Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels[⋆]

Peter Collingbourne[1⋆⋆] Alastair F. Donaldson[1] Jeroen Ketema[1] Shaz Qadeer[2]

[1] Imperial College London
peter@pcc.me.uk,{afd,jketema}@imperial.ac.uk
[2] Microsoft Research
qadeer@microsoft.com

Abstract. Graphics Processing Units (GPUs) from leading vendors employ predicated (or guarded) execution to eliminate branching and increase performance. Similarly, a recent GPU verification technique uses predication to reduce verification of GPU kernels (the massively parallel programs that run on GPUs) to verification of a sequential program.

Prior work on the formal semantics of lock-step predicated execution for kernels focused on *structured programs*, where control is organised using **if**- and **while**-statements. We provide lock-step execution semantics for GPU kernels that are represented by *arbitrary* reducible control flow graphs. We present a traditional interleaving semantics and a novel lock-step semantics based on predication, and show that for terminating kernels either both semantics compute identical results or both behave erroneously.

The method allows reducing GPU kernel verification to the verification of a sequential, lock-step program to be applied to GPU kernels with arbitrary reducible control flow. We have implemented the method in the GPUVerify tool, and present an evaluation using a set of 163 open source and commercial GPU kernels. Among these kernels, 42 exhibit unstructured control flow which our novel lock-step predication technique can handle fully automatically. This generality comes at a modest price: verification across our benchmark set was on average 2.25 times slower than using an existing approach that specifically targets structured kernels.

1 Introduction

Graphics Processing Units (GPUs) have recently found application in accelerating general-purpose computations, e.g. in image retrieval [24] and machine learning [3]. If an application exhibits significant parallelism it may be possible to extract the computational core of the application as a *kernel* and offload this kernel to run across the parallel hardware of a GPU, sometimes beating CPU performance by orders of magnitude. Writing kernels for massively parallel GPUs is challenging, requiring coordination of a large number of threads. Data races and mis-synchronisation at barriers (known as *barrier divergence*) can lead to erroneous and non-deterministic program behaviours. Worse, they can lead to bugs which manifest *only* on some GPU architectures.

⋆ This work was supported by the EU FP7 STREP project CARP (project number 287767).

⋆⋆ Peter Collingbourne is currently employed at Google.

Recently, substantial effort has been put into the design of tools for rigorous analysis of GPU kernels [7,17,8,18,16]. In prior work [7], we designed a verification technique and tool, GPUVerify, for OpenCL [14] and CUDA [22] kernels. GPUVerify achieves scalability by reducing verification of a parallel kernel to a *sequential* program verification task. This is achieved by transforming the kernel into a form where all threads execute in *lock-step* in a manner that still facilitates detection of data races arising due to arbitrary thread interleavings.

Semantics and program transformations for lock-step execution have been formally studied for *structured* GPU kernels where control flow is described by **if** and **while** constructs [7], but the technique of [7] is limited to the analysis of structured kernels. Lock-step semantics for GPU kernels where control flow is described by an *arbitrary* reducible control flow graph (CFG),[3] has *not* been studied. This is a serious limitation to the design of GPU kernel analysis techniques: kernels frequently exhibit unstructured control flow, either directly, e.g. through the use of **switch**-statements, or indirectly, through short-circuit evaluation of Boolean expressions. Dealing with CFGs also enables analysis of GPU kernels after compiler optimisations have been applied, bringing the analysis closer to the code actually executed by the GPU. It allows for the reuse of existing compiler infrastructures, such as Clang/LLVM, which use CFGs as their intermediate representation. Reusing compiler infrastructures hugely simplifies tool development, removing the burden of writing a robust front-end for C-like languages.

We present a traditional interleaving semantics and a novel lock-step semantics for GPU kernels described by CFGs. We show that if a GPU kernel is guaranteed to terminate then the kernel is correct with respect to the interleaving semantics if and only if it is correct with respect to the lock-step semantics, where *correct* means that all execution traces are free from data races, barrier divergence, and assertion failures. Our novel lock-step semantics enables the strategy of reducing verification of a multithreaded GPU kernel to verification of a sequential program to be applied to arbitrary GPU kernels, and we have implemented this method in the GPUVerify tool. We present an experimental evaluation, applying our new tool to a set of 163 open source and commercial GPU kernels. In 42 cases these kernels exhibited unstructured control flow either explicitly (e.g. through **switch**-statements), or implicitly due to short-circuit evaluation. In the former case, these kernels had to be manually simplified to be amenable to analysis using the original version of GPUVerify. In the latter case, it turned out that the semantics of short-circuit evaluation of logical operators was not handled correctly in GPUVerify. Our new, more general implementation handles all these kernels accurately and automatically. Our results show that overall GPUVerify continues to perform well, with verification across our benchmark set being on average only 2.25 times slower compared to the original version that was limited to structured kernels [7].

In summary, our main contributions are:

- A novel operational semantics for lock-step execution of GPU kernels with arbitrary reducible control flow.

[3] Henceforth, whenever we refer to a CFG we shall always mean a reducible CFG. For a definition of reducibility we refer the reader to [1]. We note that irreducibility is uncommon in practice. In particular, we have never encountered a GPU kernel with an irreducible control flow graph, and whether irreducible control flow is supported at all is implementation-defined in OpenCL [14].

```
__kernel void
scan(__global int *sum) {
 int offset = 1, temp;
 while (offset < TS) {
  if (tid >= offset)
   temp = sum[tid - offset];
  barrier();
  if (tid >= offset)
   sum[tid] = sum[tid] + temp;
  barrier();
  offset *= 2;
 }
}
```

(a) A correct kernel

```
__kernel void
scan(__global int *sum) {
 int offset = 1, temp;
 while (offset <= tid) {
  temp = sum[tid - offset];
  barrier();
  sum[tid] = sum[tid] + temp;
  barrier();
  offset *= 2;
 }
}
```

(b) A kernel with barrier divergence

Fig. 1: Two OpenCL kernels

– A proof-sketch that this semantics is equivalent to a traditional interleaving semantics for terminating GPU kernels.
– A verification method for GPU kernels described as CFGs, which uses our lock-step semantics to reduce verification of a multithreaded kernel to a sequential verification task, implemented in the GPUVerify tool.
– An experimental evaluation of this implementation with respect to a large collection of benchmarks.

After presenting a small example to provide some background on GPU kernels and illustrate the problems of data races and barrier divergence (Sect. 2), we present the interleaving semantics (Sect. 3), our novel lock-step semantics (Sect. 4) and a proof-sketch showing that the semantics are equivalent for terminating kernels (Sect. 5). We then discuss the implementation in GPUVerify, and present our experimental results (Sect. 6). We end with related work and conclusions (Sect. 7).

2 A Background Example

We use an example to illustrate the key concepts from GPU programming and provide an informal description as to how predicated lock-step execution works for structured programs. We return to this example when presenting interleaving and lock-step semantics for kernels described as CFGs in Sects. 3 and 4.

Threads, Barriers, and Shared Memory. Figure 1a shows an OpenCL kernel[4] to be executed by TS threads, where TS is a power of two. The kernel computes a *scan* operation on the `sum` array so that at the end of the kernel we have, for all $0 \leq i < TS$, $\texttt{sum}[i] = \Sigma_{j=0}^{i} \mathbf{old}(\texttt{sum})[i]$, where $\mathbf{old}(\texttt{sum})$ refers to the `sum` array at the start of the kernel. All threads execute this kernel function in parallel, and threads may follow different control paths or access distinct data by querying their unique thread id, `tid`.

