Memory Access Protocols: Certified Data-Race Freedom for GPU Kernels

Tiago Cogumbreiro[1*], Julien Lange[2], Dennis Liew[1] and Hannah Zicarelli[1]

[1]University of Massachusetts Boston, USA.
[2]Royal Holloway, University of London, UK.

*Corresponding author(s). E-mail(s): tiago.cogumbreiro@umb.edu;
Contributing authors: julien.lange@rhul.ac.uk; zhenrong.liew001@umb.edu; hannah.zicarelli001@umb.edu;

Abstract

GPUs offer parallelism as a commodity, but they are difficult to program correctly. Static analyzers that guarantee data-race freedom (DRF) are essential to help programmers establish the correctness of their programs (kernels). However, existing approaches produce too many false alarms and struggle to handle larger programs. To address these limitations we formalize a novel compositional analysis for DRF, based on memory access protocols. These protocols are behavioral types that codify the way threads interact over shared memory. Our work includes fully mechanized proofs of our theoretical results, the first mechanized proofs in the field of DRF analysis for GPU kernels. Our theory is implemented in Faial, a tool that outperforms the state-of-the-art. Notably, it can correctly verify at least **1.42×** more real-world kernels, and it exhibits a linear growth in 4 out of 5 experiments, while others grow exponentially in all 5 experiments.

Keywords: GPU, data-race, static analysis, behavioural types

1 Introduction

General purpose GPU programming has become more widespread in the last few years as it finds many applications in bio-informatics, artificial intelligence, computer vision, etc [1–4]. While GPUs promise a great return on investment, thanks for their ability to run thousands of threads concurrently, this comes at a cost: they are notoriously difficult to program. In GPU programming, hundreds of lightweight threads share portions of arrays in parallel (without locks) — very different from the programming model of multithreaded programs written in C or Java with heavy-weight heterogeneous threads. Data-race freedom analysis aims to guarantee that for all possible executions, every array cell being written by one thread cannot be concurrently accessed by another thread. Data-race freedom considers the interaction between every pair of threads among a fixed number of threads. Guaranteeing the absence of data-races is especially important when GPU programs are used in critical software (*e.g.*, self-driving cars [5] and medical imaging [6]). Indeed, in such settings, bugs must be found before the code is executed by customers to avoid potentially catastrophic consequences.

In the field of static analysis for detecting data-races in GPU programs, there is a tension between efficiency and correctness (no missed data-races and no false alarms) that thus far is unresolved. Bug finding tools [7–9] favor correctness over efficiency: they provide correct results at small scales, by simulating the program execution. Such tools are incapable of handling certain parameters symbolically (*e.g.*, array size) and can easily exhaust users' resources (*e.g.*, loops with long iteration spaces or unknown bounds). Approaches based on Hoare logic [10–12] can cope with medium-sized programs, do not miss data-races, and do not require array size information; however, they suffer from a high-rate of false alarms and require code annotations written by concurrency experts. Finally, tools that can cope with larger programs and do not require array size information either miss data-races [13] or overwhelm the user with false alarms [14].

We introduce a novel static data-race freedom analysis that can handle larger programs and produce fewer false alarms than related work, without missing data-races. Additionally our analysis does not require code annotations or array size information. Our verification framework hinges on *memory access protocols*, a new family of behavioral types [15] that codify the way threads interact through shared memory. Our behavioral types also make evident two aspects of the analysis that can be made separate: concurrency analysis (*i.e.*, could these two expressions run in parallel?) and data-race conflict detection (*i.e.*, do these array indices match?).

Contributions and synopsis This paper includes the following contributions.

- In Section 3, we formalize the syntax, semantics, and well-formedness conditions for memory access protocols, which are behavioral types for GPU programs. This behavioral abstraction results in a simpler yet

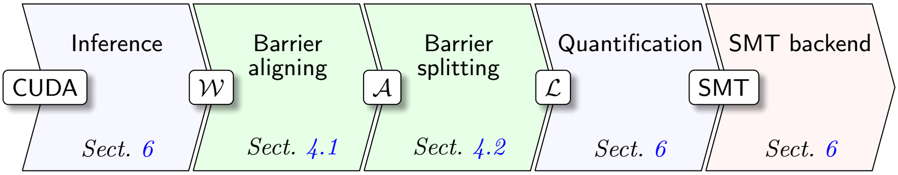


Fig. 1: Work-flow of the verification.

more expressive theory than previous works, *e.g.*, it does not require user-provided loop invariants.

- In Section 4, we show that our data-race freedom analysis of memory access protocols can be soundly and completely reduced to the satisfiability of an SMT formula, see Theorem 1. Our theory and results on memory access protocols are fully mechanized in Coq. To the best of our knowledge, this is the first mechanized proof of correctness of a data-race freedom analysis for GPU programs.
- We show that our data-race freedom analysis of memory access protocols is compositional when protocols satisfy a structural property, see Proposition 1. Additionally, we show how to transform protocols when they do not meet this property.
- In Section 5, we give the detailed proofs of Theorem 1 and Proposition 1. Our discussion includes a novel proof technique that relates (inductively) the syntactic structure of a protocol with the source of concurrent accesses to shared memory.
- In Section 6 we present Faial, which infers memory access protocols from CUDA kernels and implements our theory. Our experiments show that Faial is more precise and scales better than existing tools.
- In Section 7, we present a thorough experimental evaluation of Faial against related work [7, 8, 10, 13], the largest comparative study of GPU verification (5 tools in 260 kernels, 3 tools compared in 487 kernels). Faial verified 218 out of 227 real-world kernels (at least 1.42× more than other tools) and correctly verified more handcrafted tests than other tools (4 out of 5). In a synthetic benchmark suite (250 kernels), Faial is the only tool to exhibit linear growth in 4 out of 5 experiments, while others grow exponentially in all 5 experiments.

Our paper is accompanied by an implementation (Faial), an evaluation framework (including datasets), and proof scripts (in Coq) for each theorem. All of these are available in our artifact [16].

Comparison with our CAV 2021 paper This article revises and extends an earlier version of this work [17] with new material and full proof of results. First, we include a new section with a comprehensive presentation of the results and proofs, following closely our Coq formalism. Second, we revisit the core theory of memory access protocols, first proposed in [17]. We add conditionals and make significant changes to the presentation of the semantics and analysis. The definitions of data-race freedom, well-formed protocols, and protocol splitting, Listing 2.1: Examples of racy kernels, l.h.s. is from [19] and r.h.s. simplifies l.h.s. for clarity, with one-dimensional array and thread identifier, and 1-stride loops.

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<TILE_DIM;
       i+=BLOCK_ROWS)
   { tile [tid.y+i][tid.x] =
         idata[index_in+i*width];}
  __syncthreads();
  for (int j = 0; j<TILE_DIM;
       j+=BLOCK_ROWS)
   { odata[index_out+j*height] =
         tile [tid.x][tid.y+j];}}
```

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<M; i++)
    { tile [tid] = ...;}
  __syncthreads();
  for (int j = 0; j<M; j++)
    {... = tile [tid+j];}}
```

are also revised technically, to match better with the Coq formalism. Third, we have extended and updated the related work with recent publications.

2 Overview

This section gives an overview of our approach by examining a data-race we found in published work [18] and [19]. We discuss the challenges that such examples pose to the state-of-the-art of data-race freedom analysis. Then we introduce a verification framework based on *memory access protocols*: behavioral types [15] that codify the way threads interact via shared memory. Figure 1 gives an overview of the verification pipeline. We start from a CUDA kernel, from which we infer possibly several memory access protocols (one per shared array). Protocols are then transformed in three steps into formulas that are verified by an SMT solver.

2.1 Challenges of GPU Programming

2.1.1 GPU Programming Model

The key component of GPU programming is the *kernel* program, or just kernel, that runs according to the Single-Instruction-Multiple-Thread (SIMT) execution model, where multiple threads run a single instruction concurrently. A kernel is parameterized by a special variable that holds a thread identifier, henceforth named tid. In parallel, each member of a group of threads runs an instantiated copy of the kernel by supplying its identifier as an argument. Threads communicate via shared memory (arrays) and mediate communication via barrier synchronization (an execution point where all threads must wait for each other before advancing further). Writes are only guaranteed to be visible to other threads after a barrier synchronization.

GPU programming platforms usually group threads hierarchically in multiple levels, across which no inter-groups synchronization is possible. In both the literature [13, 20] and this work, the focus is on intra-group communication.

2.1.2 Challenges

We motivate the difficulty of analyzing data-races by studying a programming error found in the wild, reported in Listing 2.1 (left). This excerpt comes from a tutorial [19] on optimizing numeric algorithms for GPUs. The code listing transposes a matrix N-times with an outer loop indexed by variable r.

Remarkably, the tutorial [19] does not inform the readers that Listing 2.1 contains a subtle data-race: one transpose-operation starts (the writes to tile in line 3) without awaiting the termination of the previous transpose-operation (the reads from tile in line 6), thus corrupting the data over time and possibly skewing the timing of the optimization to appear faster than it should be. We found a related data-race in [18], which reuses code from [19].

Our tool, Faial, successfully identifies the program state that triggers the data-race in Listing 2.1: when r=1 and N=2. However, state-of-the-art tools struggle to accurately analyze Listing 2.1, as evaluated in Section 7 (Claim 1: Test 1). Symbolic execution tools, such as [7, 8], timeout for N>1, and, in general, cannot handle symbolic (unknown) bounds. GPUVerify [20], a tool based on Hoare logic, reports a false alarm: a spurious data-race when r=0 and N=1. PUG [13] incorrectly identifies the example as data-race free, as its analysis appears to ignore data-races originating from different iterations of a loop.

2.2 Memory Access Protocols by Example

We now use a memory access protocol to investigate the data-race that stems from the interaction between both loops manipulating array tile, *c.f.*, Listing 2.1. The analysis discussed in this paper assumes that array aliasing has been handled beforehand, thus every array is disjoint from another at this point of the analysis. Such assumption allows us to analyze each array independently, by generating one memory access protocol per array, *e.g.*, our tool yields three protocols from Listing 2.1 (l.h.s). For presentation purposes, we focus our discussion on array tile in Listing 2.1 (r.h.s.), which simplifies the l.h.s. whilst retaining the root cause of its data-race. We discuss how we support multi-dimensional arrays, multi-dimensional thread identifiers, arbitrary loop strides, kernel parameters (*e.g.*, N), and array aliasing in Section 6. In our Coq formalism the notion of "accesses" (and their dimensions) is a parameter of the theory, thus orthogonal to the theory presented here.

Consider the execution of the end of the first iteration (r=0) and the beginning of the second (r=1) iteration of the outer-loop. In this case, the execution of the j-loop when r=0 is not synchronized with the execution of the i-loop when r=1 as there is no call to `__syncthreads()` in between.

The memory access protocol in Listing 2.2 captures this *partial* execution from the viewpoint of array tile. By design, memory access protocols over-approximate kernels by abstracting away *what* data is being written to/read from an array, to focus on *where* data is being written. The protocol models the two problematic loops of Listing 2.1, *i.e.*, the j-loop when r=0 and the i-loop

Listing 2.2: Minimal representative example of a memory access protocol highlighting the data-race in Listing 2.1.

```
// r = 0
for^U j in 0..M    // for (int j = 0; j<M; j++)
  {rd[tid+j]};     //  _ = tile[tid+i];
// r = 1
for^U i in 0..M    // for (int i = 0; i<M; i++)
  {wr[tid]}        //  tile[tid] = _;
```

when r=1. The first loop reads (rd[tid+j]) from the array, while the second writes (wr[tid]) to it. Note that the iteration range includes the lower bound 0 and excludes the upper bound M. The evaluation of a protocol follows the SIMT model: each thread evaluates wr[tid] by instantiating tid with their unique identifier (hereafter, a non-negative integer), *e.g.*, thread 0 yields wr[0] and thread 1 yields wr[1].

2.2.1 Analysis of Unsynchronized Protocols

We say that two concurrent accesses are data-race free (DRF[1]) if both accesses target distinct memory locations, or if both accesses are reading from memory. We say that a protocol is DRF when all of its concurrent accesses are pairwise DRF. For instance the respective sets of concurrent accesses of threads 0 and 1 in Listing 2.2 is given below

$$\begin{array}{ccc} \mathsf{tid} = 0 & & \mathsf{tid} = 1 \\ \{\mathsf{rd}[j] \mid 0 \leq j < M\} \cup \{\mathsf{wr}[0]\} & \textit{DRF with?} & \{\mathsf{rd}[1{+}j] \mid 0 \leq j < M\} \cup \{\mathsf{wr}[1]\} \end{array}$$

When M>1, thread 0 (l.h.s) accesses rd[1] and thread 1 (r.h.s) accesses wr[1]. Thus, there is a data-race on index 1 of the array.

A fundamental challenge of static data-race freedom verification is how to handle loops. Symbolic execution approaches that unroll loops, *e.g.*, [7, 8], cannot handle large or symbolic iteration spaces. Static approaches that use Hoare logic, *e.g.*, [10–12], require user-provided loop invariants. Another approach is to reduce loops to verifying the satisfiability of a corresponding universally quantified formula, *e.g.*, [24, 25]. This has the advantage of being fast and not requiring invariants. However, its previous application to GPU programming, *i.e.*, PUG, is unsound due to the interaction between barrier synchronizations and loops, *e.g.*, PUG misses the data-race in Listing 2.1. We give more details in Section 7.

Our approach A key contribution of our work is to identify conditions that allow a kernel to be reduced to a first-order logic formula, by precisely characterizing the effect of barrier synchronization in loops. To this end, the

[1] The acronym DRF should not be confused with acronyms used in memory models, such as DRF0 [21], DRF1 [22], or DRFX [23].

language of memory access protocols distinguishes syntactically between protocol fragments that synchronize from those that do not. For instance, the protocol in Listing 2.2 is identified as *unsynchronized*, as it does not include any synchronization.

In Section 4, we show that the data-race freedom analysis of unsynchronized protocols can be precisely reduced to a first-order logic formula, where universally quantified formulae represent loops, thus obviating the need to unroll them explicitly. For instance, we reduce the verification of Listing 2.2 to asking whether for all M, t_1, and t_2, where $t_1 \neq t_2$ are thread identifiers, the following holds:

$$\forall j_1, i_1, j_2, i_2 \colon 0 \leq j_1 < M \wedge 0 \leq i_1 < M \wedge 0 \leq j_2 < M \wedge 0 \leq i_2 < M \implies \\ \{\mathsf{rd}[t_1 + j_1]\} \cup \{\mathsf{wr}[t_1]\} \quad \textit{DRF with?} \quad \{\mathsf{rd}[t_2 + j_2]\} \cup \{\mathsf{wr}[t_2]\}$$

This formula is *invalid* since $\mathsf{rd}[t_1 + j_1]$ races with $\mathsf{wr}[t_2]$ when, *e.g.*, $t_1 = 0$, $t_2 = 1$, $j_1 = 1$, and $M > 1$. Hence, our technique flags Listing 2.2 as racy.

2.2.2 Analysis of Synchronized Protocols

The protocol in Listing 2.3 (left) models *all* the interactions over the shared array tile from Listing 2.1. This protocol consists of one outer loop r that contains two inner loops separated by a barrier synchronization (sync). The first inner loop writes ($\mathsf{wr}[\mathsf{tid}]$) to the array, while the second reads ($\mathsf{rd}[\mathsf{tid} + \mathsf{j}]$) from the array.

This protocol illustrates how our language syntactically differentiates between protocol fragments that synchronize from those that do not. Concretely, our language precludes an unsynchronized loop ($\mathsf{for}^{\mathtt{U}}\ x \in n..m\ \{u\}$) from calling sync anywhere in u, and it requires that a synchronized loop ($\mathsf{for}^{\mathtt{S}}\ x \in n..m\ \{p\}$) includes at least one occurrence of sync. The superscript S (resp. U) stands for *s*ynchronized (resp. *u*nsynchronized). This distinction can be inferred automatically and yields a compositional analysis, as we explain below.

In unsynchronized loops, accesses in an iteration always happen concurrently with accesses in any other iteration. In contrast, synchronized loops are more challenging to analyze accurately because accesses in an iteration may, or may not, run concurrently with accesses in adjacent iterations (depending on where barriers are located). For instance an instruction of iteration r in Listing 2.3 (left) may race with an instruction of iteration r+1, when $\mathsf{N} \geq 1$.

Our approach In this work we show that the data-race freedom analysis of synchronized protocols can safely be reduced to a first-order logic formula when such loops are *aligned*, *i.e.*, when there is at least one synchronization exactly before the loop and one at the end of its body. In Section 4.1 we show how to transform an arbitrary memory access protocol into an aligned protocol using a syntax-driven transformation technique called *barrier aligning*. Intuitively, barrier aligning normalizes loops so that they do not “leak” accesses

Listing 2.3: Memory access protocol (left) of array tile from Listing 2.1 and its aligned version (right).

```
1 for^S r in 0..N {
2   for^U i in 0..M { wr[tid] }
3   sync;
4   for^U j in 0..M { rd[tid + j] }
5 }
```

aligns to

```
1 for^U i in 0..M { wr[tid] }
2 sync;
3 for^S r in 1..N {
4   for^U j in 0..M { rd[tid + j] }
5   for^U i in 0..M { wr[tid] }
6   sync;
7 }
8 for^U j in 0..M { rd[tid + j] }
```

between iterations. The right-hand side of Listing 2.3 shows the result of applying *barrier aligning* on the protocol from Listing 2.3 (left). Observe that the fragment before the aligned loop (line 1) corresponds to the unsynchronized part of the original loop (before sync). The original loop itself is rearranged so that the part succeeding sync is moved to the beginning of the aligned loop (lines 3–6). The fragment following the aligned loop (line 7) corresponds to the unsynchronized loop that appears after the sync in the original protocol.

