fig. 10. We use three switching rules in our non-preemptive semantics, i.e. the NP-ATOM-SW rule, the OUT-SW rule and the END-SW rule, to show that switching can occur only at the end of atomic blocks, the **print** command, and the end of the current thread, respectively. The other rules are similar to their counterparts in the preemptive semantics.

Event Traces in Non-preemptive Semantics. The definition of event traces in the non-preemptive semantics is almost the same as that in preemptive semantics (see Fig. 11). The last rule is simpler here because in the non-preemptive

semantics every context switch is tied with a non-switch step, as explained above. It is impossible for a program to keep switching without executing any code.

$$\frac{W :\overset{\gamma}{\Rightarrow}{}^{+} \mathbf{abort}}{NPEtr(W, \mathbf{abort})} \qquad \frac{W :\overset{\gamma}{\Rightarrow}{}^{*} W' \qquad W' :\xRightarrow{\mathbf{out}\ n} W'' \qquad NPEtr(W'', \mathcal{B})}{NPEtr(W, n :: \mathcal{B})}$$

$$\frac{W :\overset{\gamma}{\Rightarrow}{}^{+} \mathbf{done}}{NPEtr(W, \mathbf{done})} \qquad \frac{W :\overset{\gamma}{\Rightarrow}{}^{+} W' \quad NPEtr(W', \epsilon)}{NPEtr(W, \epsilon)}$$

$$ProgNPEtr((P, \sigma), \mathcal{B}) \text{ iff } \exists \mathtt{t}. NPEtr((P, \mathtt{t}, \sigma), \mathcal{B})$$

Fig. 11. Definition of $ProgNPEtr((P, \sigma), \mathcal{B})$.

4.2 Equivalence with Preemptive Semantics

In this section we prove that, for any DRF program, it behaves the same in the preemptive semantics as in the non-preemptive semantics. Since we define the behavior of a program by the trace of observable events, essentially we require the program to have the same set of event traces in both semantics. The goal is formalized as Theorem 1. As an implicit assumption, the programs we consider must be safe.

Since every step in the non-preemptive semantics can be easily converted to preemptive steps, it is obvious that every event trace in the non-preemptive semantics can be produced in the preemptive semantics.

However, it is non-trivial to prove the other direction: the preemptive semantics cannot generate more event traces than the non-preemptive semantics.

The main idea of the proof is that under data-race-freedom, we can exchange the execution orders of any τ-steps of different threads in preemptive semantics. Note that the orders of atomic steps or print steps cannot be exchanged even when the program is data-race-free.

Then we can fix the order of atomic steps and the print step in the preemptive semantics and reorder all the other steps to form a non-preemptive-like execution. Recall that the non-preemptive semantics allows the threads to switch at the print steps and at the end of atomic blocks. Thus by reordering the steps, we can always let the program start executions in a thread until it reaches a print or the end of an atomic block, and then switch to another thread.

In the following lemmas we describe how to exchange the execution order of threads in preemptive semantics. For convenience, we write W^i to represent a world by setting the current thread id in W to i.

Lemma 1 says the orders of local transitions can be exchanged, as long as the footprints are not conflicting. The reorder would not change the final state, the labels and the generated footprints.

Lemma 1 (Reorder of thread-local transitions).
If $(S_1,(h,s_1)) \xmapsto[\delta_1]{\iota_1} (S_1',(h',s_1')) \wedge (S_2,(h',s_2)) \xmapsto[\delta_2]{\iota_2} (S_2',(h'',s_2')) \wedge \neg(\delta_1 \frown \delta_2)$
then $\exists h'''.\ (S_2,(h,s_2)) \xmapsto[\delta_2]{\iota_2} (S_2',(h''',s_2')) \wedge (S_1,(h''',s_1)) \xmapsto[\delta_1]{\iota_1} (S_1',(h'',s_1'))$

Lemma 2 says if there are two consecutive steps from two threads generating conflicting footprints, then the predicted execution of the two threads starting from the *same* state generating conflicting footprints too. That is, the prediction would not miss the race. We need this lemma because the prediction of different threads starts from the same state in the RACE rule in Fig. 8.