[4] For ease of presentation we use a slightly simplified version of OpenCL syntax, and we assume that all threads reside in the same work group and that this work group is one dimensional. Our implementation, described in Sect. 6, supports OpenCL in full.

Communication is possible via shared memory; the `sum` array is marked as residing in global shared memory via the `__global` qualifier. Threads synchronise using a `barrier`-statement, a collective operation that requires all threads to reach the same syntactic barrier before any thread proceeds past the barrier.

Data Races and Barrier Divergence. Two common defects from which GPU kernels suffer are *data races* and *barrier divergence*. Accesses to `sum` inside the loop are guarded so that on loop iteration i only threads with id at least 2^{i-1} access the `sum` array. If either of the barriers in the example were omitted the kernel would be prone to a *data race* arising due to thread t_1 reading from $\texttt{sum}[t_1 - \texttt{offset}]$, while thread t_2 writes to $\texttt{sum}[t_2]$, where $t_2 = t_1 - \texttt{offset}$. The kernel of Fig. 1b aims to optimise the original example by reducing branches inside the loop: threads are restricted to only execute the loop body if their id is sufficiently large. This optimisation is *erroneous*; given a barrier inside a loop, the OpenCL standard requires that either all threads or zero threads reach the barrier during a given loop iteration, otherwise *barrier divergence* occurs and behaviour is undefined. In Fig. 1b, thread 0 will not enter the loop at all and thus will *never* reach the first barrier, while all other threads will enter the loop and reach the barrier. Unfortunately, on mainstream GPU architectures from AMD and NVIDIA the kernel of Fig. 1b behaves identically to the kernel of Fig. 1a, meaning that this barrier divergence bug was not detected during testing. This is problematic because the erroneous kernel code is not portable across architectures which support OpenCL.

Lock-Step Predicated Execution. We informally describe lock-step execution for structured programs as used by GPUVerify [7] and which we here generalise to CFGs.

To achieve lock-step execution, GPUVerify transforms kernels into *predicated* form [2]. The example of Fig. 2 illustrates the effect of applying predication to the example of Fig. 1a. A statement of the form $e \Rightarrow \mathit{stmt}$ is a *predicated* statement which is a no-op if e is false, and has the same effect as *stmt* if e is true. Observe that the **if**-statements in the body of the loop have been predicated: the condition (which is the same for both statements) is evaluated into a Boolean variable q, the conditional statement is removed and the statements previously inside the conditional are predicated by the associated Boolean variable. Lock-step execution of the **while**-loop is achieved by evaluating the loop condition into a Boolean variable p, predicating all statements in the loop body by p, and recomputing p at the end of the loop body. The loop condition is replaced by a guard which evaluates to false if and only if the predicate variable p is false for *every* thread. Thus all threads continue to execute the loop until every thread is ready to leave the loop; when the loop condition becomes false for a given thread the thread simply performs no-ops during subsequent loop iterations.

```
__kernel void
scan(__global int *sum) {
  bool p, q;
  int offset = 1, temp;
  p = (offset < TS);
  while (∃ t :: t.p) {
    q = (p && tid >= offset);
    q ⇒ temp = sum[tid - offset];
    p ⇒ barrier();
    q ⇒ sum[tid] = sum[tid] + temp;
    p ⇒ barrier();
    p ⇒ offset *= 2;
    p ⇒ p = (offset < TS);
  }
}
```

Fig. 2: Lock-step predicated execution for structured kernel of Fig. 1a

In predicated form, the kernel does not exhibit any thread interleavings, and thus can be regarded as a *sequential*, vector program. GPUVerify exploits this fact to reduce

GPU kernel verification to a sequential program verification task. The full technique, described in [7], involves considering lock-step execution of an arbitrary *pair* of threads, rather than all threads, and employs instrumentation variables to efficiently check for data races that could manifest due to arbitrary thread interleavings.

Predication is easy to perform at the level of structured programs built using **if**- and **while**-statements. However, as discussed in the introduction, GPU kernels in general may exhibit unstructured control flow. In Sect. 4 we present a program transformation for predicated execution of GPU kernels that are described as CFGs.

3 Interleaving Semantics for GPU Kernels

We introduce a simple language for describing GPU kernels as CFGs, and present an interleaving semantics for this language. We do not include procedures in our presentation due to space reasons, however our results do extend to a procedural setting and procedures are supported by the implementation described in Sect. 6.

3.1 Syntax

The syntax for our language is identical to the core of the Boogie programming language [5], except that it includes an additional **barrier** statement:

$$\begin{aligned}
\mathit{Program} &::= \mathit{Block}^{+} \\
\mathit{Block} &::= \mathit{BlockId} : \mathit{Stmts}\ \mathbf{goto}\ \mathit{BlockId}^{+}\ ; \\
\mathit{Stmts} &::= \varepsilon \mid \mathit{Stmt} ; \mathit{Stmts} \\
\mathit{Stmt} &::= \mathit{Var} := \mathit{Expr} \mid \mathbf{havoc}\ \mathit{Var} \mid \mathbf{assume}\ \mathit{Expr} \mid \mathbf{assert}\ \mathit{Expr} \mid \mathbf{skip} \mid \mathbf{barrier}
\end{aligned}$$

Here, ε is an empty sequence of statements. The form of expressions is mostly irrelevant. We do assume presence of (a) *equality testing* $(=)$, (b) the standard Boolean operators, and (c) a ternary operator $\mathit{Expr}_1 \mathbin{?} \mathit{Expr}_2 : \mathit{Expr}_3$, which — like the operator from C — evaluates Expr_2 in case Expr_1 is true and evaluates Expr_3 in case Expr_1 is false.

To summarise in words, a kernel consists of a number of *basic blocks*, with each block consisting of a number of *statements* followed by a **goto** that non-deterministically chooses which block to execute next based on the provided $\mathit{BlockId}$s; non-deterministic choice in combination with **assume**s is used to model branches.

Because gotos only appear at the end of blocks there is a one-to-one correspondence between kernels and CFGs. We assume that all kernels have reducible CFGs, which means that cycles in a CFG are guaranteed to form *natural loops*. A natural loop has a unique *header* node, the single entry point to the loop, and one or more *back edges* going from a loop node to the header [1].

We assume that each kernel has at least a block labelled Start. This is the block from which execution of each thread commences. Moreover, no block is labelled End; instead the occurrence of End in a **goto** signifies that the program may terminate at this point.

Figure 3 shows the kernel of Fig. 1a encoded in our simple programming language. The example uses an array; we could easily include arrays in our formalisation of GPU kernel semantics but, for brevity, we do not.

```
$Start$    : $offset := 1$ ;
             goto $W, W_{end}$ ;
$W$        : assume $offset < TS$ ;
             goto $I_1, I_1'$ ;
$I_1$      : assume $tid \geq offset$ ;
             $temp := sum[tid + offset]$ ;
             goto $B_1$ ;
$I_1'$     : assume $tid < offset$ ;
             goto $B_1$ ;
$B_1$      : barrier ;
             goto $I_2, I_2'$ ;
$I_2$      : assume $tid \geq offset$ ;
             $sum[tid] := sum[tid] + temp$ ;
             goto $B_2$ ;
$I_2'$     : assume $tid < offset$ ;
             goto $B_2$ ;
$B_2$      : barrier ;
             goto $W_{last}$ ;
$W_{last}$ : $offset := 2 \cdot offset$ ;
             goto $W, W_{end}$ ;
$W_{end}$  : assume $offset \geq TS$ ;
             goto $End$ ;
```

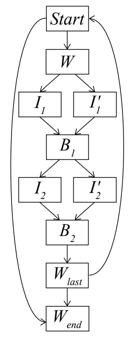

Fig. 3: The kernel of Fig. 1a encoded in our kernel language and its CFG

3.2 Operational Semantics

We now define a small-step operational semantics for our kernel programming language, which is based on interleaving the steps taken by individual threads.