In Section 4.1 we show that aligned protocols enable *compositional* verification: protocol fragments between two barriers can be analyzed independently. This compositional analysis is possible because (*i*) there is no causality between instructions, except through sync and (*ii*) aligned protocols syntactically delimit the causality induced by sync. For instance, the aligned protocol in Listing 2.3 can be reduced to analyzing the following three protocol fragments independently:

(1) $\mathsf{for}^{\mathtt{U}}\ i \in 0..M\ \{\mathsf{wr}[\mathsf{tid}]\}$

(2) $\mathsf{for}^{\mathtt{S}}\ r \in 1..N\ \{\mathsf{for}^{\mathtt{U}}\ j \in 0..M\ \{\mathsf{rd}[\mathsf{tid}+j]\};\ \mathsf{for}^{\mathtt{U}}\ i \in 0..M\ \{\mathsf{wr}[\mathsf{tid}]\};\ \mathsf{sync}\}$

(3) $\mathsf{for}^{\mathtt{U}}\ j \in 0..M\ \{\mathsf{rd}[\mathsf{tid}+j]\}$

Protocols *(1)* and *(3)* are handled like Listing 2.2 because they are unsynchronized. Protocol *(2)* requires more care as it includes a synchronized loop. Representing a synchronized loop as a formula becomes possible when the protocol is *aligned*: both threads must share the same value for r at each iteration. Hence, we reduce the verification to asking whether for all N, M, t_1, and t_2 where $t_1 \neq t_2$ and the following formula holds. We highlight the range of the synchronized loop in green.

$$\forall r, j_1, i_1, j_2, i_2\colon\ 1 \leq r < N \wedge 0 \leq j_1 < M \wedge 0 \leq i_1 < M \wedge 0 \leq j_2 < M \wedge 0 \leq i_2 < M$$
$$\implies \{\mathsf{rd}[t_1 + j_1]\} \cup \{\mathsf{wr}[t_1]\} \quad \textit{DRF with?} \quad \{\mathsf{rd}[t_2 + j_2]\} \cup \{\mathsf{wr}[t_2]\}$$

Syntax of memory access protocols

$$\begin{array}{rcl@{\qquad}rcl}
\mathbb{N} \ni i & ::= & 0 \mid 1 \mid \cdots & o & ::= & \mathsf{wr} \mid \mathsf{rd} \\
n & ::= & x \mid i \mid n \star n & \mathbb{A} \ni \alpha & ::= & i{:}o[i] \\
\mathbb{B} \ni d & ::= & \texttt{true} \mid \texttt{false} & \mathbb{P} \ni P & ::= & \{\alpha_1, \ldots, \alpha_n\} \\
b & ::= & d \mid n \diamond n \mid b \circ b & H & ::= & [] \mid P :: H
\end{array}$$

$$\begin{array}{rcl}
\mathcal{U} \ni u & ::= & \mathsf{skip} \mid o[n] \mid u ;\ u \mid \mathsf{if}\ b\ \{u\}\ \mathsf{else}\ \{u\} \mid \mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\} \\
\mathcal{W} \ni p & ::= & u ;\ \mathsf{sync} \mid p ;\ p \mid u ;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p ;\ u\}
\end{array}$$

Fig. 2: Syntax of memory access protocols.

Our technique identifies Listing 2.3 as racy since this formula is *invalid*, *i.e.*, $\mathsf{rd}[t_1+j_1]$ races with $\mathsf{wr}[t_2]$ when $r = 1$, $t_1 = 0$, $t_2 = 1$, $j_1 = 1$, $N > 1$ and $M > 1$.

3 Memory Access Protocols

A memory access protocol describes the interaction between a group of threads and a single shared-memory array. A protocol records *where* in memory accesses take place, but abstracts away from *what* data is read from/written to memory.

3.1 Syntax

The language of protocols distinguishes between an unsynchronized protocol fragment $u \in \mathcal{U}$, that disallows synchronization, from a synchronized fragment $p \in \mathcal{W}$ that must include a synchronization. The syntax of memory access protocols is given in Figure 2.

Hereafter, i, j, k are meta-variables over non-negative integers (natural numbers) picked from the set $\mathbb{N}$. An numeric expression n is either: a numeric variable x, a natural number i, or a binary operation on natural numbers that yields a natural number. A boolean expression b is either a boolean literal, an arithmetic comparison $\diamond$, or a propositional logic connective $\circ$. Meta-variable α ranges over access values, which hold the information necessary to identify data-races at run-time. In access value $i{:}o[j]$, i is the identifier of the thread performing the access (aka. owner), j is the array location, and o is the mode (read or write).

Memory access protocol An unsynchronized protocol $u \in \mathcal{U}$ either does nothing (skip), accesses a shared memory location $o[i]$ (reads from/writes to index i), performs sequential composition, branches using an if-then-else construct, or loops. Observe that accesses $o[i]$ in (source) protocols do not specify an owner, the association is only done in the semantics (see below). A synchronized protocol $p \in \mathcal{W}$ either performs barrier synchronization sync, runs

Free variables of numeric expressions $\boxed{\mathrm{fv}(n)}$

$$\mathrm{fv}(i) = \emptyset \qquad \mathrm{fv}(x) = \{x\} \qquad \mathrm{fv}(n \star m) = \mathrm{fv}(n) \cup \mathrm{fv}(m)$$

Free variables of boolean expressions $\boxed{\mathrm{fv}(b)}$

$$\mathrm{fv}(d) = \emptyset \qquad \mathrm{fv}(n \diamond m) = \mathrm{fv}(n) \cup \mathrm{fv}(m) \qquad \mathrm{fv}(b \circ b') = \mathrm{fv}(b) \cup \mathrm{fv}(b')$$

Free variables of unsynchronized protocols $\boxed{\mathrm{fv}(u)}$

$$\mathrm{fv}(\mathsf{skip}) = \emptyset \qquad \mathrm{fv}(o[n]) = \mathrm{fv}(n) \qquad \mathrm{fv}(u\,;\ u') = \mathrm{fv}(u) \cup \mathrm{fv}(u')$$

$$\mathrm{fv}(\mathsf{if}\ b\ \{u\}\ \mathsf{else}\ \{u'\}) = \mathrm{fv}(b) \cup \mathrm{fv}(u) \cup \mathrm{fv}(u')$$

$$\mathrm{fv}(\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\}) = \mathrm{fv}(n) \cup \mathrm{fv}(m) \cup (\mathrm{fv}(u) \setminus \{x\})$$

Free variables of synchronized protocols $\boxed{\mathrm{fv}(p)}$

$$\mathrm{fv}(u\,;\ \mathsf{sync}) = \mathrm{fv}(u) \qquad \mathrm{fv}(p\,;\ p') = \mathrm{fv}(p) \cup \mathrm{fv}(p')$$

$$\mathrm{fv}(u\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u'\}) = \mathrm{fv}(u) \cup \mathrm{fv}(n) \cup \mathrm{fv}(m) \cup \big((\mathrm{fv}(p) \cup \mathrm{fv}(u')) \setminus \{x\}\big)$$

Fig. 3: Free variables.

unsynchronized code u, performs sequential composition, or loops. The syntax of synchronized protocols groups each unsynchronized fragment to the closest sync or synchronized loop that appears to their right. We sometimes use parentheses for the sake of readability. For instance, we can write protocol

$$\mathsf{wr}[0]\,;\ \mathsf{rd}[1]\,;\ \mathsf{sync}\,;\ \mathsf{wr}[2]\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{\mathsf{wr}[3]\,;\ \mathsf{sync}\,;\ \mathsf{wr}[4]\}$$

as

$$\big(\mathsf{wr}[0]\,;\ \mathsf{rd}[1]\,;\ \mathsf{sync}\big)\,;\ \big(\mathsf{wr}[2]\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{(\mathsf{wr}[3]\,;\ \mathsf{sync})\,;\ \mathsf{wr}[4]\}\big)$$

3.2 Semantics

Our operational semantics is inspired by the synchronous, delayed semantics (SDV) from Betts *et al.* [20], where threads execute independently and communicate upon reaching a barrier.

We begin with variable binding. The loop constructors are the only binders in our language, and they respectively bind their loop variable in their loop bodies.

Big-step semantics for numeric n and boolean b expressions $\boxed{n \downarrow i}$ $\boxed{b \downarrow d}$

$$\textsc{nat}\ \frac{}{i \downarrow i} \qquad \textsc{n-bin}\ \frac{n \downarrow i \qquad n' \downarrow j \qquad \star(i, j) = k}{n \star n' \downarrow k} \qquad \textsc{bool}\ \frac{}{d \downarrow d}$$

$$\textsc{n-rel}\ \frac{n \downarrow i \qquad n' \downarrow j \qquad \diamond(i, j) = d}{n \diamond n' \downarrow d} \qquad \textsc{b-bin}\ \frac{b \downarrow d \qquad b' \downarrow d' \qquad \circ(d, d') = d''}{b \circ b' \downarrow d''}$$

Fig. 4: Semantics of expressions.

Definition 1 (Binding and Closed protocol)**.** *In* $\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\}$ *the displayed occurrence of* x *is* binding *with scope* u*. In* $u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}$ *the displayed occurrence of* x *is* binding *with scope* $p\,;\ u_2$*.*

Consider the definition of $\mathrm{fv}(\_)$ *(free variables) in Figure 3, we say that a boolean expression* b *is* closed *when* $\mathrm{fv}(b) = \emptyset$*. We say that protocol* $p \in \mathcal{W}$ *is* closed *when* $\mathrm{fv}(p) \subseteq \{\mathsf{tid}\}$*.*

Figure 4 introduces the big-step operational semantics of numeric and boolean expressions. We write $n \downarrow i$ when expression n evaluates to number i. Evaluating $n \star n'$ yields the result of abstract function $\star(i, j)$, when n evaluates to i and m evaluates to j. Similarly, we write $b \downarrow d$ when expression b evaluates to number d.

Observe that numeric and boolean expressions only evaluate when they are *closed* (there is no evaluation rule for variables).

Lemma 1. *For any numeric expression* n*, we have* $n \downarrow i$ *for some* i *if, and only if,* $\mathrm{fv}(n) = \emptyset$*. For any boolean expression* b*, we have* $b \downarrow d$ *for some* d *if, and only if,* $\mathrm{fv}(b) = \emptyset$*.*

Proof. [Source: lemma n_step_iff_closed, file `faial-coq/src/NExp.v`] [Source: lemma b_step_iff_closed, file `faial-coq/src/BExp.v`] The proof for each statement follows by induction on the term (⇒), and on the derivation rules of the assumption (⇐). □

Additionally, the evaluation of expressions is deterministic.

Lemma 2. *If* $n \downarrow i$ *and* $n \downarrow j$*, then* $i = j$*. If* $b \downarrow d$ *and* $b \downarrow d'$*, then* $d = d'$*.*

Proof. [Source: lemma n_step_fun, file `faial-coq/src/NExp.v`] [Source: lemma b_step_fun, file `faial-coq/src/BExp.v`] To prove each statement, we show that there is a Coq function that implements the operational semantics, which can be proved by induction on the expression. □

Big-step semantics for $\mathcal{U}$ $\boxed{u \downarrow_i P}$ $\boxed{u \downarrow P}$

$$\textsc{$\mathcal{U}$-skip}\ \frac{}{\mathsf{skip} \downarrow_i \emptyset} \qquad \textsc{$\mathcal{U}$-acc}\ \frac{n \downarrow j}{o[n] \downarrow_i \{i{:}o[j]\}} \qquad \textsc{$\mathcal{U}$-seq}\ \frac{u_1 \downarrow_i P_1 \qquad u_2 \downarrow_i P_2}{u_1\,;\ u_2 \downarrow_i P_1 \cup P_2}$$

$$\textsc{$\mathcal{U}$-if-t}\ \frac{b \downarrow \mathtt{true} \qquad u_t \downarrow_i P}{\mathsf{if}\ b\ \{u_t\}\ \mathsf{else}\ \{u_f\} \downarrow_i P} \qquad \textsc{$\mathcal{U}$-if-f}\ \frac{b \downarrow \mathtt{false} \qquad u_f \downarrow_i P}{\mathsf{if}\ b\ \{u_t\}\ \mathsf{else}\ \{u_f\} \downarrow_i P} \qquad \textsc{$\mathcal{U}$-for-1}\ \frac{(n \geq m) \downarrow \mathtt{true}}{\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\} \downarrow_i \emptyset}$$

$$\textsc{$\mathcal{U}$-for-2}\ \frac{(n < m) \downarrow \mathtt{true} \qquad u[x := n] \downarrow_i P_1 \qquad \mathsf{for}^{\mathsf{U}}\ x \in n+1..m\ \{u\} \downarrow_i P_2}{\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\} \downarrow_i P_1 \cup P_2}$$

$$\textsc{$\mathcal{U}$-par}\ \frac{P = \bigcup \{P_i \mid i \in \mathbb{T} \land u[\mathsf{tid} := i] \downarrow_i P_i\}}{u \downarrow P}$$

History concatenation and merging $\boxed{H \cdot H}$ $\boxed{H \odot H}$

$$[P_1 \ldots P_n] \cdot [P_{n+1} \ldots P_{n+k}] = [P_1 \ldots P_{n+k}] \qquad (H \cdot [P]) \odot ([P'] \cdot H') = H \cdot [P \cup P'] \cdot H'$$

Big-step semantics for $\mathcal{W}$ $\boxed{p \downarrow H}$

$$\textsc{$\mathcal{W}$-sync}\ \frac{u \downarrow P}{u\,;\ \mathsf{sync} \downarrow [P, \emptyset]} \qquad \textsc{$\mathcal{W}$-seq}\ \frac{p \downarrow H \qquad q \downarrow H'}{p\,;\ q \downarrow H \odot H'}$$

$$\textsc{$\mathcal{W}$-for-1}\ \frac{(n+1 = m) \downarrow \mathtt{true} \qquad u_1 \downarrow P_1 \qquad p[x := n] \downarrow H \qquad u_2[x := n] \downarrow P_2}{u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\} \downarrow [P_1] \odot H \odot [P_2]}$$

$$\textsc{$\mathcal{W}$-for-2}\ \frac{\begin{array}{c}(n+1 < m) \downarrow \mathtt{true} \qquad u_1 \downarrow P_1 \qquad p[x := n] \downarrow H_1 \qquad u_2[x := n] \downarrow P_2 \\ \mathsf{skip}\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n+1..m\ \{p\,;\ u_2\} \downarrow H_2\end{array}}{u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\} \downarrow [P_1] \odot H_1 \odot [P_2] \odot H_2}$$

Fig. 5: Semantics of memory access protocols.

Unsynchronized fragment Figure 5 (top) gives the semantics of unsynchronized protocols, which is parameterized by a set of thread identifiers $\mathbb{T} \subseteq \{0, 1, \ldots, k\}$, where $k \geq 1$. We enforce that there must exist at least two threads, indeed if there is only one thread, there cannot be any data-race. We start with the single-threaded evaluation of an unsynchronized protocol u by a thread identifier i, written $u \downarrow_i P$, yields a *phase*, *i.e.*, a set $P \in \mathbb{P}$ of *access values* $\alpha \in \mathbb{A}$.

Protocol $\textsf{skip}$ produces no accesses. A memory access $o[n]$ evaluates index n down to j and yields a singleton with an access value produced by thread i. Sequencing and branching are standard. Loop ranges include the lower bound and exclude the upper bound, their semantics is otherwise standard.

Rule $\mathcal{U}$-PAR defines the parallel execution of an unsynchronized protocol u, with judgment $u \downarrow P$. The resulting phase is the union of the result of single-threaded execution of u for each thread i, ($u \downarrow_i P_i$ for each i). The rule replaces the unique free-variable $\textsf{tid}$ in u by the thread identifier i; thus, term $u[i := x]$ is closed under evaluation.

Synchronized fragment Figure 5 (bottom) gives the semantics of a protocol, notation $p \downarrow H$. Evaluation of a protocol p yields a *history* (ranged over by H), *i.e.*, a list of phases (P) that records how memory was accessed. A history groups and orders the accesses performed per barrier synchronization. The first element of a history is the set of accesses issued until the first barrier synchronization. The second element of a history holds the set of accesses issued between the first and second barrier synchronizations, and so on. We use $::$ as the list constructor and $\cdot$ for the usual list concatenation operator. We capture the sequencing (or merging) of two histories with the special $\odot$-operator, which merges the last phase of the right-hand side with the first phase of the left-hand side.

Example 1. *For instance, by sequencing a history $[P_1, P_2, P_3]$ with a history $[P'_3, P_4, P_5]$ with the $\odot$-operator, we observe that the third phase consists of the union of P_3 and P'_3.*

$$[P_1, P_2, P_3] \odot [P'_3, P_4, P_5] = [P_1, P_2, P_3 \cup P'_3, P_4, P_5]$$

A barrier synchronization creates two phases (Rule $\mathcal{W}$-SYNC) corresponding to the phases before (P) and after synchronization ($\emptyset$). Sequencing two synchronized protocols p with q merges the last phase of the former with the first phase of the latter, as these two phases run concurrently. Synchronized loops are only defined for nonempty ranges. Variable $\textsf{tid}$ cannot be present in the bounds of the range, as the evaluation of boolean expressions expects closed terms, *c.f.*, Lemma 1. The base case of a synchronized loop corresponds to running its last iteration. When the loop has at least two iterations left, Rule $\mathcal{W}$-FOR-2 sequences the history of the first iteration $[P_1] \odot H_1 \odot [P_2]$ with the history H_2 (that results from running the rest of the loop) by merging the two histories.

Example 2. *Protocols containing empty synchronized loops do not evaluate, e.g., $u_1 \,;\, \textsf{for}^{\textsf{S}}\ x \in 0..0\ \{p\,;\ u_2\} \not\downarrow H$ for any H, p, u_1, and u_2. [Source: lemma* empty_loop*, file* `faial-coq/src/Main.v`*]*

Our semantics is only defined for closed terms. We discuss how we handle kernel parameters and multiple thread identifiers in Section 6.