Lemma 2 (Lemma for conflicting thread-local transitions).
If $(S_1,(h,s_1)) \xmapsto[\delta_1]{\iota_1} (S_1',(h',s_1')) \wedge (S_2,(h',s_2)) \xmapsto[\delta_2]{\iota_2} (S_2',(h'',s_2')) \wedge (\delta_1 \frown \delta_2)$
then $\exists \iota_2', \delta_2', S_2'', h''', s_2''.\ (S_2,(h,s_2)) \xmapsto[\delta_2']{\iota_2'} (S_2'',(h''',s_2'')) \wedge (\delta_1 \frown \delta_2')$

Lemma 3 says we can reorder consecutive τ-steps from threads i and j if there is no data race. Lemma 4 shows the reorder of τ steps and atomic steps from different threads. Lemma 5 reorders the internal γ-steps and a print step from different threads. These lemmas are proved by applying Lemma 1 and Lemma 2.

Lemma 3 (Reorder of silent steps).
For any $i, j, W, W_1, W_1', W_2, \delta_1, \delta_2$.
if $W.\sigma.\mathtt{d} = 0 \wedge W^i \overset{\tau}{\underset{\delta_1}{\Rightarrow}}{}^* W_1 \wedge W_1^j \overset{\tau}{\underset{\delta_2}{\Rightarrow}}{}^* W_2 \wedge i \neq j$,
then either $W \Longmapsto \mathtt{Race}$
or $\exists W_3. W^j \overset{\tau}{\underset{\delta_2}{\Rightarrow}}{}^* W_3 \wedge W_3^i \overset{\tau}{\underset{\delta_1}{\Rightarrow}}{}^* W_2^i$.

Lemma 4 (Reorder of silent steps and atomic steps).
For any $i, j, W, W_1, W_2, \delta_1, \delta_2$.
if $W.\sigma.\mathtt{d} = 0 \wedge W_2.\sigma.\mathtt{d} = 0 \wedge W^i \overset{\tau}{\underset{\delta_1}{\Rightarrow}}{}^* W_1 \wedge W_1^j \overset{\mathit{atm}}{\underset{\delta_2}{\Longrightarrow}}{}^* W_2 \wedge i \neq j$,
then either $W \Longmapsto \mathtt{Race}$
or $\exists W_3. W^j \overset{\mathit{atm}}{\underset{\delta_2}{\Longrightarrow}}{}^* W_3 \wedge W_3^i \overset{\tau}{\underset{\delta_1}{\Rightarrow}}{}^* W_2^i$.

Lemma 5 (Reorder of internal steps and a print step).
For any $i, j, W, W_1, W_2, \delta_1, n$.
if $W.\sigma.\mathtt{d} = 0 \wedge W^i \overset{\gamma}{\underset{\delta_1}{\Rightarrow}}{}^* W_1 \wedge W_1^j \overset{\mathit{out}\ n}{\underset{\mathit{emp}}{\Longrightarrow}} W_2 \wedge i \neq j$,
then $\exists W_3. W^j \overset{\mathit{out}\ n}{\underset{\mathit{emp}}{\Longrightarrow}} W_3 \wedge W_3^i \overset{\tau}{\underset{\delta_1}{\Rightarrow}}{}^* W_2^i$.

Then we can prove Theorem 1, saying that preemptive semantics and non-preemptive semantics behave the same.

Theorem 1 (Semantics equivalence).
For any P *and* σ, *if* $\mathsf{DRF}(P,\sigma)$,
then $\forall \mathcal{B}, \mathit{ProgEtr}(P,\sigma,\mathcal{B}) \iff \mathit{ProgNPEtr}(P,\sigma,\mathcal{B})$.

Proof. "$\Longleftarrow$": As explained before, since every step in the non-preemptive semantics can be easily converted to preemptive steps, it is obvious that every event trace in the non-preemptive semantics can be produced in the preemptive semantics.

"$\Longrightarrow$":

We consider the following cases of $\mathcal{B}$. We need to construct a non-preemptive execution for each case:

- case (1): $\mathcal{B} = \textbf{done}$. We prove this case by induction on the number k of atomic blocks.