Individual Threads. The behaviour of individual threads and non-barrier statements executed by these threads is presented in Figs. 4a and 4b.

The operational semantics of a thread t is defined in terms of triples $\langle \sigma, \sigma_t, b_t \rangle$, where σ is the GPU *shared* store, σ_t is the *private* store for thread t and b_t is the statement or sequence of statements the thread will execute, or more formally will *reduce*, next. Each store is a mapping from variables to the values in some domain D. We assume that *no* variable is mapped to a value in both σ and σ_t.

In Fig. 4a, $(\sigma, \sigma_t)[v \mapsto val]$ denotes a pair of stores equal to (σ, σ_t) except that v (which we assume occurs in either σ or σ_t) has been updated and is equal to val. In the same figure, $(\sigma, \sigma_t)(e)$ denotes the evaluation of the expression e given (σ, σ_t). The labels on the arrows allow us to observe (a) changes to stores and (b) the state of stores upon. A label is omitted when the stores do not change, e.g. in the SKIP-rule.

The symbols $\surd$, $\mathcal{E}$, and $\perp$ indicate, resp., *termination*, *error*, and *infeasible*. These are *termination statuses* and signify that a thread (or later kernel) has terminated with that particular status. Below, termination always means termination with status *termination*; termination with status *error* or *infeasible* is indicated explicitly.

The ASSIGN- and SKIP-rules of Fig. 4a are standard. The HAVOC-rule updates the value of a variable v with an arbitrary value from the domain D of v. The ASSERT$_{\mathrm{T}}$ and ASSUME$_{\mathrm{T}}$-rules are no-ops if the assumption or assertion $(\sigma, \sigma_t)(e)$ holds. If the assumption or assertion does not holds, ASSERT$_{\mathrm{F}}$ and ASSUME$_{\mathrm{F}}$ yield, resp., $\mathcal{E}$ and $\perp$.

In Fig. 4b, s denotes a statement and b denotes the *body of a block*, i.e. a sequence of statements followed by a **goto**. The SEQ$_{\mathrm{B}}$- and SEQ$_{\mathrm{E,I}}$-rules define reduction of $s\,;\,b$ in terms of reduction of s. The GOTO- and BLOCK-rules specify how reduction continues once the end of a block is reached. The END-rule specifies termination of a thread.

Interleaving. We give interleaving semantics for a kernel P with respect to a given thread count TS in Fig. 4c. This is defined in terms of tuples $\langle \sigma, \langle \sigma_1, b_1 \rangle, \ldots, \langle \sigma_{TS}, b_{TS} \rangle \rangle$,

$$\frac{val = (\sigma,\sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, v := e\rangle \overset{(\sigma,\sigma_t)}{\to} (\sigma,\sigma_t)[v \mapsto val]}\ \textsc{Assign}$$

$$\frac{val \in D}{P \vdash \langle \sigma, \sigma_t, \mathbf{havoc}\, v\rangle \overset{(\sigma,\sigma_t)}{\to} (\sigma,\sigma_t)[v \mapsto val]}\ \textsc{Havoc}$$

$$\frac{a \in \{\mathbf{assert}, \mathbf{assume}\} \qquad (\sigma,\sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, a\, e\rangle \to (\sigma,\sigma_t)}\ \begin{matrix}\textsc{Assert}_{\mathrm{T}}\\ \textsc{Assume}_{\mathrm{T}}\end{matrix} \qquad \frac{\neg(\sigma,\sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, \mathbf{assume}\, e\rangle \overset{(\sigma,\sigma_t)}{\to} \bot}\ \textsc{Assume}_{\mathrm{F}}$$

$$\frac{\neg(\sigma,\sigma_t)(e)}{P \vdash \langle \sigma, \sigma_t, \mathbf{assert}\, e\rangle \overset{(\sigma,\sigma_t)}{\to} \mathcal{E}}\ \textsc{Assert}_{\mathrm{F}} \qquad \frac{}{P \vdash \langle \sigma, \sigma_t, \mathbf{skip}\rangle \to (\sigma,\sigma_t)}\ \textsc{Skip}$$

(a) Statement rules

$$\frac{P \vdash \langle \sigma, \sigma_t, s\rangle \overset{(\sigma,\sigma_t)}{\to} (\tau,\tau_t)}{P \vdash \langle \sigma, \sigma_t, s\ ;\ b\rangle \overset{(\sigma,\sigma_t)}{\to} \langle \tau, \tau_t, b\rangle}\ \textsc{Seq}_{\mathrm{B}} \qquad \frac{P \vdash \langle \sigma, \sigma_t, s\rangle \overset{(\sigma,\sigma_t)}{\to} e \qquad e \in \{\mathcal{E}, \bot\}}{P \vdash \langle \sigma, \sigma_t, s\ ;\ b\rangle \overset{(\sigma,\sigma_t)}{\to} e}\ \textsc{Seq}_{\mathrm{E,I}}$$

$$\frac{1 \leq i \leq n}{P \vdash \langle \sigma, \sigma_t, \mathbf{goto}\, B_1, \ldots, B_n\, ;\rangle \to \langle \sigma, \sigma_t, B_i\rangle}\ \textsc{Goto} \qquad \frac{(B : b) \in P}{P \vdash \langle \sigma, \sigma_t, B\rangle \to \langle \sigma, \sigma_t, b\rangle}\ \textsc{Block}$$

$$\frac{}{P \vdash \langle \sigma, \sigma_t, End\rangle \overset{\sigma,\sigma_t}{\to} \surd}\ \textsc{End}$$

(b) Thread rules

$$\frac{T_{\vec{\sigma}}|_t = \langle \sigma_t, b_t\rangle \qquad P \vdash \langle \sigma, \sigma_t, b_t\rangle \overset{(\sigma,\sigma_t)}{\to} \langle \tau, \tau_t, c_t\rangle}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \overset{(\sigma,\vec{\sigma})}{\to} \langle \tau, T_{\vec{\sigma}}[\langle \tau_t, c_t\rangle]_t\rangle}\ \textsc{Thread}_{\mathrm{B}}$$

$$\frac{T_{\vec{\sigma}}|_t = \langle \sigma_t, b_t\rangle \qquad P \vdash \langle \sigma, \sigma_t, b_t\rangle \overset{(\sigma,\sigma_t)}{\to} \surd}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \overset{(\sigma,\vec{\sigma})}{\to} \langle \sigma, T_{\vec{\sigma}}[\langle \sigma_t, \surd\rangle]_t\rangle}\ \textsc{Thread}_{\mathrm{T}}$$

$$\frac{T_{\vec{\sigma}}|_t = \langle \sigma_t, b_t\rangle \qquad P \vdash \langle \sigma, \sigma_t, b_t\rangle \overset{(\sigma,\sigma_t)}{\to} s \qquad s \in \{\mathcal{E}, \bot\}}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \overset{(\sigma,\vec{\sigma})}{\to} s}\ \textsc{Thread}_{\mathrm{E,I}}$$

$$\frac{\forall\, 1 \leq t \leq TS : T_{\vec{\sigma}}|_t = \langle \sigma_t, \surd\rangle}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \overset{(\sigma,\vec{\sigma})}{\to} \surd}\ \textsc{Termination}$$