We formalize data-race freedom using the notion of *safe* phase/history in Definition 2.

Definition 2. *Racy access values, safe phases, and safe histories are defined as follows.*

$$\frac{\mathsf{wr} \in \{o, o'\} \qquad i \neq j}{i{:}o[k] \ \mathit{racy}\ j{:}o'[k]} \qquad \frac{\forall \alpha, \beta \in P \colon \alpha \ \cancel{\mathit{racy}}\ \beta}{\mathit{safe}(P)} \qquad \frac{\forall P \in H \colon \mathit{safe}(P)}{\mathit{safe}(H)}$$

Two accesses are in a data-race (or racy) when there exist two different threads that access the same index k, and at least one of these accesses is a write. Our definition of data-races includes *benign* ones, *e.g.*, a data-race in which both threads write the same value. Phase P is *safe* iff each pair of accesses it contains is not racy. History P is *safe*, *i.e.*, DRF, when all of its phases are safe.

4 DRF-Preserving Transformations of Protocols

This section presents the main steps of the data-race freedom analysis summarized in Figure 1: barrier aligning (Section 4.1) and splitting (Section 4.2). This section also includes the presentation of our key theoretical results. We establish that these steps preserve and reflect data-races (*i.e.*, any and all data-races are found), see Theorem 1. We make precise the notion of compositionality that makes our approach scalable in Proposition 1. We postpone the proofs of these results until Section 5.

4.1 Aligning Protocols

The first transformation step rewrites a protocol to facilitate its analysis. Aligning is a history-preserving operation that produces protocols belonging to language $\mathcal{A}$, see top of Figure 6.

Barrier aligning (or just aligning) is carried out by function *align*, given in the bottom half of Figure 6. The function returns a pair whose first element is an aligned and synchronized protocol, and whose second element is an unsynchronized protocol. Intuitively, a pair is a sequential composition that we distinguish syntactically. The case for synchronization is simple, *align* returns the input protocol as the first component of the pair and skip as the second component (the input protocol is already fully aligned). The case for sequence consists of the sequential composition of the pair aligned with unsynchronized code using operator $(\fatsemi)$. Sequencing two pairs $(a,\, u) \fatsemi (a',\, u')$ amounts to sequencing u to the outer-most piece of unsynchronized code present in a'.

Dealing with synchronized loops is more involved. Given a loop $u_1 \,;\, \mathsf{for}^{\mathsf{S}}\, x \in n..m\ \{p\,;\ u_2\}$, we produce a protocol consisting of the fragment preceding the loop and the synchronized part of its first iteration (a_1), an aligned loop (a_2)

Aligned protocols $\boxed{a \in \mathcal{A}}$

$$\mathcal{A} \ni a \;::=\; u\,;\,\mathsf{sync} \;\mid\; a\,;\,a \;\mid\; a\,;\,\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a\}$$

Sequencing $\boxed{\fatsemi\colon \mathcal{U} \to \mathcal{A} \to \mathcal{A}}$ $\boxed{\fatsemi\colon (\mathcal{A} \times \mathcal{U}) \to (\mathcal{A} \times \mathcal{U}) \to \mathcal{A} \times \mathcal{U}}$

$$u \fatsemi (u'\,;\,\mathsf{sync}) = (u\,;\,u')\,;\,\mathsf{sync} \qquad u \fatsemi (a\,;\,a') = (u \fatsemi a)\,;\,a'$$

$$u \fatsemi (a\,;\,\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\}) = (u \fatsemi a)\,;\,\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\}$$

$$(a,\,u) \fatsemi (a',\,u') = (a\,;\,(u \fatsemi a'),\,u')$$

Aligning protocols $\boxed{\mathit{align}\colon \mathcal{W} \to \mathcal{A} \times \mathcal{U}}$

$$\mathit{align}(u\,;\,\mathsf{sync}) = (u\,;\,\mathsf{sync},\,\mathsf{skip}) \qquad \mathit{align}(p\,;\,q) = \mathit{align}(p) \fatsemi \mathit{align}(q)$$

$$\frac{\begin{array}{c}\mathit{align}(p) = (a,\,u) \qquad a_1 = u_1 \fatsemi a[x := n] \\ a_2 = \mathsf{for}^{\mathsf{S}}\ x \in n{+}1..m\ \{(u\,;\,u_2)[x := x{-}1] \fatsemi a\} \qquad u_3 = (u\,;\,u_2)[x := m{-}1]\end{array}}{\mathit{align}(u_1\,;\,\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\,u_2\}) = (a_1\,;\,a_2,\,u_3)}$$

Fig. 6: Aligning protocols.

starting at $n{+}1$, and the unsynchronized part of its last iteration (u_3). See Listing 2.3 for an example of protocol aligning.

Function *align* can always unroll loops because the analysis only considers synchronized loops with a nonempty range; we discuss how to enforce this assumption in Section 6. Function *align* leads to an exponential blow up wrt. nesting of synchronized loops, but this has not posed problems in practice, *c.f.*, Claim 2.

We now state a fundamental property of barrier aligning: it enables *compositional* verification. To state our compositionality result, we introduce a language of contexts:

$$\begin{array}{rcl}\mathcal{E} & ::= & [\,\_\,] \;\mid\; u\,;\,\mathsf{sync} \;\mid\; a\,;\,\mathcal{E} \;\mid\; \mathcal{E}\,;\,a \\ & \mid & \mathcal{E}\,;\,\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a\} \;\mid\; a\,;\,\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{\mathcal{E}\}\end{array}$$

The base cases correspond to a hole $[\,\_\,]$ or an unsynchronized protocol (followed by sync). The remaining cases follow the structure of memory access protocols.

We define the replacement of the hole with term a in context $\mathcal{E}$ as follows.

$$[\_][a] = a \qquad (u\,;\ \mathsf{sync})[a] = u\,;\ \mathsf{sync}$$

$$(a'\,;\ \mathcal{E})[a] = a'\,;\ (\mathcal{E}[a]) \qquad (\mathcal{E}\,;\ a')[a] = \mathcal{E}[a]\,;\ a'$$

$$(\mathcal{E}\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\})[a] = (\mathcal{E}[a]\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\})$$

$$(a'\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{\mathcal{E}\})[a] = a'\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{\mathcal{E}[a]\}$$

The proposition below relies on the definition of DRF for aligned protocols, which we give in Definition 3 (see Section 5). Intuitively, a is DRF if a produces DRF histories, and $a\!\downarrow$ holds if a can produce some history.

Proposition 1. *Let $\mathcal{E}$ be a context, s.t. $\mathcal{E}[\mathsf{skip}\,;\,\mathsf{sync}]$ is DRF, and $\mathcal{E}[\mathsf{skip}\,;\ \mathsf{sync}]\!\downarrow$. For all $a \in \mathcal{A}$, if a is DRF, $a\!\downarrow$, and $fv(a) \subseteq \{\mathsf{tid}\}$, then $\mathcal{E}[a]$ is also DRF.*

Compositionality allows Faial to analyze each fragment of an aligned protocol independently, by splitting the given protocol into multiple symbolic traces. These traces can be verified independently using an SMT solver.

4.2 Splitting Protocols into Symbolic Traces

The second verification step, *splitting*, consists in transforming an aligned protocol into *symbolic traces*, *i.e.*, symbolic representations of sets of memory accesses which occur between two synchronizations.

Symbolic traces Intuitively, symbolic traces are a thin abstraction over an SMT formula. We describe how to translate a symbolic trace to a formula in Section 6.

We give the syntax and semantics of symbolic traces in Figure 7. We keep the syntax of symbolic traces close to that of protocols. A symbolic trace consists of a sequence of memory accesses and the conditional is its only form of control flow. We make explicit the thread that is issuing an access, as there is no notion of multithreaded execution here. A variable declaration states that a variable can take any value from a range of numbers. Evaluating a symbolic trace then yields one phase per interpretation, which exhaustively instantiates each variable declaration according to its range.

Expression skip represents an empty symbolic trace, so it yields a history with a single empty phase. Expression $n{:}o[m]$ defines a single access. We evaluate expression n, the thread responsible for the access, as i; we evaluate the index n as j, and produce one phase with a single access value, where thread i accesses index j with mode o.

Sequential composition $h_1\,;\ h_2$ uses $\otimes$ to distribute each phase that results from evaluating h_1 over every phase of evaluating h_2, see Example 3.

Syntax

$$o ::= \mathsf{wr} \mid \mathsf{rd} \qquad \mathbb{A} \ni \alpha ::= i{:}o[i] \qquad \mathbb{P} \ni P ::= \{\alpha_1, \ldots, \alpha_n\}$$

$$H ::= [] \mid P :: H$$

$$\mathcal{L} \ni h ::= \mathsf{skip} \mid n{:}o[m] \mid h\,;\ h \mid \mathsf{if}\ b\ \{h\}\ \mathsf{else}\ \{h\} \mid \mathsf{var}\ x\ \mathsf{in}\ n..m;\ h$$

Product of histories $\boxed{H \otimes H}$

$$H_1 \otimes H_2 = [P_1 \cup P_2 \mid P_1 \in H_1\,,\ P_2 \in H_2]$$

Big-step semantics of symbolic traces $\boxed{h \Downarrow H}$

$$\frac{}{\mathsf{skip} \Downarrow [\emptyset]} \qquad \frac{n \downarrow i \qquad m \downarrow j}{n{:}o[m] \Downarrow [\{i{:}o[j]\}]} \qquad \frac{h_1 \Downarrow H_1 \qquad h_2 \Downarrow H_2}{h_1\,;\ h_2 \Downarrow H_1 \otimes H_2}$$

$$\frac{b \downarrow \mathtt{true} \qquad h_1 \Downarrow H}{\mathsf{if}\ b\ \{h_1\}\ \mathsf{else}\ \{h_2\} \Downarrow H} \qquad \frac{b \downarrow \mathtt{false} \qquad h_2 \Downarrow H}{\mathsf{if}\ b\ \{h_1\}\ \mathsf{else}\ \{h_2\} \Downarrow H} \qquad \frac{(n \geq m) \downarrow \mathtt{true}}{\mathsf{var}\ x\ \mathsf{in}\ n..m;\ h \Downarrow [\emptyset]}$$

$$\frac{(n < m) \downarrow \mathtt{true} \qquad h[x := n] \Downarrow H_1 \qquad \mathsf{var}\ x\ \mathsf{in}\ n+1..m;\ h \Downarrow H_2}{\mathsf{var}\ x\ \mathsf{in}\ n..m;\ h \Downarrow H_1 \cdot H_2}$$

Big-step semantics of lists of symbolic traces $\boxed{l \Downarrow H}$

$$[] \Downarrow [] \qquad \frac{h \Downarrow H_1 \qquad l \Downarrow H_2}{h :: l \Downarrow H_1 \cdot H_2}$$

Fig. 7: Syntax and semantics of symbolic traces (shaded parts are recalled from Figure 2).

Example 3. *For instance, if we use $\otimes$ to distribute 2 phases over a history with 3 phases, we get a history with 6 phases:*

$$[P_1, P_2] \otimes [P_3, P_4, P_5] = [P_1 \cup P_3, P_1 \cup P_4, P_1 \cup P_5, P_2 \cup P_3, P_2 \cup P_4, P_2 \cup P_5]$$

Expression $\mathsf{var}\ x\ \mathsf{in}\ n..m;\ h$ binds variable x in h, where variable x is a natural number in the range induced from n and m. A var construct evaluates h for each natural in range $n..m$ to variable x in p, concatenating all phases together. When the range is empty, then var behaves like a skip. Otherwise, when the range has at least one element, we evaluate the assignment of the lower bound n to x in h down to H_1, evaluate the var with the rest of the range $n+1..m$ down to H_2, and yield the phases of both histories $H_1 \cdot H_2$.

Syntax

$$\mathcal{C} \ni c \ ::= \ u \ \mid \ \textsf{var}\ x\ \textsf{in}\ n..m\,;\ c$$

Tracing $\boxed{\mathit{trace}\colon \mathcal{U} \to \mathcal{L}}$

$$\mathit{trace}(\textsf{skip}) = \textsf{skip} \qquad\qquad \mathit{trace}(o[n]) = \textsf{tid}{:}o[n]$$

$$\mathit{trace}(u_1\,;\ u_2) = \mathit{trace}(u_1)\,;\ \mathit{trace}(u_2)$$

$$\mathit{trace}(\textsf{if}\ b\ \{u_1\}\ \textsf{else}\ \{u_2\}) = \textsf{if}\ b\ \{\mathit{trace}(u_1)\}\ \textsf{else}\ \{\mathit{trace}(u_2)\}$$

$$\mathit{trace}(\textsf{for}^{\texttt{U}}\ x \in n..m\ \{u\}) = \textsf{var}\ x\ \textsf{in}\ n..m;\ \mathit{trace}(u)$$

Sequentialization $\boxed{\mathit{seq}\colon \mathcal{C} \to \mathcal{L}}$ $\boxed{\mathit{seq}\colon [\mathcal{C}] \to [\mathcal{L}]}$

$$\frac{t_1, t_2 \text{ fresh} \qquad h_1 = \mathit{trace}(u)[\textsf{tid} \coloneqq t_1] \qquad h_2 = \mathit{trace}(u)[\textsf{tid} \coloneqq t_2]}{\mathit{seq}(u) = \textsf{var}\ t_1\ \textsf{in}\ 1..|\mathbb{T}|;\ \textsf{var}\ t_2\ \textsf{in}\ 0..t_1;\ h_1\,;\ h_2}$$

$$\mathit{seq}(\textsf{var}\ x\ \textsf{in}\ n..m;\ c) = \textsf{var}\ x\ \textsf{in}\ n..m;\ \mathit{seq}(c)$$

$$\mathit{seq}([c_1, \ldots, c_n]) = [\mathit{seq}(c_1), \ldots, \mathit{seq}(c_n)]$$

Splitting $\boxed{\mathit{split}\colon \mathcal{A} \to [\mathcal{C}]}$ $\boxed{\mathit{split}\colon \mathcal{A} \times \mathcal{U} \to [\mathcal{C}]}$

$$\mathit{split}(u\,;\ \textsf{sync}) = [u] \qquad\qquad \mathit{split}(a\,;\ a') = \mathit{split}(a) \cdot \mathit{split}(a')$$

$$\mathit{split}(a\,;\ \textsf{for}^{\texttt{S}}\ x \in n..m\ \{a'\}) = \mathit{split}(a) \cdot [\textsf{var}\ x\ \textsf{in}\ n..m;\ c \mid c \in \mathit{split}(a')]$$

$$\mathit{split}((a,\ u)) = \mathit{split}(a\,;\ u\,;\ \textsf{sync})$$

Fig. 8: Tracing, sequentialization, and splitting of aligned protocols.

Figure 7 (bottom) gives the semantics of lists of symbolic of traces, which is useful to state our main result (see Theorem 1). A list of traces simply evaluates to a list of histories produced by the element of the list.

Barrier splitting is the transformation from aligned protocols to symbolic traces, performed via functions *split*, *trace*, and *seq*, defined in Figure 8.

For convenience, we first introduce an intermediary language $\mathcal{C}$ (ranged over by c) called the language of *protocol closures*, which combines an unsynchronized protocol with the synchronizing loop variables that enclose that protocol. A term in $\mathcal{C}$ is either an unsynchronized protocol fragment or a variable binder.

Function *trace* translates an unsynchronized protocol into a symbolic trace. Memory accesses are tagged with i to signify that a single thread is issuing this access. Later function *seq* uses *trace* and replaces tid by either t_1 or t_2 in the output of *trace*. Tracing sequences and conditionals is straightforward, since sequence and conditionals also exist in $\mathcal{L}$. The key element of function *trace* is that it reinterprets the loop as a variable binder.

Function *seq* translates a protocol closure into a symbolic trace. The base case u turns an unsynchronized protocol into sequencing the execution of two symbolic threads, h_1 and h_2, respectively. Since variable tid is thread-local, *seq* declares fresh variables t_1 to represent the identifier (tid) of the first symbolic thread, and t_2 to represent the identifier of the second thread symbolic thread. Recall that the semantics of $\mathcal{L}$ yields one phase per distinct valuation of variables t_1 and t_2, guaranteeing that all thread-pairs are verified. The ranges assigned to t_1 and t_2 guarantee that both values are distinct, as each thread identifier is unique. Our theory assumes that there are at least two threads, as otherwise the kernel is trivially DRF. No other assumptions are made about the thread count $|\mathbb{T}|$. Our tool can represent the number of threads $|\mathbb{T}|$ symbolically, thus proving data-race freedom for any number of threads.

Function *split* takes an aligned protocol and produces a list of protocol closures. The base case is for $u\,;\,$sync, which simply returns a singleton list containing u. The case for $a\,;\,a'$ simply concatenates the result of splitting a and a'. The case for synchronized loops reinterprets the loop as a variable binder for each closure found in the body of the loop. For convenience, we also define a version of *split* that takes a pair of aligned protocol and unsynchronized protocol, *i.e.*, the return values of function *align* (see Figure 6).

Example 4. *Let* $\hat{a} = \mathsf{wr}[\mathsf{tid}+1];\ \mathsf{rd}[\mathsf{tid}+2];\ \mathsf{sync}$*. We have that* $seq(split(\hat{a}))$ *returns:*

$$[\mathsf{var}\ t_1\ \mathsf{in}\ 1..|\mathbb{T}|;\ \mathsf{var}\ t_2\ \mathsf{in}\ 0..t_1;\ t_1{:}\mathsf{wr}[t_1{+}1];\ t_1{:}\mathsf{rd}[t_1{+}2];\ t_2{:}\mathsf{wr}[t_2{+}1];\ t_2{:}\mathsf{rd}[t_2{+}2]]$$

Finally, we establish that aligning and splitting preserve and reflect data-races, *i.e.*, any and all data-races are found. Thus, the only source of approximation in our analysis stems from the inference of protocols from CUDA kernels, which we discuss in Section 6.