  0 : There is no atomic blocks. We prove this case by induction on the number of threads. If there is only one thread, we are immediately done. Otherwise, we choose any thread `t` to execute first and delay other threads by exchanging their steps with the thread `t` by Lemma 3. Then after the termination of the thread `t`, we can switch to another thread in the non-preemptive semantics. Then by induction hypothesis we are done.

  k+1 : Let `t` be the first thread that is going to execute an atomic block. Then we exchange the τ-steps from other threads with the steps of thread `t` by Lemma 3 and Lemma 4, so that we first execute thread `t` to the end of its atomic block, and then switch to execute the τ-steps from other threads. In this way we successfully reduce the number of atomic blocks by 1, and then by induction hypothesis, we are done.

- case (2): $\mathcal{B} = \textbf{abort}$. This case is vacant by the assumption of safety.
- case (3): $\mathcal{B} = \epsilon$. If the number of atomic blocks is finite, there must be a point where the last atomic block ends. Then by induction on the number of atomic blocks we can construct a non-preemptive execution to that point by swapping other thread with the thread that is going to execute atomic block, similarly to case (1). Then we know that there is at least one thread `t` keep running forever, otherwise the program will terminate. We can let `t` execute all the time by exchanging other threads with it. Then we successfully construct a diverging execution for non-preemptive semantics.

  Otherwise, there are infinite atomic blocks. The execution is a stream of small execution sections, each consisting of several silent steps in diffrent threads and a single atomic block. We can exchange any silent step with the atomic block by Lemma 4 unless the silent step is in the same thread of the atomic block. Then we can merge the exchanged silent step part into the following section since it contains no atomic block. Therefore the first section consists steps of the same thread, then it can be converted to non-preemptive execution. Then by coinduction we can construct a diverging execution for non-preemptive semantics by converting every section to non-preemptive.
- case (4): $\mathcal{B} = n :: \mathcal{B}'$. We prove this case by coinduction. Then we do induction on the number of atomic blocks and by applying Lemma 3 and Lemma 5, similarly to case (1) above.

$$\frac{W = (P, \mathtt{t}', (h, ls, 0)) \quad P(\mathtt{t}) = S \quad ls(\mathtt{t}) = s \quad (S, (h, s)) \xrightarrow[\delta]{\tau}{}^{*} (S', (h', s'))}{nppredict(W, \mathtt{t}, \delta, 0)} \text{ (NP-PREDICT-0)}$$

$$\frac{W = (P, \mathtt{t}', (h, ls, 0)) \quad P(\mathtt{t}) = S \quad ls(\mathtt{t}) = s \quad (S, (h, s)) \xrightarrow[\delta]{\tau}{}^{*} (S', (h', s')) \quad (S', (h', s')) \xrightarrow[\delta']{\mathbf{atm}}{}^{*} (S'', (h'', s''))}{nppredict(W, \mathtt{t}, \delta \cup \delta', 1)} \text{ (NP-PREDICT-1)}$$

$$\frac{\mathtt{t}_1 \neq \mathtt{t}_2 \quad nppredict(W, \mathtt{t}_1, \delta_1, \mathtt{d}_1) \quad nppredict(W, \mathtt{t}_2, \delta_2, \mathtt{d}_2) \quad \delta_1 \frown \delta_2 \quad (\mathtt{d}_1 = 0 \vee \mathtt{d}_2 = 0)}{W :\Longrightarrow \mathtt{Race}} \text{ (NP-RACE)}$$

$$\frac{(P, \mathtt{t}, \sigma) :\Rightarrow^{*} W' \qquad W' :\xRightarrow{\mathbf{sw}/(\mathbf{out}\ n)} W'' \qquad W'' :\Longrightarrow \mathtt{Race}}{(P, \sigma) :\Longrightarrow \mathtt{Race}} \text{ (SWITCH-RACE)}$$

$$\frac{(P, \mathtt{t}, \sigma) :\Longrightarrow \mathtt{Race}}{(P, \sigma) :\Longrightarrow \mathtt{Race}} \text{ (INIT-RACE)} \qquad \frac{\neg (P, \sigma) :\Longrightarrow \mathtt{Race}}{\mathsf{NPDRF}(P, \sigma)}$$

Fig. 12. Predicting Race in Non-Preemptive Semantics

5 Data-Race-Freedom in Non-Preemptive Semantics

Theorem 1 shows that we can reason about a program in non-preemptive semantics instead of in preemptive semantics, as long as the program satisfies DRF in preemptive semantics. Below we present a notion of data-race-freedom in non-preemptive semantics (NPDRF) which is equivalent to DRF, making it possible to reason about the program solely under non-preemptive semantics.