(c) Interleaving rules

$$\frac{T_{\vec{\sigma}}|_t = \langle \beta_t, \sigma_t, \mathbf{barrier}\, e_t\ ;\ b_t\rangle \wedge \neg(\sigma,\sigma_t)(e_t)}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \to \langle \sigma, T_{\vec{\sigma}}[\langle \beta_t, \sigma_t, b_t\rangle]_t\rangle}\ \textsc{Barrier}_{\textsc{Skip}}$$

$$\frac{\forall\, t : T_{\vec{\sigma}}|_t = \langle \beta, \sigma_t, \mathbf{barrier}\, e_t\ ;\ b_t\rangle \wedge (\sigma,\sigma_t)(e_t)}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \to \langle \sigma, \langle \beta, \sigma_1, b_1\rangle, \ldots, \langle \beta, \sigma_{TS}, b_{TS}\rangle\rangle}\ \textsc{Barrier}_{\textsc{S}}$$

$$\frac{\forall\, t : T_{\vec{\sigma}}|_t = \langle \beta_t, \sigma_t, \mathbf{barrier}\, e_t\ ;\ b_t\rangle \wedge (\sigma,\sigma_t)(e_t) \qquad \exists\, t_1, t_2 : T_{\vec{\sigma}}|_{t_1 \cdot \mathrm{bv}} \neq T_{\vec{\sigma}}|_{t_2 \cdot \mathrm{bv}}}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \overset{(\sigma,\vec{\sigma})}{\to} \mathcal{E}}\ \textsc{Barrier}_{\mathrm{F}}$$

(d) Barrier synchronisation rules

Fig. 4: Interleaving operational semantics

where σ is the *shared store*, σ_t is the *private store* of thread t, and b_t is the program fragment thread t will reduce next. The private store of a thread is not accessible by any other thread. In the figure, $T_{\vec{\sigma}}$ is shorthand for $(\langle\sigma_1, b_1\rangle, \ldots, \langle\sigma_{TS}, b_{TS}\rangle)$, where $\vec{\sigma} = (\sigma_1, \ldots, \sigma_{TS})$. Moreover, $T_{\vec{\sigma}}|_t$ denotes $\langle\sigma_t, b_t\rangle$ and $T_{\vec{\sigma}}[\langle\sigma', b\rangle]_t$ denotes $T_{\vec{\sigma}}$ with the tth element replaced by $\langle\sigma', b\rangle$.

The THREAD$_\text{B}$-rule defines how a single step is performed by a single thread, cf. the rules in Fig. 4b. The THREAD$_\text{T}$-rule defines termination of a single thread, where the thread enters the termination state $\surd$ from which no further reduction is possible. The THREAD$_\text{E,I}$-rule specifies that a kernel should terminate with status *error* or *infeasible* in case one of the threads terminates as such. The TERMINATION-rule specifies that a kernel terminates when all threads have terminated.

The THREAD rules are non-deterministic and define an interleaving semantics, as a step might be possible in multiple threads. A thread *cannot* access the private store of any other thread, while the shared store is accessible by *all* threads.

Given stores $\sigma, \sigma_1, \ldots, \sigma_{TS}$, we define a *reduction* of a kernel P with threads $1 \leq t \leq TS$ as sequence of applications of the operational rules where each threads starts reduction from $Start$ and where the *initial* shared store is σ and the *initial* private store of thread t is σ_t. A reduction is *maximal* if it is either infinite or if termination with status *termination*, *error*, or *infeasible* has occurred.

Our interleaving semantics effectively has a sequentially consistent memory model, which is not the case for GPUs in practice. However, because our viewpoint is that GPU kernels that exhibit data races should be regarded as erroneous, this is of no consequence.

Barrier Synchronisation. When we define lock-step predicated execution of barriers in Sect. 4 we will need to model execution of a barrier by a thread in a *disabled* state. In preparation for this, let us say that barrier statement has the form $\mathbf{barrier}\, e$, with e a Boolean expression. In Sect. 4, e will evaluate to $true$ if and only if the barrier is executed in an enabled state. The notion of thread-enabledness is not relevant to our interleaving semantics: we can view a thread as *always* being enabled. Thus we regard the $\mathbf{barrier}$ syntax of our kernel programming language as short for $\mathbf{barrier}\ true$.

Figure 4d defines rules for (mis-)synchronisation between threads at barriers. Our aim here is to formalise the conditions for correct barrier synchronisation in OpenCL, which are stated informally in the OpenCL specification as follows [14]:

B1 If $\mathbf{barrier}$ is inside a conditional statement, then all [threads] must enter the conditional if any [thread] enters the conditional statement and executes the barrier.
B2 If $\mathbf{barrier}$ is inside a loop, all [threads] must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.

The rules of Fig. 4d capture these conditions using a number of special *barrier variables* that we assume are implicit in definition of each kernel:

- For every loop L in the kernel, every thread has a private *loop counter* variable v_L. The variable v_L of each thread t is initialised to zero, incremented each time the header node for L is reduced by t, and reset to zero on exit from L.
- Every thread has a private variable v_{barrier}. We assume that each barrier appearing in the kernel has a unique id. The variable v_{barrier} of each thread t is initialised to a

special value $(-)$ different from every barrier id. When t reaches a barrier, v_{barrier} is set to the id of that barrier, and it is reset to $(-)$ after reduction of the barrier.

The variable v_{barrier} codifies that each thread is synchronising on the same barrier, capturing condition ***B1*** above. The loop counters codify that each thread must have executed the same number of loop iterations upon synchronisation, capturing ***B2***.

In Fig. 4d, we write (β_t, σ_t), in case we want to make explicit the barrier variables β_t of each thread t. We write $T_{\vec{\sigma}}|_{t \cdot \text{bv}} = \beta_t$ where $T_{\vec{\sigma}}|_t = \langle \beta_t, \sigma_t, b_t \rangle$.

The $\text{B}{\small\text{ARRIER}}_{\text{SKIP}}$-rule specifies that $\mathbf{barrier}\, e$ is a no-op if e is *false*. Although this can never occur for kernels written directly in our kernel programming language, our equivalence proof in Sect. 5 requires this detail to be accounted for.

The $\text{B}{\small\text{ARRIER}}_{\text{S}}$-rule specifies that reduction continues beyond a barrier if all threads are at a barrier and the barrier variables agree across threads. The $\text{B}{\small\text{ARRIER}}_{\text{F}}$-rule specifies that a kernel should terminate with *error* in case the threads have reached barriers with disagreeing barrier variables, i.e. when *barrier divergence* has occurred.

Data Races. We say that a thread t is *responsible* for a step in a reduction if a THREAD rule (see Fig. 4c) was employed in the step and the premise of the rule was instantiated with t. Moreover, we say that a thread t *accesses* a variable v in a step if t is responsible for the step and if in the step either (a) the value of v is used to evaluate an expression or (b) v is updated. The definition is now as follows:

Definition 3.1. *If P is a kernel, then P* has data race *if there is a maximal reduction ρ of P not ending in the* infeasible *status $\perp$ and a shared variable v such that v is accessed by distinct threads t and t' during ρ, where at least one of the threads updates the variable and where no application of* $\text{B}{\small\text{ARRIER}}_{\text{S}}$ *occurs between accesses.*

Terminating and Race Free Kernels. We say that a kernel P is *terminating* with respect to the interleaving semantics if all maximal reductions of P are finite and do not end with status *error*. We say that P is *race free* with respect to the interleaving semantics if P has no data races according to Definition 3.1.