Theorem 1. *If* $p \downarrow H_1$ *and* $seq(split(align(p))) \Downarrow H_2$*, then* $safe(H_1)$ *if and only if* $safe(H_2)$*.*

5 Proofs of Theorem 1 and Proposition 1

In this section we explain in details the proof of Theorem 1 and Proposition 1. Here we aim to give a high-level, yet precise, view of the Coq formalization of

$$\begin{tikzcd}
p \arrow[r] \arrow[d, dotted] & \mathit{align}(p) = (a,\, u) \arrow[r] \arrow[d, dotted] & \mathit{split}((a,\, u)) = c \arrow[r] \arrow[d, dotted] & \mathit{seq}(c) = h \arrow[d, dotted] \\
\phi \subseteq^{\mathtt{W}} p \arrow[r, bend right, "\text{Lemma 10}"'] & \phi \subseteq^{\mathtt{A}} (a,\, u) \arrow[l, bend right, "\text{Lemma 6}"'] \arrow[r, bend right, "\text{Lemma 11}"'] & \phi \subseteq^{\mathtt{C}} c \arrow[l, bend right, "\text{Lemma 12}"'] \arrow[r, bend right, "\text{Lemma 18}"'] & \phi \subseteq^{\mathtt{H}} h \arrow[l, bend right, "\text{Lemma 17}"']
\end{tikzcd}$$

Fig. 9: Overview of the key lemmas used to establish Theorem 1.

our results. For each step in the proof, we give the key details and hide the more tedious parts. The interested reader may refer to the Coq files directly [16].

The crux of our proof strategy is to establish a family of *pair-inclusion judgments* that, for each intermediate language in our analysis, bridge the notion of DRF with their respective semantics. The key is to observe that the existence of a race is defined in terms of *pairs* of access values that appear in a single phase. Hence, to show that our transformations preserve and reflect DRF, it suffices to show that each pair of accesses within individual phases is maintained by each transformation. Figure 9 gives an overview of the key judgments that we use in the proof of Theorem 1, where ϕ ranges over pairs of access values (α_1, α_2).

The simplest pair-inclusion judgment is the one defined for unsynchronized protocols, *i.e.*, $(\alpha_1, \alpha_2) \subseteq^{\mathtt{U}} u$ (not shown in Figure 9). This judgment says that access values α_1 and α_2 both belong to the (single) phase produced by protocol u. All other pair-inclusion judgments build on the one for unsynchronized protocols. For instance, $(\alpha_1, \alpha_2) \subseteq^{\mathtt{W}} p$ holds if p includes an unsynchronized fragment which contains both α_1 and α_2 (see Figure 11). Note that pair-inclusion judgments are tightly related to the semantics of protocols and are *not* meant to be computed statically, *e.g.*, they rely on evaluating loop bounds, instantiating loop indices, etc.

The rest of this section follows the structure of our transformation pipeline, *c.f.*, Figure 1. In Section 5.1, we describe the proof of correctness for function *align*. In Section 5.2, we describe the proof of correctness for function *split*. In Section 5.3, we describe the proof of correctness for function *seq*. Section 5.4 shows how these intermediate results fit together to produce our main theorem. Finally, Section 5.5 gives an overview of the proof of Proposition 1.

5.1 Correctness of *align*

Our proof technique abstracts the operational semantics of Figure 5 in terms of concurrent accesses contained in single phases. Recall that the semantics of a protocol yields a history, *i.e.*, a list of sets of accesses, one set for each phase (between synchronizations). Since our goal is to show DRF preservation, it

Notation

$$\phi \ ::= \ (\alpha, \alpha) \qquad\qquad \mathit{owner}(i{:}o[j]) = i$$

Phase membership / pair-inclusion (unsynchronized) $\boxed{\alpha \in^{\mathtt{U}} u}$ $\boxed{\phi \subseteq^{\mathtt{U}} u}$

$$\frac{\alpha \in \{i{:}o[j] \mid i \in \mathbb{T} \land n[\mathsf{tid} := i]\downarrow j\}}{\alpha \in^{\mathtt{U}} o[n]} \qquad \frac{\alpha \in^{\mathtt{U}} u_1}{\alpha \in^{\mathtt{U}} u_1 \,;\, u_2} \qquad \frac{\alpha \in^{\mathtt{U}} u_2}{\alpha \in^{\mathtt{U}} u_1 \,;\, u_2}$$

$$\frac{b[\mathsf{tid} := \mathit{owner}(\alpha)]\downarrow \texttt{true} \quad \alpha \in^{\mathtt{U}} u_1}{\alpha \in^{\mathtt{U}} \mathsf{if}\ b\ \{u_1\}\ \mathsf{else}\ \{u_2\}} \qquad \frac{b[\mathsf{tid} := \mathit{owner}(\alpha)]\downarrow \texttt{false} \quad \alpha \in^{\mathtt{U}} u_2}{\alpha \in^{\mathtt{U}} \mathsf{if}\ b\ \{u_1\}\ \mathsf{else}\ \{u_2\}}$$

$$\frac{\begin{array}{cc} n[\mathsf{tid} := \mathit{owner}(\alpha)]\downarrow i & m[\mathsf{tid} := \mathit{owner}(\alpha)]\downarrow k \\ \exists j : i \leq j < k & \alpha \in^{\mathtt{U}} u[x := j] \end{array}}{\alpha \in^{\mathtt{U}} \mathsf{for}^{\mathtt{U}}\ x \in n..m\ \{u\}} \qquad \frac{\alpha_1 \in^{\mathtt{U}} u \quad \alpha_2 \in^{\mathtt{U}} u}{(\alpha_1, \alpha_2) \subseteq^{\mathtt{U}} u}$$

First-phase membership (well-formed): $\boxed{\alpha \in^{\mathtt{W}} \mathit{fst}(p)}$

$$\frac{\alpha \in^{\mathtt{U}} u}{\alpha \in^{\mathtt{W}} \mathit{fst}(u \,;\, \mathsf{sync})} \qquad \frac{\alpha \in^{\mathtt{W}} \mathit{fst}(p)}{\alpha \in^{\mathtt{W}} \mathit{fst}(p \,;\ q)} \qquad \frac{\alpha \in^{\mathtt{U}} u_1}{\alpha \in^{\mathtt{W}} \mathit{fst}(u_1 \,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p \,;\ u_2\})}$$

$$\frac{\alpha \in^{\mathtt{W}} \mathit{fst}(p[x := n])}{\alpha \in^{\mathtt{W}} \mathit{fst}(u_1 \,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p \,;\ u_2\})}$$

First-phase membership (aligned) $\boxed{\alpha \in^{\mathtt{A}} \mathit{fst}(a)}$

$$\frac{\alpha \in^{\mathtt{U}} u}{\alpha \in^{\mathtt{A}} \mathit{fst}(u \,;\, \mathsf{sync})} \qquad \frac{\alpha \in^{\mathtt{A}} \mathit{fst}(a)}{\alpha \in^{\mathtt{A}} \mathit{fst}(a \,;\ a')} \qquad \frac{\alpha \in^{\mathtt{A}} \mathit{fst}(a)}{\alpha \in^{\mathtt{A}} \mathit{fst}(a \,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\})}$$

Last-phase membership (well-formed) $\boxed{\alpha \in^{\mathtt{W}} \mathit{lst}(p)}$

$$\frac{\alpha \in^{\mathtt{W}} \mathit{lst}(q)}{\alpha \in^{\mathtt{W}} \mathit{lst}(p \,;\ q)} \qquad \frac{\alpha \in^{\mathtt{W}} \mathit{lst}(p[x := m - 1])}{\alpha \in^{\mathtt{W}} \mathit{lst}(u_1 \,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p \,;\ u_2\})}$$

$$\frac{\alpha \in^{\mathtt{W}} \mathit{lst}(u_2[x := m - 1])}{\alpha \in^{\mathtt{W}} \mathit{lst}(u_1 \,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p \,;\ u_2\})}$$

Fig. 10: Phase membership for memory access protocols.

suffices to show a stronger result: any pair of concurrent accesses originating from a well-formed protocol is present in the aligned protocol, and vice versa.

Discussion In our experience the challenge of proving this result centered around discovering the pair-inclusion judgments and defining them correctly. The axiomatization of these judgments made the Coq proofs simpler, because it allowed us to separate concerns and prove them independently: (*Concern 1*) abstract pair-inclusion from the semantics, (*Concern 2*) show that aligning preserves pair-inclusion. A downside of our approach is that it required us to restate a substitution lemma for each judgment. We do not present these substitutions lemmas here. While their proofs are not the easiest, they are mostly repetitive.

Phase membership, first and last phases. Our running example (Section 2.2) highlights the main challenge of our analysis: when sequencing two protocols, the only way for them to affect each other is when an access from the last phase of the former protocol conflicts with an access from the first phase of the latter protocol.

Figure 10 formalizes the *phase membership* judgment for unsynchronized protocols, then defines first and last phase membership judgments. Phase membership judgments are the building blocks of pair-inclusion judgment, as they evaluate whether a single access belong to a protocol. We use ϕ to range over pairs of accesses (or access values). Intuitively $\phi \subseteq^{\mathtt{U}} u$ holds when both elements of ϕ occur in u, *i.e.*, $\alpha \in^{\mathtt{U}} u$ for each α in pair ϕ. Observe that judgment $\alpha \in^{\mathtt{U}} u$ instantiates all free variables in u by replacing tid by the owner (thread identifier) of α and by picking suitable loop indices.

Judgments $\alpha \in^{\mathtt{W}} \mathit{fst}(p)$ and $\alpha \in^{\mathtt{A}} \mathit{fst}(a)$ respectively formalize when an access α belong to the first phase of a protocol and an aligned protocol. Observe how the rules for a are simpler. Indeed aligned protocols always synchronize at the end of loops, thus phases are easier to identify.

Judgment $\alpha \in^{\mathtt{W}} \mathit{lst}(p)$ holds when α belongs to the last phase of p. Note that this judgment only holds when p includes a loop, *i.e.*, there is no case when $p = u\,;\,$ sync since the last (second) phase of p is empty.

Pair-inclusion for well-formed protocols Figure 11 defines $\phi \subseteq^{\mathtt{W}} p$, *i.e.*, the pair-inclusion judgment for well-formed protocols. The base case of this judgment boils down to the one for unsynchronized protocols (see Rule $\mathcal{W}$-ACC). The other cases essentially enumerate all possible phases that p may produce, then checks whether it includes the pair ϕ. Observe how Rule $\mathcal{W}$-SEQ-B makes use of the first-phase and last-phase membership judgments. Indeed as $p\,;\,q$ is not aligned, p may not terminate with a synchronization. Similar situations may arise in loops, which explains the number of rules dealing with loops.

Pair-inclusion for aligned protocols Figure 12 defines $\phi \subseteq^{\mathtt{A}} a$, *i.e.*, the pair-inclusion judgment for aligned protocols. This judgment follows similar lines to judgment $\phi \subseteq^{\mathtt{W}} p$ but it requires many fewer rules. For instance, in $a\,;\,a'$ it must be the case that a ends with a sync, thus a and a' produce non-overlapping phases. Therefore, the pair ϕ must occur in either a or a'.

Pair-inclusion (well-formed) $\boxed{\phi \subseteq^{\mathtt{W}} p}$

$$\textsc{$\mathcal{W}$-acc}\ \dfrac{\phi \subseteq^{\mathtt{U}} u}{\phi \subseteq^{\mathtt{W}} u\,;\ \mathsf{sync}} \qquad \textsc{$\mathcal{W}$-seq-l}\ \dfrac{\phi \subseteq^{\mathtt{W}} p}{\phi \subseteq^{\mathtt{W}} p\,;\ q} \qquad \textsc{$\mathcal{W}$-seq-r}\ \dfrac{\phi \subseteq^{\mathtt{W}} q}{\phi \subseteq^{\mathtt{W}} p\,;\ q} \qquad \textsc{$\mathcal{W}$-seq-b}\ \dfrac{\alpha_1 \in^{\mathtt{W}} \mathit{lst}(p) \qquad \alpha_2 \in^{\mathtt{W}} \mathit{fst}(q)}{(\alpha_1, \alpha_2) \subseteq^{\mathtt{W}} p\,;\ q}$$

$$\textsc{$\subseteq^{\mathtt{W}}$-for-1}\ \dfrac{\phi \subseteq^{\mathtt{U}} u_1}{\phi \subseteq^{\mathtt{W}} u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}} \qquad \textsc{$\subseteq^{\mathtt{W}}$-for-2}\ \dfrac{\alpha_1 \in^{\mathtt{U}} u_1 \qquad \alpha_2 \in^{\mathtt{W}} \mathit{fst}(p[x := n])}{(\alpha_1, \alpha_2) \subseteq^{\mathtt{W}} u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}}$$

$$\textsc{$\subseteq^{\mathtt{W}}$-for-3}\ \dfrac{n \downarrow j \qquad m \downarrow k \qquad \exists i \colon j \leq i < k \qquad \phi \subseteq^{\mathtt{W}} p[x := i]}{\phi \subseteq^{\mathtt{W}} u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}}$$

$$\textsc{$\subseteq^{\mathtt{W}}$-for-4}\ \dfrac{n \downarrow j \qquad m \downarrow k \qquad \exists i \colon j \leq i < k \qquad \phi \subseteq^{\mathtt{W}} u_2[x := i]}{\phi \subseteq^{\mathtt{W}} u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}}$$

$$\textsc{$\subseteq^{\mathtt{W}}$-for-5}\ \dfrac{n \downarrow j \qquad m \downarrow k \qquad \exists i \colon j \leq i < k \qquad \alpha_1 \in^{\mathtt{W}} \mathit{lst}(p[x := i]) \qquad \alpha_2 \in^{\mathtt{U}} u_2[x := i]}{(\alpha_1, \alpha_2) \subseteq^{\mathtt{W}} u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}}$$

$$\textsc{$\subseteq^{\mathtt{W}}$-for-6}\ \dfrac{n \downarrow j \quad m \downarrow k \quad \exists i \colon j \leq i < k \quad \alpha_1 \in^{\mathtt{U}} u_2[x := i] \quad \alpha_2 \in^{\mathtt{W}} \mathit{fst}(q[x := i + 1])}{(\alpha_1, \alpha_2) \subseteq^{\mathtt{W}} u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}}$$

$$\textsc{$\subseteq^{\mathtt{W}}$-for-7}\ \dfrac{n \downarrow j \quad m \downarrow k \quad \exists i \colon j \leq i < k \quad \alpha_1 \in^{\mathtt{W}} \mathit{lst}(q[x := i]) \quad \alpha_2 \in^{\mathtt{W}} \mathit{fst}(q[x := i + 1])}{(\alpha_1, \alpha_2) \subseteq^{\mathtt{W}} u_1\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;\ u_2\}}$$

Fig. 11: Pair-inclusion (well-formed).

Results Now that we have formalized all the necessary judgments for this section, we can state their properties formally. Our first lemma states that function *align* preserves and reflects the set of accesses of the last phase.

Lemma 3. *If $p \downarrow H$ for some H, and $\mathit{align}(p) = (a, u)$, then for any access α, $\alpha \in^{\mathtt{U}} u$ if, and only if, $\alpha \in^{\mathtt{W}} \mathit{lst}(p)$.*

Proof. [Source: lemma i_last_align, file `faial-coq/src/Align.v`] We prove each direction of the equivalence independently. In each case the proof follows by induction on the derivation of $p \downarrow H$. □

Pair-inclusion (aligned) $\boxed{\phi \subseteq^{\mathtt{A}} a}$ $\boxed{\phi \subseteq^{\mathtt{A}} (a,\, u)}$

$$\subseteq^{\mathtt{A}}\text{-}\textsc{acc}\ \frac{\phi \subseteq^{\mathtt{U}} u}{\phi \subseteq^{\mathtt{A}} u\,;\ \mathsf{sync}} \qquad \subseteq^{\mathtt{A}}\text{-}\textsc{seq-l}\ \frac{\phi \subseteq^{\mathtt{A}} a}{\phi \subseteq^{\mathtt{A}} a\,;\ a'} \qquad \subseteq^{\mathtt{A}}\text{-}\textsc{seq-r}\ \frac{\phi \subseteq^{\mathtt{A}} a'}{\phi \subseteq^{\mathtt{A}} a\,;\ a'} \qquad \subseteq^{\mathtt{A}}\text{-}\textsc{for-1}\ \frac{\phi \subseteq^{\mathtt{A}} a}{\phi \subseteq^{\mathtt{A}} a\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\}}$$

$$\subseteq^{\mathtt{A}}\text{-}\textsc{for-2}\ \frac{\phi \subseteq^{\mathtt{A}} a'[x := i] \qquad (n \le i < m)\downarrow \mathtt{true}}{\phi \subseteq^{\mathtt{A}} a\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\}} \qquad \subseteq^{\mathtt{A}}\text{-}\textsc{def}\ \frac{\phi \subseteq^{\mathtt{A}} a\,;\ u\,;\ \mathsf{sync}}{\phi \subseteq^{\mathtt{A}} (a,\, u)}$$

Fig. 12: Pair-inclusion (aligned).

The following lemma establishes that function *align* preserves and reflects the set of accesses of the first phase.

Lemma 4. *If $p \downarrow H$ for some H, and $align(p) = (a, u)$, then for any access α, $\alpha \in^{\mathtt{W}} fst(p)$ if, and only if $\alpha \in^{\mathtt{A}} fst(a)$.*

Proof. [Source: lemma `i_first_align`, file `faial-coq/src/Align.v`] We prove each direction of the equivalence independently. In each case the proof follows by induction on the derivation of $p \downarrow H$. □

We establish the source of an access over function $\_ \fatsemi \_$.