We define NPDRF in Fig. 12. Similar to the DRF defined in Fig. 8, we predict the footprints of the execution (in the non-preemptive semantics now) of any two threads and see if they are conflicting (see the NP-RACE rule). The INIT-RACE rule and the SWITCH-RACE rule say the prediction can only be made either at the initial program configuration, or at a switch point. This is to ensure the prediction is made only at states from which the thread can indeed be switched to. Otherwise the prediction may not correspond to any actual execution.

The NP-PREDICT-0 rule is similar to the PREDICT-0 rule in Fig 8. The tricky part is in the NP-PREDICT-1 rule. To predict the footprints of program steps inside atomic blocks, we need to execute the preceding silent steps as well (i.e. the τ-steps in the NP-PREDICT-1 rule). This is because the prediction starts only at a switching point, which must be outside of atomic blocks. Therefore we may never reach the atomic block directly without executing the preceding code outside of the atomic block first.

For example, in the program in Fig. 13, both the statements ($\langle [0] := 1 \rangle$) and ($[0] := 1$) writes to address 0. We can predict at the point before ($\langle [0] := 1 \rangle$) to get a data race in the preemptive semantics, following the DRF definition in

$$\begin{array}{l} \textbf{skip}; \\ \langle [0] := 1 \rangle \end{array} \quad \Big\| \quad [0] := 1$$

Fig. 13. Example of data race

Fig. 8. However, in our non-preemptive semantics, it is impossible to predict at the program point right before ($\langle [0] := 1 \rangle$), which is not a switch point. Instead, we have to do the prediction from the beginning of the left thread and execute the preceding **skip** as well.

Note that the prediction will never go across a switch point (e.g. the end of an atomic block). That's why we only consider τ-steps and **atm** steps in the NP-PREDICT-0 rule and the NP-PREDICT-1 rule.

Equivalence between DRF and NPDRF. Below we prove that our novel notion NPDRF under the non-preemptive semantics is equivalent to DRF in the preemptive semantics. First we prove that the two different ways to predict data races in Fig. 8 and Fig. 12 are equivalent, as shown in Lemma. 6 and Lemma. 7.

Lemma 6 (Lemma for prediction and np-prediction).
For any W,$\mathtt{t}$,δ,d, if predict$(W, \mathtt{t}, \delta, d)$, then nppredict$(W, \mathtt{t}, \delta, d)$.

Proof. If $d = 0$ then it is immediate by the rules PREDICT-0 and NP-PREDICT-0.

Otherwise $d = 1$. In NP-PREDICT-1 rule, zero step is acceptable for the first part of silent steps, and the union of a footprint δ and `emp` is δ. Then by unfolding the definition of *predict* and by NP-PREDICT-1 rule we are done.

Lemma 7 (Lemma for data race prediction).
For any k, W and W', if $W \overset{\tau/sw}{\Longrightarrow}^k W' \wedge W' \Longmapsto \mathtt{Race}$, then $W :\Longmapsto \mathtt{Race}$.

Proof. By induction on k.

0 : By unfolding the definition, we know there exist $\mathtt{t}_1$, $\mathtt{t}_2$, δ_1, δ_2, d_1 and d_2 such that $\mathtt{t}_1 \neq \mathtt{t}_2$, *predict*$(W, \mathtt{t}_1, \delta_1, d_1)$, *predict*$(W, \mathtt{t}_2, \delta_2, d_2)$, $\delta_1 \frown \delta_2$ and $d_1 = 0 \vee d_2 = 0$.
Then by applying Lemma 6, we can transform *predict* to *nppredict*.
By NP-RACE rule it is done.

k+1 : We know there exists W_0 and δ_0 such that $W \overset{\tau/\mathbf{sw}}{\underset{\delta_0}{\Longrightarrow}} W_0$ and $W_0 \overset{\tau/\mathbf{sw}}{\Longrightarrow}^k W'$.
Then from the induction hypothesis, we know $W_0 :\Longmapsto \mathtt{Race}$. By unfolding the definition, we know there exist $\mathtt{t}_1$, $\mathtt{t}_2$, δ_1, δ_2, d_1 and d_2 such that $\mathtt{t}_1 \neq \mathtt{t}_2$, *predict*$(W, \mathtt{t}_1, \delta_1, d_1)$, *predict*$(W, \mathtt{t}_2, \delta_2, d_2)$, $\delta_1 \frown \delta_2$ and $d_1 = 0 \vee d_2 = 0$. Suppose the step generating δ_0 is executed by the thread $\mathtt{t}_0$.