4 Lock-Step Semantics for GPU Kernels

We define lock-step execution semantics for GPU kernels represented as arbitrary CFGs in two stages. First, in Sect. 4.1, we present a program transformation which turns the sequential program executed by a single thread into a predicated form where control flow is *flattened*: all branches, except for loop back edges, are eliminated. Then, in Sect. 4.2, we use this transformation to express lock-step execution of all threads in a kernel as a sequential program in *vector* form. Each statement of this sequential program will perform the work of all threads in a single step.

To avoid many corner cases we assume that kernels always synchronise on a barrier immediately preceding termination. Moreover, if a block B ends with $\mathbf{goto}\, B_1, \ldots, B_n$ then at most one of $B_1, \ldots, B_n$ is a loop head. A kernel can be trivially preprocessed to satisfy these restrictions.

Sort Order. Predication of CFGs involves flattening control flow, rewriting branches by predicating blocks and executing these blocks in a linear order. For CFGs without loops any topological sort gives a suitable order: it ensures that if block B is a predecessor of C in the original CFG then B will appear before C in the predicated CFG. In the presence of loops the order must ensure that once execution of the blocks in a loop commences this loop will be executed completely before any node outside the loop is executed.

Formally, we require a total order $\leq$ on blocks satisfying the following conditions:

– For all blocks B and C, if there is a path from B to C in the CFG, then $B \leq C$ unless a back edge occurs on the path.
– For all loops L, if $B \leq D$ and $B, D \in L$, then $C \in L$ for all $B \leq C \leq D$.

A total order satisfying the above conditions always exists and can be computed as follows. Consider any innermost loop of the kernel and perform a topological sort of the blocks in the loop body (disregarding back edges). Replace the loop body by an abstract block. Repeat until no loops remain and perform a topological sort of resulting CFG. The sort order is now the order obtained by the final topological sort where we recursively replace each abstract node by the nodes it represents, i.e., if $B \leq L \leq D$ with L an abstract node, then for any $C \leq C'$ in the loop body represented by L we define $B \leq C \leq C' \leq D$. End can always be sorted last, as no **goto** occurs in it.

Considering the kernel of Fig. 3, we have that $L = \{W, I_1, I_1', B_1, I_2, I_2', B_2, W_{last}\}$ is a loop and that $Start \leq W \leq I_1 \leq I_1' \leq B_1 \leq I_2 \leq I_2' \leq B_2 \leq W_{last} \leq W_{end}$ satisfies our requirements; reversing I_1 and I_1' or I_2 and I_2' is possible.

In what follows we assume that a total order satisfying the above conditions has been chosen, and refer to this as the *sort order*. For a block B we use $next(B)$ to denote the block that follows B in the sort order. If B is the final block in the sort order we define $next(B)$ to be End, the block id denoting thread termination.

4.1 Predication of a Single Thread

We now describe how predication of the body of a kernel thread is performed.

Predication of Statements. To predicate statements, we introduce a fresh private variable v_{active} for each thread, to which we assign $BlockId$s; the assigned $BlockId$ indicates the block that needs to be executed. If the value of v_{active} is not equal to the block that is currently being executed, all statements in the block will effectively be no-ops. In the case of **barrier** this follows by the BARRIER$_{\text{SKIP}}$-rule.

Assuming the $BlockId$ of the current block is B, predication of statements is defined in Table 1. In the case of **havoc**, the variable v_{havoc} is fresh and private.

Original form	Predicated form
$v := e\,;$	$v := (v_{\text{active}} = B)\ ?\ e : v\,;$
havoc $v\,;$	**havoc** $v_{\text{havoc}}\,;$ $v := (v_{\text{active}} = B)\ ?\ v_{\text{havoc}} : v\,;$
assert $e\,;$	**assert** $(v_{\text{active}} = B) \Rightarrow e\,;$
assume $e\,;$	**assume** $(v_{\text{active}} = B) \Rightarrow e\,;$
skip ;	**skip** ;
barrier ;	**barrier** $(v_{\text{active}} = B)\,;$

Table 1: Predication of statements

Predication of Blocks. Denoting by $\pi(s)$ the predication of a single statement s, and applying π to sequences of statements in a pointwise fashion. Predication of blocks is now defined by default using the top row of Table 2, where v_{next} is a fresh, private

Original form	Predicated form
$B\ :\ ss$ $\quad\textbf{goto}\, B_1, \ldots, B_n\,;$ (B *is not* the last node of a loop according to the sort order)	$B\ :\ \pi(ss)$ $\quad v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,;$ $\quad v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$ $\quad\textbf{goto}\, \mathit{next}(B)\,;$
$B\ :\ ss$ $\quad\textbf{goto}\, B_1, \ldots, B_n\,;$ (B *is* the last node of a loop according to the sort order)	$B\ :\ \pi(ss)$ $\quad v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,;$ $\quad v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$ $\quad\textbf{goto}\, B_{\text{back}}, B_{\text{exit}}\,;$ $B_{\text{back}}\ :\ \textbf{assume}\, v_{\text{active}} = B_{\text{head}}\,;$ $\quad\textbf{goto}\, B_{\text{head}}\,;$ $B_{\text{exit}}\ :\ \textbf{assume}\, v_{\text{active}} \neq B_{\text{head}}\,;$ $\quad\textbf{goto}\, \mathit{next}(B)\,;$

Table 2: Predication blocks

variable, and $:\in$ is shorthand for $\textbf{havoc}\, v_{\text{next}}\,; \textbf{assume} \bigvee_{i=1}^{n}(v_{\text{next}} = B_i)$. Effectively, v_{active} is set to the value of the block that should be reduced next, while actual reduction will continue according to the sort order with block $next(B)$.

This method of predicating blocks does not deal correctly with loops: no block can be executed more than once as no back-edges are introduced. To take care of this, we predicate block B in a special manner if B belongs to a loop L and B occurs last in the sort order among all the blocks of L. Assume B_{head} is the head of L. The block B is predicated as in the bottom row of Table 2, where B_{back} and B_{exit} are fresh (see also Fig. 5). Our definition of a sort order guarantees that B_{head} is always sorted first among the blocks of L. By the introduction of B_{back}, reduction jumps back to B_{head} if L needs to be reduced again, otherwise reduction will continue beyond L by definition of B_{exit}.

```
$Start$        : $v_{\text{active}} := Start\,;$
                 $offset := (v_{\text{active}} = Start)\ ?\ 1 : offset\,;$
                 $v_{\text{next}} :\in \{W, W_{exit}\}\,;$
                 $v_{\text{active}} := (v_{\text{active}} = Start)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$
                 $\textbf{goto}\, W\,;$
                 $\vdots$
$B_2$          : $\textbf{barrier}\,(v_{\text{active}} = B_2)\,;$
                 $v_{\text{next}} :\in \{W_{last}\}\,;$
                 $v_{\text{active}} := (v_{\text{active}} = B_2)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$
                 $\textbf{goto}\, W_{last}\,;$
$W_{last}$     : $offset :=$
                     $(v_{\text{active}} = W_{end})\ ?\ (2 \cdot offset) : offset\,;$
                 $v_{\text{next}} :\in \{W, W_{exit}\}\,;$
                 $v_{\text{active}} := (v_{\text{active}} = W_{last})\ ?\ v_{\text{next}} : v_{\text{active}}\,;$
                 $\textbf{goto}\, W_{\text{back}}, W_{\text{exit}}\,;$
$W_{\text{back}}$ : $\textbf{assume}\, v_{\text{active}} = F\,;$
                 $\textbf{goto}\, F\,;$
$W_{\text{exit}}$ : $\textbf{assume}\, v_{\text{active}} \neq W\,;$
                 $\textbf{goto}\, W_{end}\,;$
$W_{end}$      : $\textbf{assume}\,(v_{\text{active}} = W_{end}) \Rightarrow (offset \geq TS)\,;$
                 $\textbf{goto}\, End\,;$
```