Lemma 5. *Let $\phi = (\alpha_1, \alpha_2)$. If $\phi \subseteq^{\mathtt{A}} u \fatsemi a$, then either $\phi \subseteq^{\mathtt{U}} u$, or $\phi \subseteq^{\mathtt{A}} a$, or $\alpha_1 \in^{\mathtt{U}} u$ and $\alpha_2 \in^{\mathtt{A}} fst(a)$, or $\alpha_2 \in^{\mathtt{U}} u$ and $\alpha_1 \in^{\mathtt{A}} fst(a)$.*

Proof. [Source: lemma `i_pair_in_inv_n_seq`, file `faial-coq/src/ALang.v`] The proof follows by induction on the derivation of $\subseteq^{\mathtt{A}}$. □

Lemma 6. *If $p \downarrow H$ for some H, then for any pair ϕ, if $\phi \subseteq^{\mathtt{A}} align(p)$, then $\phi \subseteq^{\mathtt{W}} p$.*

Proof. [Source: lemma `in_1`, file `faial-coq/src/Align.v`] The proof develops by induction on the derivation of $p \downarrow H$, at each case we invert hypothesis (H1) $\phi \subseteq^{\mathtt{A}} align(p)$ so that we can range over all the origins of a pair of accesses ϕ and get the necessary conditions to apply each constructor of $\subseteq^{\mathtt{W}}$. We only show the proof of Rule $\mathcal{W}$-SEQ, as other cases are similar. We note that the proof for the loop construct is quite laborious, yet unsurprising; the inversion of (H1) in our Coq proof yields 18 different cases.

Our goal is to show $\phi \subseteq^{\mathtt{W}} p\,;\ q$. Our assumptions are $align(p\,;\ q) = \big(a_1\,; (u_1 \fatsemi a_2), u_2\big)$, $align(p) = (a_1, u_1)$, $align(q) = (a_2, u_2)$, (H1) $\phi \subseteq^{\mathtt{A}} \big(a_1\,; (u_1 \fatsemi a_2), u_2\big)$. Our two induction hypotheses are: (IH.1) if $\phi \subseteq^{\mathtt{A}} a_1$ or $\phi \subseteq^{\mathtt{U}} u_1$, then $\phi \subseteq^{\mathtt{W}} p$; and, (IH.2) if $\phi \subseteq^{\mathtt{A}} a_2$ or $\phi \subseteq^{\mathtt{U}} u_2$, then $\phi \subseteq^{\mathtt{W}} q$.

Performing case analysis on (H1) and using Lemma 5 yields three cases:

- **Case 1:** *When accesses ϕ are in $align(p)$, we show they are also in p.* That is, we have that (H1) $\phi \subseteq^{\mathtt{A}} a_1$, or that (H1) $\phi \subseteq^{\mathtt{U}} u_1$. Obtain (2) $\phi \subseteq^{\mathtt{W}} p$ from (H1) and (IH.1). Conclude applying $\mathcal{W}$-SEQ-L and (2).
- **Case 2:** *When access ϕ are in $align(q)$, we show they are also in q.* That is, we have that (H1) $\phi \subseteq^{\mathtt{A}} a_2$, or that (H1) $\phi \subseteq^{\mathtt{U}} u_2$. Obtain (2) $\phi \subseteq^{\mathtt{W}} q$ from (H1) and (IH.2). Conclude applying $\mathcal{W}$-SEQ-R and (2).
- **Case 3:** *When α_1 is in the last phase of $align(p)$, α_2 is in the last phase of $align(q)$, and $\phi = (\alpha_1, \alpha_2)$, then we conclude that α_1 is in the last phase of p and that α_2 is in the first phase of q.* That is, we have (H1) $\alpha_1 \in^{\mathtt{U}} u_1$ and (H2) $\alpha_2 \in^{\mathtt{A}} \mathit{fst}(a_2)$. We get (2) $\alpha_1 \in^{\mathtt{W}} \mathit{lst}(p)$ from (H1) and Lemma 3. We get (3) $\alpha_2 \in^{\mathtt{W}} \mathit{fst}(q)$ from Lemma 4 and (H2). Conclude with $\mathcal{W}$-SEQ-B and (2) and (3).

□

Before we show the reverse direction, we establish three auxiliary lemmas to sequence an unsynchronized protocol with a synchronized one, which together establish the reverse of Lemma 5. First, we show that pair-inclusion is preserved on the right-hand side of sequence.

Lemma 7. *If $\phi \subseteq^{\mathtt{U}} u$, then $\phi \subseteq^{\mathtt{A}} u \mathbin{;} a$ for any a.*

Proof. [Source: lemma i_pair_in_n_seq_l, file `faial-coq/src/Align.v`] The proof follows by induction on the structure of a; on each case we use the constructors of $\subseteq^{\mathtt{A}}$. □

Second, we show that pair-inclusion is preserved on the left-hand side of sequence.

Lemma 8. *If $\phi \subseteq^{\mathtt{A}} a$, then $\phi \subseteq^{\mathtt{A}} u \mathbin{;} a$ for any u.*

Proof. [Source: lemma i_pair_in_n_seq_r, file `faial-coq/src/Align.v`] The proof follows by induction on the derivation of $\phi \subseteq^{\mathtt{A}} a$; on each case we use the constructors of $\subseteq^{\mathtt{A}}$. □

Third, we show that if an access is in the left-hand side and an other access is in the right-hand side, then the pair of accesses is in the sequence.

Lemma 9. *If $\alpha_1 \in^{\mathtt{U}} u$ and $\alpha_2 \in^{\mathtt{A}} \mathit{fst}(a)$, then $(\alpha_1, \alpha_2) \subseteq^{\mathtt{A}} u \mathbin{;} a$.*

[Source: lemma i_pair_in_n_seq_2, file `faial-coq/src/Align.v`]

We are now ready to show the reverse direction.

Lemma 10. *If $p \downarrow H$ and $\phi \subseteq^{\mathtt{W}} p$, then $\phi \subseteq^{\mathtt{A}} align(p)$.*

Proof. [Source: lemma in_2, file `faial-coq/src/Align.v`] The proof follows by induction on the derivation of $\phi \subseteq^{\mathtt{W}} p$. For each case we apply the constructors of $\subseteq^{\mathtt{A}}$. We show the cases related to sequencing. The remaining cases are similar. The rules for loops must additionally distinguish between the first, last, and middle (non-first and non-last) iterations. In the following three cases we have: $\phi \subseteq^{\mathtt{W}} p$, $\mathit{align}(p) = (a_1, u_1)$, and $\mathit{align}(q) = (a_2, u_2)$. Our goal is to prove $\phi \subseteq^{\mathtt{A}} \mathit{align}(p\,;\ q)$, but as we can show below, it is enough to show $\phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2)\,;\,(u_2\,;\ \mathsf{sync})$.

$$\begin{aligned}
&\implies \phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2)\,;\,(u_2\,;\ \mathsf{sync}) && \text{By } \subseteq^{\mathtt{A}}\text{-DEF}\\
&\implies \phi \subseteq^{\mathtt{A}} \big(a_1\,;\,(u_1 \mathbin{⨾} a_2), u_2\big) && \text{By definition of align}\\
&\implies \phi \subseteq^{\mathtt{A}} \mathit{align}(p\,;\ q)
\end{aligned}$$

Case $\subseteq^{\mathtt{W}}$-SEQ-L. We have $\phi \subseteq^{\mathtt{W}} p$, $\mathit{align}(p) = (a_1, u_1)$, $\mathit{align}(q) = (a_2, u_2)$. Our induction hypothesis is that $\phi \subseteq^{\mathtt{A}} \mathit{align}(p)$, thus (IH) $\phi \subseteq^{\mathtt{A}} a_1$ or (IH) $\phi \subseteq^{\mathtt{U}} u_1$. As we show below, we only need to show that either $\phi \subseteq^{\mathtt{A}} a_1$ holds or $\phi \subseteq^{\mathtt{A}} u_1 \mathbin{⨾} a_2$ holds, which is easy to conclude from our induction hypothesis and Lemma 7.

$$\begin{aligned}
&\phi \subseteq^{\mathtt{A}} a_1 \lor \phi \subseteq^{\mathtt{A}} u_1 \mathbin{⨾} a_2 && \text{By } \subseteq^{\mathtt{A}}\text{-SEQ-L or } \subseteq^{\mathtt{A}}\text{-SEQ-R}\\
\implies &\phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2) && \text{By } \subseteq^{\mathtt{A}}\text{-SEQ-L}\\
\implies &\phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2)\,;\,(u_2\,;\ \mathsf{sync})
\end{aligned}$$

Case $\subseteq^{\mathtt{W}}$-SEQ-R. Our induction hypothesis is that $\phi \subseteq^{\mathtt{A}} \mathit{align}(q)$, thus (IH) $\phi \subseteq^{\mathtt{A}} a_2$ or (IH) $\phi \subseteq^{\mathtt{U}} u_2$. As we show below, we only need to show that either $\phi \subseteq^{\mathtt{A}} u_1 \mathbin{⨾} a_2$ holds or $\phi \subseteq^{\mathtt{U}} u_2$ holds, which is easy to conclude from our induction hypothesis and Lemma 8.

$$\begin{aligned}
&\phi \subseteq^{\mathtt{A}} u_1 \mathbin{⨾} a_2 \lor \phi \subseteq^{\mathtt{U}} u_2 && \text{By } \subseteq^{\mathtt{A}}\text{-SEQ-R and } \subseteq^{\mathtt{A}}\text{-SYNC, resp.}\\
\implies &\phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2) \lor \phi \subseteq^{\mathtt{A}} u_2\,;\ \mathsf{sync} && \text{By } \subseteq^{\mathtt{A}}\text{-SEQ-L or } \subseteq^{\mathtt{A}}\text{-SEQ-R, resp.}\\
\implies &\phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2)\,;\,(u_2\,;\ \mathsf{sync})
\end{aligned}$$

Case $\subseteq^{\mathtt{W}}$-SEQ-B. We have that $\phi = (\alpha_1, \alpha_2)$, (Ha) $\alpha_1 \in^{\mathtt{W}} \mathit{lst}(p)$, and (Hb) $\alpha_2 \in^{\mathtt{W}} \mathit{fst}(q)$. The proof of this case is simple.

$$\begin{aligned}
&\alpha_1 \in^{\mathtt{W}} \mathit{lst}(p) \land \alpha_2 \in^{\mathtt{W}} \mathit{fst}(q) && \text{By Lemma 3 and Lemma 4, resp.}\\
\implies &\alpha_1 \in^{\mathtt{U}} u_1 \land \alpha_2 \in^{\mathtt{A}} \mathit{fst}(a_2) && \text{By Lemma 9}\\
\implies &\phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2) && \text{By } \subseteq^{\mathtt{A}}\text{-SEQ-L}\\
\implies &\phi \subseteq^{\mathtt{A}} a_1\,;\,(u_1 \mathbin{⨾} a_2)\,;\,(u_2\,;\ \mathsf{sync})
\end{aligned}$$

□

5.2 Correctness of *split*

First Figure 13 (top) formalizes an intermediary pair-inclusion judgment for protocol closures, following similar lines to the judgment for symbolic traces in Figure 14. Next we show that if a pair-inclusion holds for an aligned protocol, Pair-inclusion (protocol closures) $\boxed{\phi \subseteq^{\mathtt{C}} c}$

$$\frac{\phi \subseteq^{\mathtt{U}} u}{\phi \subseteq^{\mathtt{C}} u} \qquad \frac{n\downarrow i \qquad n\downarrow k \qquad \exists i\colon i \le j < k \qquad \phi \subseteq^{\mathtt{C}} c[x := j]}{\phi \subseteq^{\mathtt{C}} \mathsf{var}\ x\ \mathsf{in}\ n..m\,;\ c}$$

Induction principle for aligned protocols $\boxed{a\downarrow}$

$$\frac{}{u\,;\ \mathsf{sync}\downarrow} \qquad \frac{a\downarrow \quad a'\downarrow}{a\,;\ a'\downarrow}$$

$$\frac{a\downarrow \quad \forall i,k\colon n\downarrow i \implies m\downarrow k \implies \forall j\colon i \le j < k \implies a'[j := x]\downarrow}{a\,;\ \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{a'\}\downarrow}$$

Fig. 13: Auxiliary judgments for protocol closures and aligned protocols.

then there must be a protocol closure produced by *split* that also contains that pair (using the judgment in Figure 13).

Lemma 11. *If $\phi \subseteq^{\mathtt{A}} (a,\ u)$, then $\exists c, c \in \mathit{split}((a,\ u))$ and $\phi \subseteq^{\mathtt{C}} c$.*

Proof. [Source: lemma in_phases_1, file `faial-coq/src/PhaseSplit.v`] The proof is trivial once we prove the following: we can show that if $\phi \subseteq^{\mathtt{A}} a$, then $\exists c, c \in \mathit{split}(a)$ and $\phi \subseteq^{\mathtt{C}} c$, which follows by induction on the derivation of assumption $\phi \subseteq^{\mathtt{A}} a$. □

Finally, Figure 13 (bottom) introduces another intermediate judgment, which we use as a basis for a specialized induction principle in Lemma 12. This lemma is key to formalize the transformation from $\mathcal{C}$ to $\mathcal{A}$. Essentially, $a\downarrow$ formalizes the big step semantics of an aligned protocol a without producing histories.

Lemma 12. *If $a\downarrow$, $c \in \mathit{split}((a,\ u))$, and $\phi \subseteq^{\mathtt{C}} c$, then $\phi \subseteq^{\mathtt{A}} (a,\ u)$.*

Proof. [Source: lemma i_pair_in_ph, file `faial-coq/src/PhaseSplit.v`] The interesting part is to show that if $a\downarrow$, $c \in \mathit{split}(a)$, and $\phi \subseteq^{\mathtt{C}} c$, then $\phi \subseteq^{\mathtt{A}} a$, which follows by induction on the derivation of $a\downarrow$. □

5.3 Correctness of sequentialization (*seq*)

We define the phase membership and pair inclusion judgments for symbolic traces in Figure 14. Phase membership is straightforward and resembles that of unsynchronized protocols.

Pair inclusion is more interesting, for two reasons. Firstly, since symbolic traces have no notion of parallelism, pairs cannot be split in a conditional, both

Phase membership (symbolic traces) $\boxed{\alpha \in^{\mathtt{H}} h}$

$$\frac{n\downarrow i \qquad m\downarrow j}{i{:}o[j] \in^{\mathtt{H}} n{:}o[m]} \qquad \frac{\alpha \in^{\mathtt{H}} h}{\alpha \in^{\mathtt{H}} h\,;\, h'} \qquad \frac{\alpha \in^{\mathtt{H}} h'}{\alpha \in^{\mathtt{H}} h\,;\, h'} \qquad \frac{b\downarrow \mathtt{true} \qquad \alpha \in^{\mathtt{H}} h}{\alpha \in^{\mathtt{H}} \textsf{if}\ b\ \{h\}\ \textsf{else}\ \{h'\}}$$

$$\frac{b\downarrow \mathtt{false} \qquad \alpha \in^{\mathtt{H}} h'}{\alpha \in^{\mathtt{H}} \textsf{if}\ b\ \{h\}\ \textsf{else}\ \{h'\}} \qquad \frac{n\downarrow i \qquad m\downarrow k \qquad \exists j\colon i \le j < k \qquad \alpha \in^{\mathtt{H}} h[x \coloneqq j]}{\alpha \in^{\mathtt{H}} \textsf{var}\ x\ \textsf{in}\ n..m;\ h}$$

Pair-inclusion (symbolic traces) $\boxed{\phi \subseteq^{\mathtt{H}} h}$

$$\frac{\phi \subseteq^{\mathtt{H}} h}{\phi \subseteq^{\mathtt{H}} h\,;\, h'} \qquad \frac{\phi \subseteq^{\mathtt{H}} h'}{\phi \subseteq^{\mathtt{H}} h\,;\, h'} \qquad \frac{\alpha_1 \in^{\mathtt{H}} h \qquad \alpha_2 \in^{\mathtt{H}} h'}{(\alpha_1, \alpha_2) \subseteq^{\mathtt{H}} h\,;\, h'}$$

$$\frac{b\downarrow \mathtt{true} \qquad \phi \subseteq^{\mathtt{H}} h}{\phi \subseteq^{\mathtt{H}} \textsf{if}\ b\ \{h\}\ \textsf{else}\ \{h'\}} \qquad \frac{b\downarrow \mathtt{false} \qquad \phi \subseteq^{\mathtt{H}} h'}{\phi \subseteq^{\mathtt{H}} \textsf{if}\ b\ \{h\}\ \textsf{else}\ \{h'\}}$$

$$\frac{n\downarrow i \qquad m\downarrow k \qquad \exists j\colon i \le j < k \qquad \phi \subseteq^{\mathtt{H}} h[x \coloneqq j]}{\phi \subseteq^{\mathtt{H}} \textsf{var}\ x\ \textsf{in}\ n..m;\ h}$$

Fig. 14: Phase membership and pair inclusion (symbolic traces).

accesses of a pair ϕ must originate from the same branch. Secondly, there is no rule for a single access because splitting preserves concurrent accesses that originate from distinct threads. Note that splitting does not encode concurrent accesses from a *same* thread, *e.g.*, any two accesses from different iterations of the same loop and of the same thread. We are therefore only able to prove that $(\alpha_1, \alpha_2) \subseteq^{\mathtt{U}} u$ implies $(\alpha_1, \alpha_2) \subseteq^{\mathtt{H}} \mathit{seq}(u)$ when $\mathit{owner}(\alpha_1) \neq \mathit{owner}(\alpha_2)$. We do not need a pair-inclusion rule for accesses as the only way for a pair to originate from a memory access is when the same access appears in both elements of the pair (that case that is irrelevant for correctness).

We first relate phase memberships of unsynchronized protocols and symbolic traces with Lemma 13 (whose proof is omitted here).

Lemma 13. *If* $0 \le i < |\mathbb{T}|$, *then* $\alpha \in^{\mathtt{H}} \mathit{trace}(u)[\textsf{tid} \coloneqq i]$ *if, and only if,* $\alpha \in^{\mathtt{U}} u$.