- If $(\mathtt{t}_0 = \mathtt{t}_1 \wedge \neg(\delta_0 \frown \delta_2)) \vee (\mathtt{t}_0 = \mathtt{t}_2 \wedge \neg(\delta_0 \frown \delta_1))$, then we can merge it into the prediction by swaping the other thread with this step.
- If $(\mathtt{t}_0 \neq \mathtt{t}_1 \wedge \mathtt{t}_0 \neq \mathtt{t}_2 \wedge \neg(\delta_0 \frown \delta_1) \wedge \neg(\delta_0 \frown \delta_2))$, then we know the step is irrelavent and we can delay the thread $\mathtt{t}_0$ by swaping the two threads with this step.

- Otherwise, $(\mathtt{t}_0 = \mathtt{t}_1 \wedge \delta_0 \frown \delta_2) \vee (\mathtt{t}_0 = \mathtt{t}_2 \wedge \delta_0 \frown \delta_1) \vee (\mathtt{t}_0 \neq \mathtt{t}_1 \wedge \mathtt{t}_0 \neq \mathtt{t}_2 \wedge (\delta_0 \frown \delta_1 \vee \delta_0 \frown \delta_2)$.
  Then we can predict a data race from W.
  In all cases, we can predict a data race from W.

It is very important that a data race should be predicted at switching point. Lemma 7 only concerns the equivalence of prediction. Thus we need to prove the equivalence of data races in preemptive semantics and non-preemptive semantics in Lemma 8.

Lemma 8 (Equivalence of data races in preemptive semantics and non-preemptive semantics).
For any P and σ, we have $((P,\sigma) \Longmapsto \mathtt{Race}) \iff ((P,\sigma) :\Longmapsto \mathtt{Race})$.

Proof. "$\Longleftarrow$": According to the semantics, non-preemptive steps can be converted to preemptive steps directly. Then the problem is reduced to proving that the multi-step prediction of NP-PREDICT-1 in Fig. 12 can be simulated by the prediction in Fig. 8. Informally, we can rearrange the predicting executions defined in Fig. 12 by making the non-conflicting steps sequentially proceeding until the real conflicting steps, and predict in the way as in Fig. 8.

"$\Longrightarrow$": After unfolding the definitions, we prove this case by induction on the number k of event steps.

0 : By induction on the number i of atomic blocks during the execution (except the predicted racing atomic steps).
  0 :By applying Lemma 7 and INIT-RACE rule.
i+1 : Similar to the proof for Theorem 1. If there is no preemptive data race until the end of the first atomic block, then the thread of the first atomic block can execute without being interrupted by other threads, and then the number of atomic blocks can be reduced by 1. Afterwards we apply the induction hypothesis and predict a non-preemptive data race either at the end of the first atomic block (INIT-RACE rule) or a few steps later and right after a switching point (SWITCH-RACE rule). In both cases we are done by SWITCH-RACE rule.
  Otherwise, there is at least one preemptive data race before the end of first atomic block. Since data race cannot be predicted inside the atomic block, the prediction must be made before the first atomic block. Then by Lemma 7 we can predict a non-preemptive data race at the beginning of the execution. Thus we are done by INIT-RACE rule.
k+1 : We know the first thread to event step is thread m. By induction on the number i of atomic blocks before the first event step.
  0 :By applying Lemma 3 and Lemma 5 we can let thread m execute first unless there is a data race before the first event step, which is reduced to case (k=0). Then the number of event steps is reduced by 1 and then by applying induction hypothesis we can predict a non-preemptive data race after the first event step (INIT-RACE rule) or a few steps later and right after a switching point (SWITCH-RACE rule). Then by SWITCH-RACE rule we are done.

i+1 :If there is no preemptive data race until the end of the first atomic block, then the thread of the first atomic block can execute without being interrupted by other threads, and then the number of atomic blocks can be reduced by 1. Afterwards we apply the induction hypothesis and predict a non-preemptive data race either at the end of the first atomic block (INIT-RACE rule) or a few steps later and right after a switching point (SWITCH-RACE rule). In both cases we are done by SWITCH-RACE rule.
Otherwise, there is at least one preemptive data race before the end of first atomic block, which is reduced to case (k=0).