Fig. 5: Predication of the kernel of Fig. 3

Predication of Kernels. Predicating a complete kernel P now consists of three steps: (1) Compute a sort order on blocks as detailed above; (2) Predicate every block with

$$\frac{\boldsymbol{val} = (\sigma, \sigma_t)(\boldsymbol{e})}{P \vdash \langle (\sigma, \sigma_t), \boldsymbol{v} := \boldsymbol{e} \rangle \overset{(\sigma,\sigma_t)}{\to} (\sigma, \sigma_t)[\boldsymbol{v} \mapsto \boldsymbol{val}]} \textsc{ Assigns}$$

$$\frac{\boldsymbol{val} \in \boldsymbol{D}}{P \vdash \langle (\sigma, \sigma_t), \mathbf{havoc}\, \boldsymbol{v} \rangle \overset{(\sigma,\sigma_t)}{\to} (\sigma, \sigma_t)[\boldsymbol{v} \mapsto \boldsymbol{val}]} \textsc{ Havocs}$$

$$\frac{\exists i : \sigma(e_i) \wedge val = (\sigma, \sigma_t)(e'_i)}{P \vdash \langle \sigma, v := \psi(\langle e_i, e'_i \rangle_{i=1}^{n}) \rangle \overset{(\sigma,\sigma_t)}{\to} \sigma[v \mapsto val]} \ \psi_{\mathrm{T}}$$

$$\frac{\forall\, i : \neg(\sigma, \sigma_t)(e_i)}{P \vdash \langle (\sigma, \sigma_t), v := \psi(\langle e_i, e'_i \rangle_{i=1}^{n}) \rangle \to (\sigma, \sigma_t)} \ \psi_{\mathrm{F}}$$

Fig. 6: Operational semantics vector and synchronisation statements

respect to the sort order, according to the rules of Table 2; (3) Insert the assignment $v_{\text{active}} := \mathit{Start}$ at the beginning of $\pi(\mathit{Start})$. The introduction of $v_{\text{active}} := \mathit{Start}$ ensures that the statements from $\pi(\mathit{Start})$ are always reduced first (see also Fig. 5).

4.2 Lock-Step Execution of All Threads

We now use the predication scheme of Sect. 4.1 for a single thread to define lock-step execution semantics for a kernel as a whole. We achieve this by encoding the kernel as a sequential program, each statement of which is a *vector* statement that performs the work of all threads simultaneously. To enable this, we first extend our programming language with these vector statements, as well as a statement related to barrier synchronisation.

Vector and Synchronisation Statements. We extend our language as follows:

$$\mathit{Stmt} ::= \cdots \mid \mathit{Var}^* := \mathit{Expr}^* \mid \mathbf{havoc}\ \mathit{Var}^* \mid \mathit{Var} := \psi((\mathit{Expr} \times \mathit{Expr})^*) \mid \mathbf{sync}\, \mathit{Expr}$$

The new vector assignment statement simultaneously assigns values to multiple variables, where we assume that the variables assigned to are all distinct and where the number of expressions is equal to the number of variables. Similarly, the new vector **havoc**-statement havocs multiple variables at once; again the variables are assumed to be distinct. The ψ-assignment will be used to model simultaneous writes to a shared variable by all threads. It takes a sequence $(e_1, e'_1), \ldots, (e_n, e'_n)$ with each e_i Boolean and non-deterministically assigns a value from $\{\sigma(e'_i) \mid 1 \leq i \leq n \wedge \sigma(e_i)\}$ to the variable v (if the set is empty, v is left unchanged).

The **sync**-statement, behaves exactly as an **assert**; we introduce an additional keyword for assertions to be able to differentiate between assertions in our lock-step programs that originate, resp., from barriers and assertions.

The semantics for the new statements is presented in Fig. 6, where $\langle e_i, e'_i \rangle_{i=1}^{n}$ denotes $(e_1, e'_1), \ldots, (e_n, e'_n)$. Since **sync** behaves like **assert**, it is omitted from the figure.

Lock-Step Execution. To encode a kernel P as a single-threaded program $\phi(P)$ which effectively executes all threads in lock-step, we assume for each *private* variable v

Predicated form		Lock-step form
$v_{\text{active}} := \mathit{Start}\,;$		$\langle v_{\text{active}} t\rangle_{t=1}^{TS} := \langle \mathit{Start}\rangle_{t=1}^{TS}$
$v := (v_{\text{active}} = B)\ ?\ e : v\,;$	v private	$\langle v_t\rangle_{t=1}^{TS} := \langle (v_{\text{active},t} = B)\ ?\ \phi_t(e) : v_t\rangle_{t=1}^{TS}\,;$
	v shared	$v := \psi(\langle v_{\text{active},t} = B, \phi_t(e)\rangle_{t=1}^{TS})\,;$
$\textbf{havoc}\, v_{\text{havoc}}\,;$ $v := (v_{\text{active}} = B)\ ?\ v_{\text{havoc}} : v\,;$	v private	$\textbf{havoc}\, \langle v_{\text{havoc},t}\rangle_{t=1}^{TS}\,;$ $\langle v_t\rangle_{t=1}^{TS} := \langle (v_{\text{active},t} = B)\ ?\ v_{\text{havoc},t} : v_t\rangle_{t=1}^{TS}\,;$
	v shared	$\textbf{havoc}\, \langle v_{\text{havoc},t}\rangle_{t=1}^{TS}\,;$ $v := \psi(\langle v_{\text{active},t} = B, v_{\text{havoc},t}\rangle_{t=1}^{TS})\,;$
$\textbf{assert}\,(v_{\text{active}} = B) \Rightarrow e\,;$		$\textbf{assert}\,\bigwedge_{t=1}^{TS}((v_{\text{active},t} = B) \Rightarrow \phi_t(e))$
$\textbf{assume}\,(v_{\text{active}} = B) \Rightarrow e\,;$		$\textbf{assume}\,\bigwedge_{t=1}^{TS}((v_{\text{active},t} = B) \Rightarrow \phi_t(e))$
$\textbf{skip}\,;$		$\textbf{skip}\,;$
$\textbf{barrier}\,(v_{\text{active}} = B)\,;$		$\textbf{sync}\,\left(\bigvee_{t=1}^{TS}(v_{\text{active},t} = B)\right) \Rightarrow \left(\bigwedge_{t=1}^{TS}(v_{\text{active},t} = B)\right)\,;$

(a) Statements

Predicated form		Lock-step form	
B	$:\ ss$	B	$:\ \phi(ss)$
	$v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,;$		$\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle\{B_1, \ldots, B_n\}\rangle_{t=1}^{TS}\,;$
	$v_{\text{active}} :=$		$\langle v_{\text{active},t}\rangle_{t=1}^{TS} :=$
	$(v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$		$\langle (v_{\text{active},t} = B)\ ?\ v_{\text{next},t} : v_{\text{active},t}\rangle_{t=1}^{TS}\,;$
	$\textbf{goto}\,\mathit{next}(B)\,;$		$\textbf{goto}\,\mathit{next}(B)\,;$
B	$:\ ss$	B	$:\ \phi(ss)$
	$v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,;$		$\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle\{B_1, \ldots, B_n\}\rangle_{t=1}^{TS}\,;$
	$v_{\text{active}} :=$		$\langle v_{\text{active},t}\rangle_{t=1}^{TS} :=$
	$(v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$		$\langle (v_{\text{active},t} = B)\ ?\ v_{\text{next},t} : v_{\text{active},t}\rangle_{t=1}^{TS}\,;$
	$\textbf{goto}\,B_{\text{back}}, B_{\text{exit}}\,;$		$\textbf{goto}\,B_{\text{back}}, B_{\text{exit}}\,;$
B_{back}	$:\ \textbf{assume}\,v_{\text{active}} = B_{\text{head}}\,;$	B_{back}	$:\ \textbf{assume}\,\bigvee_{t=1}^{TS}(v_{\text{active},t} = B_{\text{head}})\,;$
	$\textbf{goto}\,B_{\text{head}}\,;$		$\textbf{goto}\,B_{\text{head}}\,;$
B_{exit}	$:\ \textbf{assume}\,v_{\text{active}} \neq B_{\text{head}}\,;$	B_{exit}	$:\ \textbf{assume}\,\bigwedge_{t=1}^{TS}(v_{\text{active},t} \neq B_{\text{head}})\,;$
	$\textbf{goto}\,\mathit{next}(B)\,;$		$\textbf{goto}\,\mathit{next}(B)\,;$