[Source: lemma s_in_p_in_iff, file `faial-coq/src/Sequentialization.v`]

Next, we show that the accesses in the sequentialization of u are those in u. Recall that *seq* uses *trace* to instantiate two symbolic threads (see Figure 8).

Lemma 14. *If* $\alpha \in^{\mathtt{H}} \mathit{seq}(u)$, *then* $\alpha \in^{\mathtt{U}} u$.

Proof. [Source: lemma i_in_sequentialize_to_t_in, file `faial-coq/src/-Sequentialization.v`] We have $h_1 = \mathit{trace}(u)[\mathsf{tid} := t_1]$, $h_2 = \mathit{trace}(u)[\mathsf{tid} := t_2]$, and $\mathit{seq}(u) = \mathsf{var}\ t_1\ \mathsf{in}\ 1..|\mathbb{T}|;\ \mathsf{var}\ t_2\ \mathsf{in}\ 0..t_1;\ h_1 \mathbin{;} h_2$, and (1) $\alpha \in^{\mathtt{H}} \mathit{seq}(u)$. We have to show $\alpha \in^{\mathtt{U}} u$. From (1), we obtain there exist i and j such that $0 \leq i < j$ and $1 \leq j < |\mathbb{T}|$, and (2) $\alpha \in^{\mathtt{H}} (\mathit{trace}(u)[\mathsf{tid} := t_1]) \mathbin{;} (\mathit{trace}(u)[\mathsf{tid} := t_2])$ By inverting (2) we get that $\alpha \in^{\mathtt{H}} \mathit{trace}(u)[\mathsf{tid} := t_1]$ or $\alpha \in^{\mathtt{H}} \mathit{trace}(u)[\mathsf{tid} := t_2]$, and we can conclude each by applying Lemma 13. □

We build on Lemma 14 to obtain the result for pair-inclusions.

Lemma 15. *If $\phi \subseteq^{\mathtt{H}} \mathit{seq}(u)$, then $\phi \subseteq^{\mathtt{U}} u$.*

Proof. [Source: lemma i_pair_in_1, file `faial-coq/src/Sequentialization.v`] Let $\phi = (\alpha_1, \alpha_2)$. We get (1) $\alpha_1 \in^{\mathtt{H}} \mathit{seq}(u)$ and (2) $\alpha_2 \in^{\mathtt{H}} \mathit{seq}(u)$ from our hypothesis. We get (3) $\alpha_1 \in^{\mathtt{U}} u$ and (4) $\alpha_2 \in^{\mathtt{U}} u$ from Lemma 14 applied to (1) and to (2). We conclude using the definition of pair-inclusion. □

The other direction exposes the subtlety we anticipated, pair-inclusion is only preserved when the accesses in the pair have different owners.

Lemma 16. *Let $\phi = (\alpha_1, \alpha_2)$. If $\phi \subseteq^{\mathtt{U}} u$ and $\mathit{owner}(\alpha_1) \neq \mathit{owner}(\alpha_2)$, then $\phi \subseteq^{\mathtt{H}} \mathit{seq}(u)$.*

Proof. [Source: lemma i_pair_in_2, file `faial-coq/src/Sequentialization.v`] We have $h_1 = \mathit{trace}(u)[\mathsf{tid} := t_1]$, $h_2 = \mathit{trace}(u)[\mathsf{tid} := t_2]$, and $\mathit{seq}(u) = \mathsf{var}\ t_1\ \mathsf{in}\ 1..|\mathbb{T}|;\ \mathsf{var}\ t_2\ \mathsf{in}\ 0..t_1;\ h_1 \mathbin{;} h_2$, and (1) $\alpha \in^{\mathtt{U}} u$. We have to show $\alpha \in^{\mathtt{H}} \mathit{seq}(u)$. We know that either $\mathit{owner}(\alpha) = 0$ or $\mathit{owner}(\alpha) > 0$.

Case $\mathit{owner}(\alpha) = 0$. The only thread that could have emitted α is thread t_2, as $t_1 > 0$. Thus, let t_1 be 1 and t_2 be 0. It is enough to show,

$$\begin{aligned} &\alpha \in^{\mathtt{H}} (\mathit{trace}(u)[\mathsf{tid} := 1]) \mathbin{;} (\mathit{trace}(u)[\mathsf{tid} := 0]) \\ \Longrightarrow\ &\alpha \in^{\mathtt{H}} \mathit{trace}(u)[\mathsf{tid} := 0] \end{aligned}$$

which holds from Lemma 13.

Case $\mathit{owner}(\alpha) > 0$. Let t_1 be $\mathit{owner}(\alpha)$ and t_2 be 0. We must show

$$\begin{aligned} &\alpha \in^{\mathtt{H}} (\mathit{trace}(u)[\mathsf{tid} := \mathit{owner}(\alpha)]) \mathbin{;} (\mathit{trace}(u)[\mathsf{tid} := 0]) \\ \Longrightarrow\ &\alpha \in^{\mathtt{H}} \mathit{trace}(u)[\mathsf{tid} := \mathit{owner}(\alpha)] \end{aligned}$$

which also holds from Lemma 13. □

5.4 Main Result

We first show that sequentialization reflects pair-inclusion.

Lemma 17. *If $\phi \subseteq^{\mathtt{H}} \mathit{seq}(c)$, then $\phi \subseteq^{\mathtt{C}} c$.*

Proof. [Source: lemma `in_1`, file `faial-coq/src/Main.v`] The proof follows by induction on the derivation of assumption $\phi \subseteq^{\mathtt{H}} \mathit{seq}(c)$. To prove the base case we need Lemma 15; the inductive case follows easily. □

Next, we show that sequentialization preserves pair-inclusion.

Lemma 18. *Let* $\phi = (\alpha_1, \alpha_2)$*. If* $\mathit{owner}(\alpha_1) \neq \mathit{owner}(\alpha_2)$ *and* $\phi \subseteq^{\mathtt{C}} c$*, then* $\phi \subseteq^{\mathtt{H}} \mathit{seq}(c)$*.*

Proof. [Source: lemma `in_2`, file `faial-coq/src/Main.v`] The proof follows by induction on the derivation of assumption $\phi \subseteq^{\mathtt{C}} c$. To prove the base case we use Lemma 16 and the inductive case follows easily. □

Finally, we restate and prove our main result. We depict graphically the key elements of the proof below.

$$\begin{tikzcd}
\phi \subseteq^{\mathtt{W}} p \arrow[r, bend right, "\text{Lemma 10}"'] & \phi \subseteq^{\mathtt{A}} \mathit{align}(p) \arrow[l, bend right, "\text{Lemma 6}"'] \arrow[r, bend right, "\text{Lemma 11}"'] & \phi \subseteq^{\mathtt{C}} \mathit{split}(\mathit{align}(p)) \arrow[l, bend right, "\text{Lemma 12}"'] \arrow[r, bend right, "\text{Lemma 18}"'] & \phi \subseteq^{\mathtt{H}} \mathit{seq}(\mathit{split}(\mathit{align}(p))) \arrow[l, bend right, "\text{Lemma 17}"']
\end{tikzcd}$$

Theorem 1. *If* $p \downarrow H_1$ *and* $\mathit{seq}(\mathit{split}(\mathit{align}(p))) \Downarrow H_2$*, then* $\mathit{safe}(H_1)$ *if and only if* $\mathit{safe}(H_2)$*.*

Proof. **Case ($\Longrightarrow$).** [Source: lemma `drf_2`, file `faial-coq/src/Main.v`] In order to show $\mathit{safe}(H_2)$, we must show that for any $\phi = (\alpha_1, \alpha_2)$ where $\mathit{owner}(\alpha_1) \neq \mathit{owner}(\alpha_2)$ and $\phi \in P \in H_2$ for some P, then $\alpha_1 \mathrel{\cancel{\mathit{racy}}} \alpha_2$. By applying $\mathit{safe}(H_1)$, we have that there is a phase P_2 such that $\phi \in P_2 \in H_2$, and we are left with showing that there exists a phase P_1 where $\phi \in P_1 \in H_1$. Thus, we need to show $\phi \subseteq^{\mathtt{H}} p$.

From $\mathit{seq}(\mathit{split}(\mathit{align}(p))) \downarrow H_2$ and $\phi \in P_2 \in H_2$, we have that there exist a symbolic history h and a history H_h such that $h \in \mathit{seq}(\mathit{split}(\mathit{align}(p)))$ and $\phi \in P_2 \in H_h$. Thus, there exists c such that (1) $\phi \subseteq^{\mathtt{H}} \mathit{seq}(c)$ and (2) $\mathit{seq}(c) \in \mathit{split}(\mathit{align}(p))$. Applying Lemma 17 to $\phi \subseteq^{\mathtt{H}} \mathit{seq}(c)$ yields (3) $\phi \subseteq^{\mathtt{C}} c$. We can obtain (4) $\mathit{align}(p)\downarrow$ from $p \downarrow H_1$. By applying Lemma 12 to (2), (3), and (4), we obtain $\phi \subseteq^{\mathtt{A}} \mathit{align}(p)$. Finally, we use Lemma 6 to conclude.

Case ($\Longleftarrow$). [Source: lemma `drf_1`, file `faial-coq/src/Main.v`] In order to show $\mathit{safe}(H_1)$, we must show that for any $\phi = (\alpha_1, \alpha_2)$ where $\mathit{owner}(\alpha_1) \neq \mathit{owner}(\alpha_2)$ and $\phi \in P \in H_1$ for some P, then $\alpha_1 \mathrel{\cancel{\mathit{racy}}} \alpha_2$. By applying $\mathit{safe}(H_2)$, we have that there is a phase P_1 such that $\phi \in P_1 \in H_2$, and we are left with showing that there exists a phase P_2 where $\phi \in P_2 \in H_2$. Thus, we need to show that there exists a c such that $\phi \subseteq^{\mathtt{H}} \mathit{seq}(c)$ and $\mathit{seq}(c) \in \mathit{split}(\mathit{align}(p))$.

From $p \downarrow H_1$ and $\phi \in P_1 \in H_1$, we have that (1) $\phi \subseteq^{\mathtt{W}} p$. Applying Lemma 10 to (1) yields (2) $\phi \subseteq^{\mathtt{A}} \mathit{align}(p)$. Applying Lemma 11 to (2) yields that there exists c such that (3) $c \in \mathit{split}(\mathit{align}(p))$ and (4) $\phi \subseteq^{\mathtt{C}} c$. We apply Lemma 18 to (4) and obtain (5) $\phi \subseteq^{\mathtt{W}} \mathit{seq}(c)$. □

5.5 Proof of compositionality

We first give the formal definition of DRF for aligned protocols, then re-state Proposition 1 which establishes the compositionality of our approach.

Definition 3. *We say that a is DRF if, and only if, for all (α_1, α_2), if $\phi \subseteq^{\mathtt{A}} a$ then $\alpha_1 \mathrel{\cancel{\mathit{racy}}} \alpha_2$.*

Proposition 1. *Let $\mathcal{E}$ be a context, s.t. $\mathcal{E}[\mathsf{skip}\,;\,\mathsf{sync}]$ is DRF, and $\mathcal{E}[\mathsf{skip}\,;\,\mathsf{sync}]\downarrow$. For all $a \in \mathcal{A}$, if a is DRF, $a\downarrow$, and $\mathit{fv}(a) \subseteq \{\mathsf{tid}\}$, then $\mathcal{E}[a]$ is also DRF.*

Proof. [Source: lemma compositionality, file `faial-coq/src/Compositionality.v`] The proof follows by induction on the derivation of $a\downarrow$. □

6 Implementation

In this section we present our tool, Faial, that implements the steps described in Figure 1. Faial takes a CUDA kernel as input and produces results that either identify the kernel as DRF or list specific data-races. In this section, we describe the implementation of the protocol inference, well-formedness checks, and transformation to SMT.

Inference This step transforms a CUDA kernel into memory access protocols (one for each shared array). We use `libclang` [26] to parse the kernel, a standard single static assignment (SSA) transformation to simplify the analysis of indices and arrays, and a form of code slicing to only retain code related to *shared* array accesses. We note that Faial supports constructs of the CUDA programming model that are not directly modeled by memory access protocols, *e.g.*, unstructured loops, function calls, and multi-dimensional arrays.

Well-formedness This step ensures that kernels Faial analyzes meet the restrictions induced by the semantics of $\mathcal{W}$, *e.g.*, synchronized loops iterate at least once. First, Faial annotates loops with a synchronized/unsynchronized tag according to the presence of sync in the loop body, then adjusts the precedence of sequencing to group all unsynchronized code preceding a sync or a synchronized loops. Synchronized loops of well-formed protocols cannot manipulate thread-local variables (*i.e.*, tid), an assumption shared by the CUDA programming model. Hence, Faial flags such kernels as erroneous. Next, Faial adds assertions before/after synchronized loops to check that the loop range is non-empty, *i.e.*, loops execute at least once. Similarly to loops, conditionals are tagged as synchronized when at least one branch has a barrier synchronization. Then, Faial inlines synchronized conditionals, *i.e.*, when a synchronized conditional is found, two copies of the input program are created and each copy is prefixed by a global assertion corresponding to the condition. Faial does not support synchronized conditionals that appear within synchronized loops. We have not found real-world kernels that include such a construction.

Quantification This step transforms each symbolic trace (Figure 7) into an SMT formula that proves the existence of a data-race, *c.f.*, Figure 5. The generated formula tests whether any two threads access the same location of the array and at least one them is writing. A kernel is DRF when the formula is *not* satisfiable. For multi-dimensional arrays, we generate one pair of indices per dimension, and check that at least one pair is distinct. We illustrate this straightforward transformation with Example 5.

Example 5. *The formula generated from the trace in Example 4 is given below:*

$$\begin{aligned}\forall t_1, t_2 \colon 1 \leq t_1 < |\mathbb{T}| \land 0 \leq t_2 < t_1 \land (\mathsf{m}_1 = \mathsf{wr} \lor \mathsf{m}_2 = \mathsf{wr}) \implies \\ \big((\mathsf{idx}_1 = t_1 + 1 \land \mathsf{m}_1 = \mathsf{wr}) \lor (\mathsf{idx}_1 = t_1 + 2 \land \mathsf{m}_1 = \mathsf{rd})\big) \\ \land \big((\mathsf{idx}_2 = t_2 + 1 \land \mathsf{m}_2 = \mathsf{wr}) \lor (\mathsf{idx}_2 = t_2 + 2 \land \mathsf{m}_2 = \mathsf{rd})\big) \land \mathsf{idx}_1 \neq \mathsf{idx}_2\end{aligned}$$

*where each symbolic access is translated to a conjunction representing its index (*idx*) and access mode (*m*). Observe that the formula enforces that indices* idx_1 *and* idx_2 *(executed by distinct threads) are different.*

Faial uses Z3 [27] to check the satisfiability of the generated formulas. Our theory makes little assumptions about the expression language, which in turn lets users of Faial to pick any logic that Z3 supports. Faial generates formulas without existential quantifiers and pulls out the forall-binders to the beginning of the formula. Hence, Faial can use unquantified logics. The version of Faial included in the artifact [16] uses QF_LIA. Users can change the logic parameter to handle more intricate kernels.

Multi-dimensional thread identifiers and kernel parameters We extend memory access protocol with thread-local parameters to support multi thread identifiers, and with thread-global parameters to support kernel parameters. We handle thread-local parameters similarly to tid: declare two variables per thread-local parameter when invoking function *seq*. We handle thread-global parameters as a synchronized loop variable: declare one variable per thread-global parameter. For instance, say that N, an unsigned C-integer, is a parameter of some kernel p, then we encode the parameter essentially as $\mathsf{for}^{\mathsf{s}}\ \mathsf{N} \in 0..\mathsf{MAX\_INT}\ \{p\}$, where MAX_INT is the maximum C-integer.

Forms of synchronization CUDA supports multiple levels of synchronization. Within a single accelerator card, threads are divided into *grids*, grids are divided into *groups*, and groups are divided into *warps*. CUDA enables all-to-all barrier synchronization between elements of: devices, grids, groups, and warps. CUDA further supports: a low-level form of message-passing (*i.e.*, data-exchange) synchronization for warps, dynamic parallelism (creating new ways to group threads), graph based concurrency, cooperative scheduling, atomics, and overlapping kernel calls. Presently, we have only implemented support for group synchronization, as this is the most common form of synchronization in use, *e.g.*, in [28], out of 389 kernels, only 7 (1.7%) use warp synchronization, and 22 (5.7%) use atomics. Adding support for atomics and barrier synchronization at other levels appears to be mostly an engineering effort with little impact on our theory. For instance, in [10, 28] adding support for grid-synchronization, warp-synchronization, and atomics only affects the definition of data-race marginally. We discuss which new CUDA features we plan to support in Section 9.

Loops Whenever possible loops are transformed to loops with a stride of 1 following ideas from loop normalization [13] and abstraction [25]. For instance, in **for**(int i=lb;i<ub;i+=s){S} we change the stride from s into 1 by executing the loop body S when the loop variable i is divisible by stride, *i.e.*, the loop becomes **for**(int i=lb;i<ub;i++) **if**((i+lb)%s==0){S}. Similarly, a loop ranging over powers of n, *e.g.*, **for**(int i=lb;i<ub;i*=s), becomes **for**(int i=lb;i<ub;i++) **if**(powerof(i,s)){S}, where function powerof(i,s) tests whether i is a power of base s. We approximate **while**s as a structured loop with an unknown upper bound.

Arrays in global and shared memory Threads can issue shared access in CUDA via *global memory* and via *shared memory*. Faial supports arrays declared in both kinds of memory. Any array either in the kernel parameters, or marked with a __device__ modifier is in the global memory. Any variable declared with a __share__ modifier is in the shared memory. Note that even scalar variables with a __shared__ modifier can be subject to data-races, so our tool represents scalars in the shared memory as arrays of length 1.