Theorem 2 (Equivalence between DRF and NPDRF).
For any P and σ, we have $\mathsf{DRF}(P, \sigma) \iff \mathsf{NPDRF}(P, \sigma)$

Proof. By applying Lemma 8.

6 Conclusion

In this paper, we propose a formal definition of the non-preemptive semantics, which restricts the interleavings of concurrent threads to certain carefully-chosen program points. We prove that data-race-free programs behave the same in our non-preemptive semantics as in the standard preemptive semantics. Here the behaviors include termination and I/O events. Our results can be used to reduce the complexity of reasoning about data-race-free programs.

We also define a notion of data-race-freedom in non-preemptive semantics (called NPDRF), which is proved to be equivalent to the standard data-race-freedom in preemptive semantics. This makes reasoning solely under our non-preemptive semantics possible.

References

1. Abadi, M., Plotkin, G.: A model of cooperative threads. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09). pp. 29–40. ACM, New York, NY, USA (2009)
2. Adve, S.V., Hill, M.D.: Weak ordering - A new definition. In: Proc. of the 17th Annual International Symposium on Computer Architecture (ISCA'90). pp. 2–14. ACM (1990)
3. Adve, S.V., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. **4**(6), 613–624 (1993)
4. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: Scalable threads for internet services. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03). pp. 268–281. ACM, New York, NY, USA (2003)
5. Beringer, L., Stewart, G., Dockins, R., Appel, A.W.: Verified compilation for shared-memory c. In: Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP'14). pp. 107–127. Springer-Verlag New York, Inc., New York, NY, USA (2014)

6. Boudol, G.: Fair cooperative multithreading. In: Caires, L., Vasconcelos, V.T. (eds.) Proc. 18th International Conference on Concurrency Theory (CONCUR'07). pp. 272–286. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
7. Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-step semantics for analysis and verification of GPU kernels. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22rd European Symposium on Programming Languages and Systems (ESOP'13). pp. 270–289. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Ferreira, R., Feng, X., Shao, Z.: Parameterized memory models and concurrent separation logic. In: Gordon, A.D. (ed.) Proceedings of the 19th European Symposium on Programming Languages and Systems (ESOP'10). pp. 267–286. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
9. Hower, D.R., Hechtman, B.A., Beckmann, B.M., Gaster, B.R., Hill, M.D., Reinhardt, S.K., Wood, D.A.: Heterogeneous-race-free memory models. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS'14). pp. 427–440 (2014)
10. Kojima, K., Igarashi, A.: A Hoare Logic for GPU kernels. ACM Trans. Comput. Logic **18**(1), 3:1–3:43 (Feb 2017)
11. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07). pp. 189–199. ACM, New York, NY, USA (2007)
12. Loring, M.C., Marron, M., Leijen, D.: Semantics of asynchronous javascript. In: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages (DLS'17). pp. 51–62. ACM, New York, NY, USA (2017)
13. Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S.: Drfx: A simple and efficient memory model for concurrent programming languages. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10). pp. 351–362. ACM, New York, NY, USA (2010)
14. Vouillon, J.: Lwt: A cooperative thread library. In: Proceedings of the 2008 ACM SIGPLAN Workshop on ML (ML'08). pp. 3–12. ACM, New York, NY, USA (2008)
15. Yi, J., Disney, T., Freund, S.N., Flanagan, C.: Cooperative types for controlling thread interference in java. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12). pp. 232–242. ACM (2012)
16. Yi, J., Sadowski, C., Flanagan, C.: Cooperative reasoning for preemptive execution. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11). pp. 147–156. ACM, New York, NY, USA (2011)