(b) Blocks

Table 3: Lock-step construction

from P that there exists a variable v_t in $\phi(P)$ for every $1 \leq t \leq TS$. For each *shared* variable v from P we assume there exists an identical variable in $\phi(P)$. Construction of a lock-step program for P starts from $\pi(P)$ — the predicated version of P.

Statements. The construction for the predicated statements from Table 1 is presented in Table 3a. The construction involves making a copy of each statement for each thread. In the table, ϕ_t denotes a map over expressions which replaces each private variable v by v_t. Note that for every thread t, there exists a variable $v_{\text{active},t}$, as variables freshly introduced by the predication scheme of Sect. 4.1 are private. Hence, we always know for each thread which block should be reduced next. We discuss the statements in turn.

Initially, each $v_{\text{active},t}$ is assigned to Start. For every other assignment, we distinguish between assignments to private and shared variables. For a private variable v, the assignment is replaced by a vector assignment to the variables v_t, where ϕ_t is applied to e

```
$B_2$        : $\textbf{sync}\left(\bigvee_{t=1}^{TS}(v_{\text{active},t} = B)\right) \Rightarrow \left(\bigwedge_{t=1}^{TS}(v_{\text{active},t} = B)\right)$ ;
               $\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle\{W_{last}\}\rangle_{t=1}^{TS}$ ;
               $\langle v_{\text{active},t}\rangle_{t=1}^{TS} := \langle(v_{\text{active},t} = B_2)\ ?\ v_{\text{next},t} : v_{\text{active},t}\rangle_{t=1}^{TS}$ ;
               $\textbf{goto}\, W_{last}$ ;
$W_{last}$   : $\langle \mathit{offset}_t\rangle_{t=1}^{TS} := \langle(v_{\text{active},t} = F_{end})\ ?\ (2 \cdot \mathit{offset}_t) : \mathit{offset}_t\rangle_{t=1}^{TS}$ ;
               $\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle\{F, F_{exit}\}\rangle_{t=1}^{TS}$ ;
               $\langle v_{\text{active},t}\rangle_{t=1}^{TS} := \langle(v_{\text{active},t} = W_{end})\ ?\ v_{\text{next},t} : v_{\text{active},t}\rangle_{t=1}^{TS}$ ;
               $\textbf{goto}\, W_{\text{back}}, W_{\text{exit}}$ ;
$W_{\text{back}}$ : $\textbf{assume}\, \bigvee_{t=1}^{TS}(v_{\text{active},t} = W)$ ;
               $\textbf{goto}\, W$ ;
$W_{\text{exit}}$ : $\textbf{assume}\, \bigwedge_{t=1}^{TS}(v_{\text{active},t} \neq W$ ;
               $\textbf{goto}\, W_{end}$ ;
```

Fig. 7: Part of the lock-step program for the kernel of Fig. 3

as appropriate. For a shared variable v, it is not obvious which value needs to be assigned to v, as there might be multiple threads t with $v_{\text{active},t} = B$; we non-deterministically pick the value from one of the threads with $v_{\text{active},t} = B$, employing a ψ-assignment.

In the case of a **havoc** followed by an assignment, there is again a case distinction between private and shared variables. For a private variable, the **havoc** and assignment are simply replaced by corresponding vector statements. For a shared variable, a vector havoc is used to produce an arbitrary value for each thread, and then the value associated with one of the threads t with $v_{\text{active},t} = B$ non-deterministically assigned employing ψ.

In the case of **assert** and **assume**, we test whether $(v_{\text{active},t} = B) \Rightarrow \phi_t(e)$ for each thread $1 \leq t \leq TS$. The **skip**-statement remains a no-op.

Lock-step execution of a **barrier** statement with condition $v_{\text{active}} = B$ translates to an assertion checking that if $v_{\text{active},t} = B$ holds for *some* thread t then it must hold for *all* threads. We shall sketch in Sect. 5 that checking for barrier divergence in this manner is equivalent to checking for barrier divergence using the interleaving semantics of Sect. 3. However, contrary to the interleaving case, there is no need to consider barrier variables in the lock-step case.

Blocks. The lock-step construction for blocks is presented in Table 3b, where $\phi(ss)$ denotes the lock-step form for a sequence of statements.

If a block is *not* sorted last among the blocks of a loop (see the top row of Table 3b), we simply vectorise the updating v_{active}, where $:\in$ is extended in the obvious way to non-deterministically assign values from multiple sets to multiple variables.

If a block *is* sorted last among blocks in a loops L then the successors of the block in the predicated program are B_{back}, which leads to the loop head, and B_{exit}, which leads to a node outside the loop. Our goal is to enforce the rule that *no* thread should leave the loop until *all* threads are ready to leave the loop, discussed informally in Sect. 2 and illustrated for structured programs by the guard of the **while** loop in Fig. 2. To achieve this, the bottom row of Table 3b uses an **assume** statement in B_{back} requiring that $v_{\text{active}} = B_{\text{head}}$ for *some* thread, and an **assume** statement in B_{exit} requiring $v_{\text{active}} \neq B_{\text{head}}$ for *all* threads (see also Fig. 7 for a concrete example).

Lock-Step Semantics and Data Races. Having completed our definition of the lock-step construction $\phi(P)$ for a kernel P, we now say that the lock-step semantics for P is the *interleaving* semantics for $\phi(P)$, with respect to a *single* thread (i.e. with $TS = 1$). Our use of **sync** statements, which behave like assertions, captures the notion of barrier divergence. However, we need to define how data races can be detected by examining lock-step execution traces.

We say that thread t is *enabled* during a reduction step if the statement being reduced occurs in block B and $v_{\text{active},t} = B$ holds at the point of reduction.

Let v be variable. We say that thread t *reads from* v during a reduction step if t is enabled during the reduction step and the reduction step involves evaluating an expression containing v. We say that t *writes to* v during a reduction step if t is enabled during the reduction step and the statement being reduced is an assignment to v. Note that in the case of a write, if multiple threads are enabled then v will be updated non-deterministically using one of the values supplied by the enabled threads. Nevertheless, we regard *all* enabled threads as having written to v.

A data race in a lock-step program is defined as follows:

Definition 4.1. *Let $\phi(P)$ be the lock-step form of a kernel P. Then* $\phi(P)$ has a data race *if there is a maximal reduction ρ, distinct threads t and t', and a shared variable v such that: ρ does not end in* infeasible*; t writes to v during ρ; t' either writes to or reads from v during ρ; no* SYNC *occurs between the accesses (i.e. no barrier separates them).*

Terminating and Race Free Kernels. We say that a kernel P is *terminating* with respect to the lock-step semantics if all maximal reductions of $\phi(P)$ are finite and do not end with status *error*. We say that P is *race free* with respect to the lock-step semantics if $\phi(P)$ has no data races according to Definition 4.1.