Array aliasing Our tool assumes that all arrays (kernel parameters and __share__ arrays) and are disjoint. Faial has some support for aliasing inside the kernel, *e.g.*, a programmer could write the statement float*y =x + threadIdx.x; and then use y[0] as an alias to x[threadIdx.x]. Our tool does not support alias in loops and has limited support for array aliasing in conditional branches.

Function calls Function calls that manipulate shared memory are uncommon in GPU programming. Additionally auxiliary functions that manipulate shared memory have a compiler annotation to inline their bodies, hence we can inline such calls easily. Faial cannot handle recursive functions, but these rarely occur in practice. Function calls that do not access shared memory are simply discarded.

7 Experimental Evaluation

We evaluate Faial over several datasets and show how it fares against existing approaches. We structure this evaluation in three claims.

Claim 1: Correctness. We claim that our approach finds more bugs and raises fewer false alarms than existing tools. To evaluate this claim, we compare Faial against four state-of-the-art kernel verification tools over 10 kernels that are known to be tricky to analyze.
Claim 2: Scalability. We claim that our approach scales better to larger programs. To evaluate this claim, we compare Faial against other tools over a set of synthetic benchmarks designed to test the limits of each tool, in terms of run time and memory usage.
Claim 3: Real-world usability. We claim that our approach is more usable than existing static verification tools on real-world CUDA programs. To evaluate this claim, we use a varied dataset of real-world data-race free kernels and measure the false alarm rate, run time, and memory usage of Faial, GPUVerify, and PUG.

Benchmarking environment To make our evaluation reproducible, we developed a benchmarking framework to automate our experiments over the different tools and datasets. For Claim 1 and Claim 3, we designed a tool-agnostic file format for kernel functions and associated metadata (*e.g.*, expected result of DRF analysis, grid and block dimensions, and include directives). For Claim 2, we created a tool that generates kernels according to given templates, *e.g.*, see Figure 17.

We evaluate Faial against the following verification tools: GPUVerify [10] v2018-03-22[2], PUG [13] v0.2[3], GKLEE [7] v3.0[4], and SESA [8] v3.0[5]. We picked the last release of each tool as of March of 2021. The binaries, source code, and URL of each tool included in our artifact [16]. Experiments for Claim 1 use an Intel i5-6500 CPU, 7.7GiB RAM, and Fedora 33 OS, while Claim 2 and Claim 3 use an Intel i7-10510U CPU, 16GiB RAM, and Pop! OS.

Excluded tools We excluded ESBMC-GPU [9] and Simulee [14] from the evaluation because we were unable to get them to run satisfactorily. Both tools have rudimentary support for verifying arbitrary CUDA kernels. ESBMC-GPU did not find a single data-race in our benchmarks, while Simulee produced false alarms for every DRF-kernel given.

Claim 1: Correctness

We have selected a set of tricky kernels to expose false alarms and missed data-races in Faial, GPUVerify, PUG, GKLEE, and SESA. Our results are reported in Table 1. The dataset consists of 5 tests, each consisting of two variations of the same kernel: one racy and one DRF. The racy version of Test 1 (*c.f.*, Listing 2.1) contains an inter-iteration data-races. The DRF version adds a sync after the second inner loop. Tests 2 to 4 expose various loop-related data-races. Their protocols are given in Figure 15. In the racy version of Test 2, $\mathsf{wr}[\mathsf{tid} + 1]$ conflicts with $\mathsf{wr}[\mathsf{tid}]$ of the first iteration. Similarly, in the racy version of

[2] https://github.com/mc-imperial/gpuverify/releases/tag/2018-03-22
[3] http://formalverification.cs.utah.edu/PUG/distributions/pug-v0.2_x64.tar.gz
[4] https://github.com/Geof23/Gklee/tree/77214669c83ac2f20802299df61940f2dd91360c
[5] https://github.com/Geof23/Gklee/tree/77214669c83ac2f20802299df61940f2dd91360c

Table 1: Results for Claim 1. DRF indicates that a (static analysis) tool reported a test case as DRF. NRR indicates that a (symbolic execution) tool did not report any data-race. Label x/y indicates that the tool reported y data-races, x of which are actual races. Label *timeout* indicates that the tool did not terminate within 90s. A test passes if the tool returns the expected result and all reported races are valid.

Test	Expected	Faial	GPUVerify	PUG	GKLEE	SESA
1 transposeDiagonal	Racy	**1/1**	*0/2*	*DRF*	*timeout*	*timeout*
	DRF	**DRF**	*0/1*	**DRF**	*timeout*	*timeout*
2 first-iter	Racy	**1/1**	*0/1*	**1/1**	*timeout*	*timeout*
	DRF	**DRF**	*0/1*	*0/1*	*timeout*	*timeout*
3 last-iter	Racy	**1/1**	**1/1**	*0/1*	*timeout*	*timeout*
	DRF	**DRF**	*0/1*	**DRF**	*timeout*	*timeout*
4 last-iter-first-iter	Racy	**1/1**	*0/1*	*0/1*	*timeout*	*timeout*
	DRF	**DRF**	*0/1*	*0/1*	*timeout*	*timeout*
5 read-index	Racy	*0/1*	**1/1**	*0/1*	*NRR*	*NRR*
	DRF	*0/1*	**DRF**	*0/1*	**NRR**	**NRR**
Number of tests passed (of 5):		4	1	0	0	0

```
// first-iter
wr[tid+1];
for^S x in 0..N {
  if (x > 0)
    {wr[tid]};
    sync}
```

```
// last-iter
for^S x in 0..N {
  sync;
  if (tid < T-1)
    {wr[tid+1]}};
wr[tid + |T|]
```

```
// last-iter-first-iter
for^S x in 1..N+1 {
    for^S y in 1..x+1 {
        sync; wr[tid+x+y]}};
for^S z in N*2..N*3 {
    wr[tid+z+1]; sync}
```

Fig. 15: Protocols for Tests 2 to 4, *c.f.*, Claim 1, where N is a free thread-global variable. Yellow shaded code only appears in the DRF version of first-iter and last-iter. Red shaded code only appears in the racy version of last-iter-first-iter.

```
// Racy kernel        // Protocol A
A[tid] = tid;         wr[tid];
int x = A[tid];       rd[tid];
A[x+1] = 0;           wr[x+1]
```

```
// DRF kernel         // Protocol A
A[tid] = tid;         wr[tid];
int x = A[tid];       rd[tid];
A[x] = 0;             wr[x]
```

Fig. 16: Kernels and protocols for Test 5 (read-index), *c.f.*, Claim 1. Note that x becomes a free thread-local variable as protocols do not model array elements.

Test 3, wr[tid + 1] of the last iteration races with wr[tid]. In the racy version of Test 4 the last iteration of a nested loop races with the first iteration of the following loop. Test 5 exposes the abstraction gap between kernel and memory access protocols (which abstract away array elements), see Figure 16.

Faial passes more tests than any other tool. Failed Test 5 is caused by memory access protocols abstracting away from *what* data is being read

```
// accesses
rd[tid + n1*|T|];
wr[tid + 1*|T|];
rd[tid + n2*|T|];
wr[tid + 2*|T|];
// ...
```

```
// barriers
wr[tid];
sync;
wr[tid];
sync;
// ...
```

```
// conditionals
if tid==0
  {wr[tid]};
if tid==1
  {wr[tid]};
// ...
```

```
// unsynchronized loops
for^U i1 in 0..N {
  wr[tid];
  for^U i2 in 0..N {
    wr[tid];
    // ...          }}
```

```
// synchronized loops
for^S i1 in 0..N {
  wr[tid]; sync;
  for^S i2 in 0..N {
    wr[tid]; sync;
    // ...          }}
```

Fig. 17: Synthetic protocols generated for Claim 2. N is a free thread-global variable, and n_1, n_2... are positive integer literals.

from/written to arrays, *i.e.*, array elements. In each case, Faial reports one spurious data race (*0/1*). We report on performance trade-offs wrt. tracking array elements in Claim 2.

GPUVerify passes Test 5 because it tracks array elements, but fails the remaining 4 tests. Some reported false alarms are ill-formed, *e.g.*, on the racy component of Test 2, the report (0 : wr[tid]; 16 : wr[tid]) has disjoint indices.

PUG obtains the worst score amongst static tools. Notably, the tool misses a data-race in Test 1, demonstrating its unsoundness, *c.f.*, Section 2.1.

GKLEE and SESA timeout for tests that include loops, as the loop bounds are unknown. Both tools miss the data-race in Test 5. Symbolic tools may be able to report data-races when the bound is known, *e.g.*, timeouts start in Test 1 when the bound is at least 2, in Test 2 when the bound is at least 23, 000.

Claim 2: Scalability

We evaluate how different kernel constructs affect run time and memory usage of Faial, GKLEE, GPUVerify, PUG, and SESA with a synthetic dataset. This experiment tries to push the limits of each tool to measure their asymptotic behavior. Our dataset is divided into five categories, one per syntactical construct in the language of memory access protocols. Figure 17 shows the protocols of the kernel patterns we generate in each category: (*i*) repeated accesses (read then write), (*ii*) repeated barrier synchronizations separated by writes, (*iii*) repeated conditionals, (*iv*) increasingly nested unsynchronized loops, and (*v*) increasingly nested synchronized loops. In each category, we vary the problem size by repeating a pattern from 1 to 50 times. Note that all kernels generated this way are DRF.

Figure 18 shows the average run time and memory usage over five runs on logarithmic and linear scales, respectively. For each run, we set a timeout of

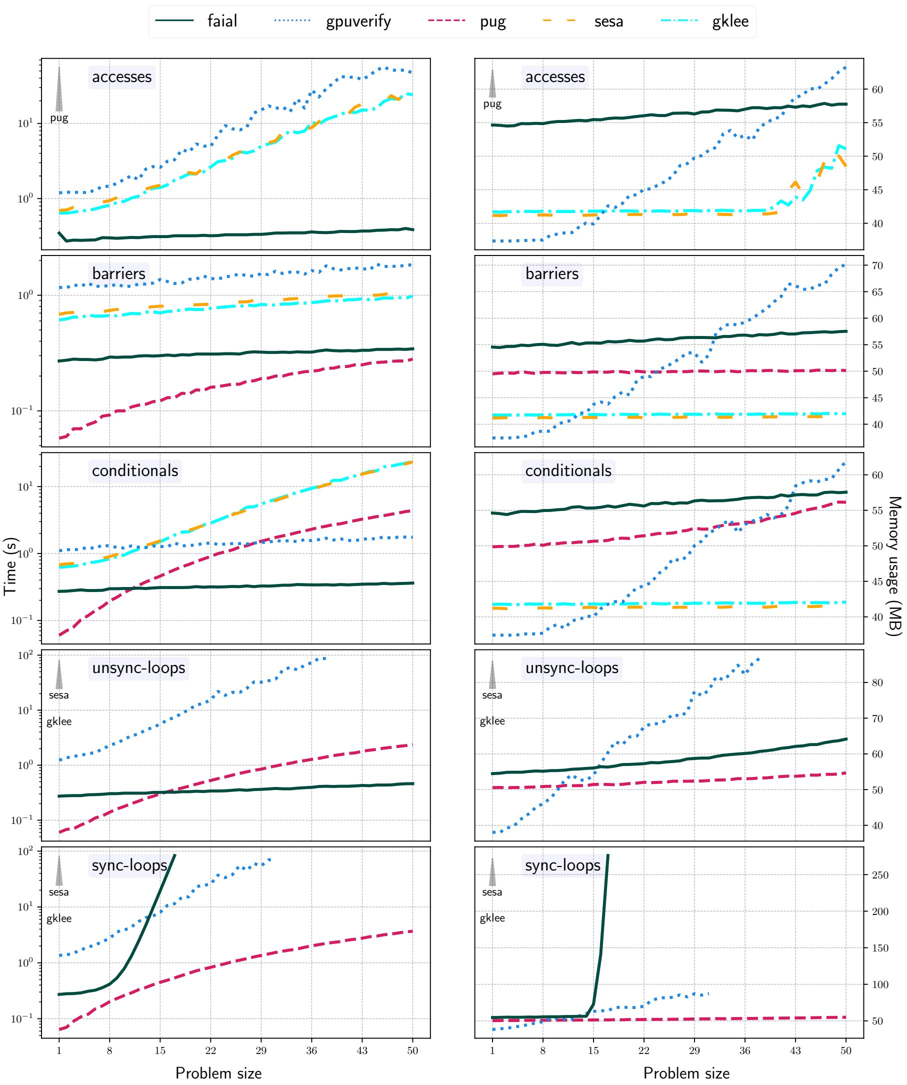


Fig. 18: Results for Claim 2. Run time (left plots) are given on a logarithmic scale, and memory (right plots) are given on a linear scale. Flatter and lower curve is better. Tools annotated with a triangle are excluded due to timeouts or errors.

90s and we exclude any run that times out or reports a false alarm. Cutoffs in the memory plots are determined by the cutoffs in the run time plots.

Overall Faial is the most scalable tool. In 4 out of 5 categories, Faial has the slowest growth for all experiments, and verifies all tests within 0.46s. In the largest problem sizes, our tool is the fastest in 3 categories (access, conditional, unsynchronized loop), 2^{nd} for barriers, and 3^{rd} for synchronized loops. Overall, the memory usage of Faial is competitive with other tools. Faial is the only tool with a near constant time/memory for up to 50 unsynchronized loops, indicating the scalability of reducing unsynchronized loops to universally quantified formulas. Faial only times out for kernels which consists of >17 nested synchronized loops. However such kernels are uncommon, *e.g.*, the levels of nested synchronized loops in the real-word kernels studied in Claim 3 are at most 3.

GPUVerify remains stable in the barrier and conditional categories but is affected negatively by loops and accesses. Loops are a known bottleneck in GPUVerify [29]. In the access category there is an exponential slowdown due to GPUVerify keeping track of what data is being written to/read from array.

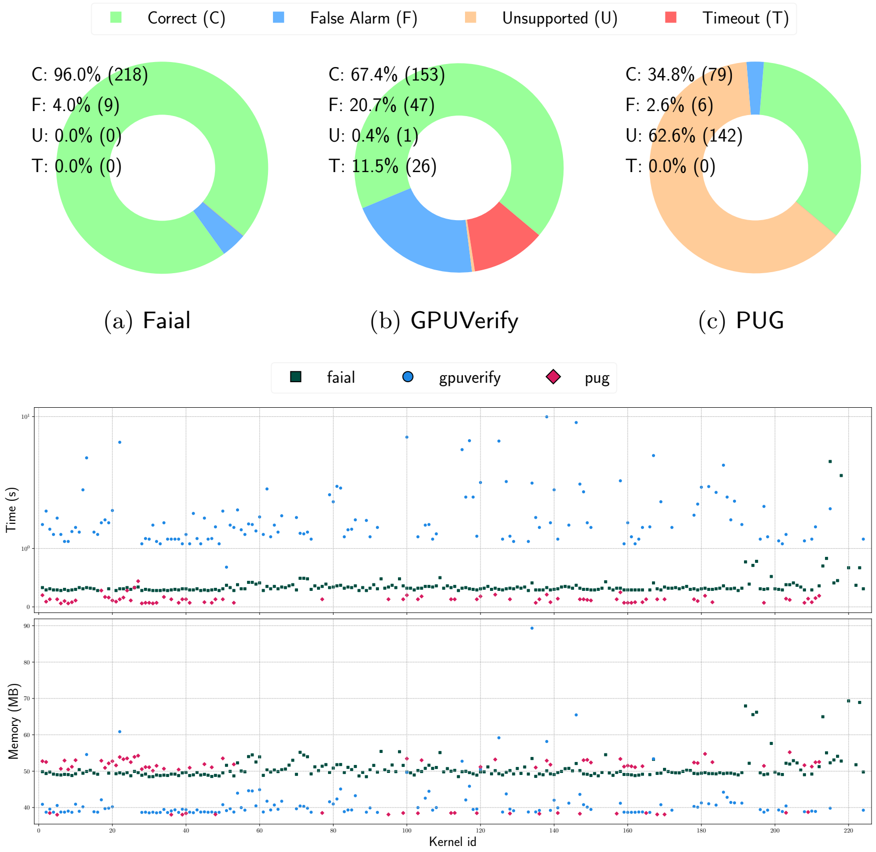
PUG tool remains stable with the number of barrier synchronizations but is affected negatively by the number of conditionals and loops. PUG is the fastest tool with smaller inputs, but it raises false alarms in the access category, hence these measurements are omitted from the corresponding plots.

We discuss GKLEE and SESA together since SESA processes GKLEE's NVCC byte code output by concretizing variables, before passing it to GKLEE itself. There are two main factors that affect negatively these symbolic execution tools: (*i*) the number of loops, since they unroll each loop; and (*ii*) the amount of bookkeeping required to keep track of what is read from/written to memory. Figure 18 shows clear exponential curves for the access and barrier synchronization categories. Observe that these tools timeout immediately in the loop categories.

Claim 3: Real-World Usability

We evaluate the usability of our approach by comparing Faial with other static verification tools (GPUVerify and PUG) on real-world kernels wrt. rate of false alarm, run time and memory. We curated a set of CUDA kernels from [29], which consists of 3 benchmark suites (totaling 227 CUDA kernels): NVIDIA GPU Computing SDK v2.0 (8 CUDA kernels); NVIDIA GPU Computing SDK v5.0 (166 CUDA kernels); Microsoft C++ AMP Sample Projects (20 kernels); gpgpu-sim benchmarks (33 kernels). The gpgpu-sim benchmarks is detailed in [30], and includes kernels from weather simulation [31], sequence alignment [32], AES cryptography [33], among others. All kernels are DRF and have been pre-processed by the authors of [29] to facilitate verification. Each kernel is in a distinct file, all dependencies are available, and kernels are annotated with minimal pre-conditions to allow for automatic analysis (*e.g.*, thread count is given). The authors of [29] do not document how these pre-conditions were obtained.

As we aim to evaluate fully-automatic verification, we removed any code annotation specific to GPUVerify, which included 64 loop invariants. For a detailed discussion on the challenges of generating the invariants included in the original dataset see [29]. The code annotations that remain in the dataset are 251 constraints on parameters, *e.g.*, `dx ==512` or `index_in < width`. Additionally, we made minor changes to some kernels to meet the limitations of the



(a) Faial (b) GPUVerify (c) PUG

(d) Run time (top) and memory usage (bottom) of true-positives. Time (resp. memory) is cropped at 10s (resp. 100MB) and plotted on a logarithmic (resp. linear) scale.

Fig. 19: Results for Claim 3, on a set of 227 DRF CUDA kernels.

front-end of Faial and PUG. For instance we converted nested array lookups to use temporary variables and inlined functions calls that operate on arrays in 22 kernels. Another 8 kernels were modified to simplify their control flows. Our curated dataset is included in our artifact [16].

Figures 19a, 19b, and 19c give the correctness results of Faial, GPUVerify, and PUG, respectively. Correct refers to the true-positive rate, *i.e.*, when the tool correctly identifies the kernel as DRF. False Alarm refers to the false alarm rate, *i.e.*, when the tool incorrectly identifies the kernel as racy. A kernel is Unsupported if it makes the tool crash. A Timeout occurs when the tool exceeds the limit of 60s to verify a kernel. The values shown are an average calculated over five runs. Figure 19d shows the average run time and memory usage of every true-positive report (we omit invalid reports) across the three tools.

Overall Faial has the highest rate of true-positives at 96%. Our tool is second in terms of run time and memory usage, showing a good compromise w.r.t. time and space. Faial verifies most kernels within 1s, and all kernels that need more time are only verified by Faial. GPUVerify shows lower memory usage at the cost of a higher verification run time. PUG verifies the lowest number of kernels (34.8%), as most kernels are unsupported (62.6%).

8 Related Work

SMT-based data-race freedom analyses Li and Gopalakrishnan propose a direct encoding of data-race freedom analysis of GPU programs in SMT, with PUG [13, 24]. Both PUG and Faial follow a similar approach of barrier splitting: having a symbolic representation of a canonical interleaving, and dividing up the analysis over barrier intervals. The two major distinctions are that (1) PUG misses inter-iteration data-races in synchronized loops, *e.g.*, Listing 2.1, and (2) the algorithms of PUG are unspecified and lack soundness proofs. In [13, Section 6.3] the authors identify the challenge of detecting inter-iteration data-races, but do not elaborate a solution. Ma *et al.* [25] present a similar technique to detect data-races and deadlocks in OpenMP programs (CPU-based parallelism). However, their work does not guarantee data-race freedom, and they do not formalize their algorithms. In [34], Prasanth *et al.* propose a polyhedral encoding of DRF for OpenMP programs, which is only applicable to programs with affine array accesses. However the prevalence of linearized array expressions in GPU kernels is known to stump polyhedral analysis [35].

Hoare-logic-based data-race freedom analyses The main drawback of Hoare-logic based tools is their high rate of false alarms. They also require code annotations from a concurrency expert to handle loops. GPUVerify [10, 20, 29, 36, 37] can verify CUDA and OpenCL kernels using Boogie [38] as a backend. GPUVerify also relies on a two-thread abstraction (pen and paper proof) — in this paper, we present the first *machine-checked* proof of the two-thread abstraction idea. VeriCUDA [39, 40] focuses on reasoning about the functional correctness of GPU programs using Hoare-logic. In [11] the authors extend VeriCUDA to guarantee data-race freedom. In a similar vein, VerCors [12] uses separation logic to prove the functional correctness and data-race freedom of GPU kernels. Both VeriCUDA and VerCors expect a tool-specific language, hence cannot handle real-world kernels directly.

Data-race finders We use the term data-race finder to refer to tools that rely on dynamic data-race detection, symbolic-execution, or model-checking. Such techniques are better suited for highly detailed analysis in smaller kernels, and typically are unable to prove data-race freedom. Dynamic data-race detection executes a kernel to find data-races on a fixed input, *e.g.*, [41–47]. This technique only reports real data-races, but suffers from a slowdown of at least 10× compared to the non-instrumented program, and requires the kernel input data, which might be unavailable or unknown. Symbolic execution and model checking have been extended to detect data-races [7, 9, 14, 48, 49]. These techniques do without the kernel input data and can detect more data-races than dynamic data-race detection.

Verification of GPU kernels Ferrel *et al.* introduce a machine-checked formalism to reason about the semantics of CUDA assembly [50]. Muller and Hoffmann present a logic to reason about the evaluation cost of CUDA kernels [51]. Alur *et al.* introduce a formalism to detect uncoalesced accesses in GPU kernels [52, 53].

Miscellaneous Dabrowski *et al.* mechanize the data-race freedom analysis of multithreaded programs [54]. Other behavioral types-based techniques have been used to verify parallel and multithreaded systems that communicate via message-passing [55–57]. However these do not capture shared memory (only message-passing), thus cannot address data-races.

9 Conclusion

We tackle the problem of statically checking data-race freedom in GPU kernels, with a new family of behavioral types, *i.e.*, memory access protocols. We provide a novel compositional analysis of memory access protocols, along with fully mechanized proofs and an implementation. Our evaluation explores challenging and diverse benchmarks (229 real-world and 258 synthetic kernels) to demonstrate that our approach is more precise (false alarms and missed alarms), scalable (time/memory growth), and usable (real-world kernels correctly verified) than other tools.

Future works includes adding support for more synchronization constructs, proposing techniques to prove that some data-race alarms are sound, and surveying GPU programs in the wild. We want to expand our definitions to support atomics, group synchronization (also known as inter-block synchronization), and device synchronization. We anticipate a low impact in the project, as such features appear to only be instantiations of our theory, *c.f.*, [28]. In [58], we have started laying foundations to prove that certain alarms are truthful; our next steps in the theory-front is to continue this line of work. Finally, we want to survey a large dataset of CUDA programs so that we can have more experimental data to justify future research directions.

Acknowledgements We thank Rumyana Neykova, Stephen Chang, and the anonymous reviewers for their insightful feedback on earlier versions of this work. This material is based upon work supported by the National Science Foundation under Grant No. 2204986.

References

[1] Nguyen, G., Dlugolinsky, S., Bobák, M., Tran, V., López García, Á., Heredia, I., Malík, P., Hluchý, L.: Machine Learning and Deep Learning

frameworks and libraries for large-scale data mining: a survey. Artificial Intelligence Review **52**(1), 77–124 (2019). https://doi.org/10.1007/s10462-018-09679-z

[2] Stone, J.E., Hardy, D.J., Ufimtsev, I.S., Schulten, K.: GPU-accelerated molecular modeling coming of age. Journal of Molecular Graphics and Modelling **29**(2), 116–125 (2010). https://doi.org/10.1016/j.jmgm.2010.06.010

[3] Dematté, L., Prandi, D.: GPU computing for systems biology. Briefings in Bioinformatics **11**(3), 323–333 (2010). https://doi.org/10.1093/bib/bbq006

[4] Shi, L., Liu, W., Zhang, H., Xie, Y., Wang, D.: A survey of GPU-based medical image computing techniques. Quantitative imaging in medicine and surgery **2**, 188–206 (2012). https://doi.org/10.3978/j.issn.2223-4292.2012.08.02

[5] Chajan, E., Schulte-Tigges, J., Reke, M., Ferrein, A., Matheis, D., Walter, T.: GPU based model-predictive path control for self-driving vehicles. In: Proceedings of IV, pp. 1243–1248. IEEE, Piscataway, NJ, USA (2021). https://doi.org/10.1109/IV48863.2021.9575619

[6] Kalaiselvi, T., Sriramakrishnan, P., Somasundaram, K.: Survey of using GPU CUDA programming model in medical image analysis. Informatics in Medicine Unlocked **9**, 133–144 (2017). https://doi.org/10.1016/j.imu.2017.08.001

[7] Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: Concolic verification and test generation for GPUs. In: Proceedings of PPoPP, vol. 47, pp. 215–224. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2370036.2145844

[8] Li, P., Li, G., Gopalakrishnan, G.: Practical symbolic race checking of GPU programs. In: Proceedings of SC, pp. 179–190. IEEE, Piscataway, NJ, USA (2014). https://doi.org/10.1109/SC.2014.20

[9] Pereira, P., Albuquerque, H., Marques, H., Silva, I., Carvalho, C., Cordeiro, L., Santos, V., Ferreira, R.: Verifying CUDA programs using SMT-based context-bounded model checking. In: Proceedings of SAC, pp. 1648–1653. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2851613.2851830

[10] Betts, A., Chong, N., Donaldson, A.F., Ketema, J., Qadeer, S., Thomson, P., Wickerson, J.: The design and implementation of a verification technique for GPU kernels. Transactions on Programming Languages and Systems **37**(3), 1–49 (2015). https://doi.org/10.1145/2743017

[11] Kojima, K., Imanishi, A., Igarashi, A.: Automated verification of functional correctness of race-free GPU programs. Journal of Automated Reasoning **60**(3), 279–298 (2018). https://doi.org/10.1007/s10817-017-9428-2

[12] Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Science of Computer Programming **95**(P3), 376–388 (2014). https://doi.org/10.1016/j.scico.2014.03.013

[13] Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of FSE, pp. 187–196. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1882291.1882320

[14] Wu, M., Ouyang, Y., Zhou, H., Zhang, L., Liu, C., Zhang, Y.: Simulee: Detecting CUDA synchronization bugs via memory-access modeling. In: Proceedings of ICSE, pp. 937–948. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380358

[15] Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P.-M., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2-3), 95–230 (2016). https://doi.org/10.1561/2500000031

[16] Cogumbreiro, T., Lange, J., Liew Zhen Rong, D., Zicarelli, H.: Memory Access Protocols: Certified Data-Race Freedom for GPU Kernels (Artifact). GitLab (2021). https://gitlab.com/umb-svl/faial-artifact-journal

[17] Cogumbreiro, T., Lange, J., Rong, D.L.Z., Zicarelli: Checking data-race freedom of GPU kernels, compositionally. In: Proceedings of CAV. LNCS, vol. 12759, pp. 403–426. ACM, New York, NY, USA (2021). https://doi.org/10.1007/978-3-030-81685-8_19

[18] ul Hassan Khan Khan, A., Al-Mouhamed, M., Fatayer, A., Almousa, A., Baqais, A., Assayony, M.: Padding free bank conflict resolution for CUDA-based matrix transpose algorithm. In: Proceedings of SNPD, pp. 1–6. IEEE, Piscataway, NJ, USA (2014). https://doi.org/10.1109/SNPD.2014.6888709

[19] Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in CUDA. NVIDIA CUDA SDK Application Note **18** (2009)

[20] Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of OOPSLA, pp. 113–132. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2384616.2384625

[21] Adve, S.V., Hill, M.D.: Weak ordering — a new definition. In: Proceedings of ISCA, pp. 2–14. ACM, New York, NY, USA (1990). https://doi.org/10.1145/325164.325100

[22] Adve, S.V., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Transactions on Parallel and Distributed Systems **4**(6), 613–624 (1993). https://doi.org/10.1109/71.242161

[23] Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S.: DRFX: A simple and efficient memory model for concurrent programming languages. In: Proceedings of PLDI, pp. 351–362. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1806596.1806636. https://doi.org/10.1145/1806596.1806636

[24] Li, G., Gopalakrishnan, G.: Parameterized verification of GPU kernel programs. In: Proceedings of IPDPSW, pp. 2450–2459. IEEE, Piscataway, NJ, USA (2012). https://doi.org/10.1109/IPDPSW.2012.302

[25] Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis of concurrency errors in OpenMP programs. In: Proceedings of ICPP, pp. 510–516. IEEE, Piscataway, NJ, USA (2013). https://doi.org/10.1109/ICPP.2013.63

[26] Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of CGO, pp. 75–88. IEEE, Piscataway, NJ, USA (2004). https://doi.org/10.1109/CGO.2004.1281665

[27] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of TACAS, pp. 337–340. Springer, Berlin, Heidelberg (2008)

[28] Bardsley, E., Donaldson, A.F.: Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In: Proceedings of NFM, pp. 230–245. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-06200-6_18

[29] Bardsley, E., Betts, A., Chong, N., Collingbourne, P., Deligiannis, P., Donaldson, A.F., Ketema, J., Liew, D., Qadeer, S.: Engineering a static verification tool for GPU kernels. In: Proceedings of CAV, vol. 8559, pp. 226–242. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-319-08867-9_15

[30] Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: Proceedings of ISPASS, pp. 163–174. IEEE, Piscataway, NJ, USA (2009). https://doi.org/10.1109/ISPASS.2009.4919648

[31] Michalakes, J., Vachharajani, M.: GPU acceleration of numerical weather prediction. In: Proceedings of IPDPS, pp. 1–7. IEEE, Piscataway, NJ, USA (2008). https://doi.org/10.1109/IPDPS.2008.4536351

[32] Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney , A.: High-throughput sequence alignment using graphics processing units. BMC Bioinformatics **8**, 474–483 (2007). https://doi.org/10.1186/1471-2105-8-474

[33] Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: Proceedings of ICSPC, pp. 65–68. IEEE, Piscataway, NJ, USA (2007). https://doi.org/10.1109/ICSPC.2007.4728256

[34] Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for SPMD programs and its use in static data race detection. In: Proceedings of LCPC'16, pp. 106–120. Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-319-52709-3_10

[35] Grosser, T., Ramanujam, J., Pouchet, L.-N., Sadayappan, P., Pop, S.: Optimistic delinearization of parametrically sized arrays. In: Proceedings of ICS, pp. 351–360. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2751205.2751248

[36] Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-step semantics for analysis and verification of GPU kernels. In: Proceedings of ESOP, pp. 270–289. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_16

[37] Bardsley, E., Donaldson, A.F., Wickerson, J.: KernelInterceptor: Automating GPU kernel verification by intercepting kernels and their parameters. In: Proceedings of IWOCL, pp. 1–5. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2664666.2664673

[38] Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proceedings of FMCO, pp. 364–387. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11804192_17

[39] Kojima, K., Igarashi, A.: A Hoare logic for SIMT programs. In: Proceedings of APLAS, vol. 8301, pp. 58–73. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-319-03542-0_5

[40] Kojima, K., Igarashi, A.: A Hoare logic for GPU kernels. Transactions on Computational Logic **18**(1), 1–43 (2017). https://doi.org/10.1145/3001834

[41] Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GRace: A low-overhead

mechanism for detecting data races in GPU programs. In: Proceedings of PPoPP, pp. 135–146. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1941553.1941574

[42] Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GMRace: Detecting data races in GPU programs via a low-overhead scheme. Transactions on Parallel and Distributed Systems **25**(1), 104–115 (2014). https://doi.org/10.1109/TPDS.2013.44

[43] Li, P., Hu, X., Chen, D., Brock, J., Luo, H., Zhang, E.Z., Ding, C.: LD: Low-overhead GPU race detection without access monitoring. Transactions on Architecture and Code Optimization **14**(1), 1–25 (2017). https://doi.org/10.1145/3046678

[44] Peng, Y., Grover, V., Devietti, J.: CURD: A dynamic CUDA race detector. In: Proceedings of PLDI, pp. 390–403. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192368

[45] Holey, A., Mekkat, V., Zhai, A.: HAccRG: Hardware-accelerated data race detection in GPUs. In: Proceedings of ICPP, pp. 60–69. IEEE, Piscataway, NJ, USA (2013). https://doi.org/10.1109/ICPP.2013.15

[46] Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA programs. In: Proceedings of PLDI, pp. 126–140. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062342

[47] Kamath, A.K., George, A.A., Basu, A.: ScoRD: A scoped race detector for GPUs. In: Proceedings of ISCA, pp. 1036–1049. IEEE, Piscataway, NJ, USA (2020). https://doi.org/10.1109/ISCA45697.2020.00088

[48] Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic crosschecking of floating-point and SIMD code. In: Proceedings of EuroSys, pp. 315–328. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1966445.1966475

[49] Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: Proceedings of HVC, pp. 203–218. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_18

[50] Ferrell, B., Duan, J., Hamlen, K.W.: CUDA au Coq: A framework for machine-validating GPU assembly programs. In: Proceedings of DATE, pp. 474–479. IEEE, Piscataway, NJ, USA (2019). https://doi.org/10.23919/DATE.2019.8715160

[51] Muller, S.K., Hoffmann, J.: Modeling and analyzing evaluation cost of CUDA kernels. Proceedings of the ACM on Programming Languages

5(POPL) (2021). https://doi.org/10.1145/3434306

[52] Alur, R., Devietti, J., Leija, O.S.N., Singhania, N.: Static detection of uncoalesced accesses in GPU programs. Formal Methods in System Design (2021). https://doi.org/10.1007/s10703-021-00362-8

[53] Alur, R., Devietti, J., Leija, O.S.N., Singhania, N.: GPUDrano: Detecting uncoalesced accesses in GPU programs. In: Proceedings of CAV. LNCS, vol. 10426, pp. 507–525. Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-319-63387-9_25

[54] Dabrowski, F., Pichardie, D.: A certified data race analysis for a Java-like language. In: Proceedings of TPHOL, pp. 212–227. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_16

[55] Vasconcelos, V.T., Ravara, A., Gay, S.: Session types for functional multithreading. In: Proceedings of CONCUR, pp. 497–511. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_32

[56] Vasconcelos, V.T.: Session types for linear multithreaded functional programming. In: Proceedings of PPDP, pp. 1–6. ACM, New York, NY, USA (2009). https://doi.org/10.1145/1599410.1599411

[57] López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: Proceedings of OOPSLA, pp. 280–298. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2814270.2814302

[58] Liew, D., Cogumbreiro, T., Lange, J.: Provable GPU data-races in static race detection. In: Proceedings of PLACES. EPTCS, vol. 356, pp. 36–45. OPA, Waterloo, Australia (2022). https://doi.org/10.4204/EPTCS.356.4