5 Equivalence Between Interleaving and Lock-Step Semantics

We are now in a position to prove our main result, an equivalence between the kernels with interleaving semantics of Sect. 3 and lock-step programs of Sect. 4. The result applies to *well-formed* kernels, which we define as follows:

Definition 5.1. *A kernel P is* well-formed *if the following conditions hold for every block B in P:*

1. *The first statement of B is* **assume** *e where e refers only to private variables.*
2. *No other* **assume***-statements appear in B.*
3. *If B ends with* **goto** *$B_1, \ldots, B_n$, and every B_i begins with* **assume** *e_i, then $\bigvee_{i=1}^{n} e_i$ is a tautology.*

Well-formedness means that, with the interleaving semantics, the *infeasible* status can only result from non-deterministic branching in a **goto** statement leading to a failing **assume** statement, and that in this case there is always an alternative way in which the branch could be resolved that would *not* lead to *infeasible*.

We can trivially enforce condition 1 of Definition 5.1 by inserting statements of the form **assume** *true* where necessary and removing shared variable accesses from

$\mathbf{assume}$s via private temporary variables. Conditions 2 and 3 are guaranteed to hold if the CFG for P has been obtained from a kernel written in a C-like language such as OpenCL or CUDA.

Our main theorem, the following soundness and completeness result, applies to well-formed kernels:

Theorem 5.2. *Let P be a well-formed kernel and let $\phi(P)$ be the lock-step version of P. Then, P is race free and terminating with respect to the interleaving semantics iff P is race free and terminating with respect to the lock-step semantics. Moreover, if race-freedom holds then for every terminating reduction of P there exists a terminating reduction of $\phi(P)$, and vice versa, such that every shared variable v has the same value at the end of both reductions.*

To see why well-formedness is required, consider the following kernel, where each thread t has a private variable tid whose value is t and where v is a shared:

```
Start : goto B_1, B_2 ;     B_1 : assume tid = 1 ;     B_2 : assume tid ≠ 1 ;
                                  v := 4 ;                   v := 5 ;
                                  assume v = 5 ;             goto End ;
                                  goto End ;
```

The interleaving semantics allows for reduction of $\mathbf{assume}\, v = 5$ after all assignments in all threads have taken place. Hence, if the assignment by thread 1 is not last among these, $v = 5$ evaluates to $true$, and eventually termination occurs, with a data race. In the case of lock-step execution and assuming $Start \leq B_1 \leq B_2$ in the sort order, we have that $\mathbf{assume}\, v = 5$ is always reduced immediately after $v := 4$. Hence, every reduction terminates with *infeasible* and no data race occurs.

That termination is required follows by adapting the counterexamples from [12,11] showing that CUDA hardware does not necessarily schedule threads from a non-terminating kernel in a way that that is fair from an interleaving point-of-view.

The proof of the theorem proceeds by showing that P and its predicated version $\pi(P)$ are stutter equivalent, and then establishing a relationship between $\pi(P)$ and $\phi(P)$.

Equality of P and $\pi(P)$. To show that P and $\pi(P)$ are stutter equivalent [15], we define a denotational semantics of kernels is defined in terms of *execution traces* [5], i.e., sequences of tuples $(\sigma, \vec{\sigma}) = (\sigma, \sigma_1, \ldots, \sigma_{TS})$ with σ the shared store and σ_t the private store of thread t.

Definition 5.3. *Let ρ be a maximal reduction. The* denotation *or* execution trace $\mathcal{D}(\rho)$ *of ρ is the sequence of $\rightarrow$-labels of ρ together with the termination status of ρ in case ρ terminates. Let $(b_1, \ldots, b_{TS})$ be a tuple block bodies or block ids. The* denotation $\mathcal{D}(b_1, \ldots, b_{TS})$ of $(b_1, \ldots, b_{TS})$ *is the set of denotations of all maximal reductions of $(b_1, \ldots, b_{TS})$ for all initial stores $\sigma, \sigma_1, \ldots, \sigma_t$* not *terminating as* infeasible. *Let P be a kernel. The* denotation $\mathcal{D}(P)$ of P *is* $\mathcal{D}(Start, \ldots, Start)$.

Observe that *infeasible* traces are *not* included in the denotations of $(b_1, \ldots, b_{TS})$ and P; these traces do not constitute actual program behaviour.

Stutter equivalence is defined on subsets of variables, where a *restriction* of a store σ to a set of variables V is denoted by $\sigma\restriction_V$ and, where given a tuple $(\sigma, \vec{\sigma}) = (\sigma, \sigma_1, \ldots, \sigma_{TS})$, the restriction $(\sigma, \vec{\sigma})\restriction_V$ is $(\sigma, \vec{\sigma})\restriction_V = (\sigma\restriction_V, \sigma_1\restriction_V, \ldots, \sigma_{TS}\restriction_V)$.

Definition 5.4. *Let V be a set of variables. Define the map δ_V over execution traces as the map that replaces every maximal subsequence $(\sigma_1, \vec{\sigma}_1)\,(\sigma_2, \vec{\sigma}_2) \cdots (\sigma_n, \vec{\sigma}_n) \cdots$ where $(\sigma_1, \vec{\sigma}_1)\restriction_V = (\sigma_2, \vec{\sigma}_2)\restriction_V = \ldots = (\sigma_n, \vec{\sigma}_n)\restriction_V = \ldots$ by $(\sigma_1, \vec{\sigma}_1)$.*

Let Σ and T be execution traces. The traces are stutter equivalent *with respect to V, denoted $\Sigma \sim_{\text{st}}^{V} T$, iff:*

- *Σ and T are both finite with equal termination statuses and $\delta_V(\Sigma) = \delta_V(T)$;*
- *Σ and T are both infinite and $\delta_V(\Sigma) = \delta_V(T)$.*

Let P and Q be kernels. The kernels are stutter equivalent *with respect to V, denoted $P \sim_{\text{st}}^{V} Q$, iff for every $\Sigma \in \mathcal{D}(P)$ there is a $T \in \mathcal{D}(Q)$ with $\Sigma \sim_{\text{st}}^{V} T$, and vice versa.*

Recall that $\pi(P)$ denotes predication of P, we have the following:

Theorem 5.5. *If P is a kernel with variables V, then $\pi(P) \sim_{\text{st}}^{V} P$. A data race occurs in P iff a data race occurs in $\pi(P)$ where, during reduction of neither of the two statements causing the data race, $v_{\text{active}} \neq B$ with B is the block containing the statement.*

The result follows immediately by a case distinction on the statements that may occur in kernels once we establish the following lemma, which is a direct consequence of our construction and the first requirement on the sort order of blocks.

Lemma 5.6. *Let P be a kernel with variables V. For any thread t and each block B of P, if (σ, σ_t) is a store of t and $(\hat{\sigma}, \hat{\sigma}_t)$ is a store of in t in $\pi(P)$ such that $\hat{\sigma}\restriction_V = \sigma$ and $\hat{\sigma}(v_{\text{active},t}) = B$, then*

1. *if the reduction of B is immediately followed by the reduction of a block C, then there exists a reduction of $\pi(B)$ such that $v_{\text{active},t} = C$ at the end of $\pi(B)$ and eventually $\pi(C)$ is reduced;*
2. *if the reduction of $\pi(B)$ ends with $v_{\text{active},t} = C$, then there exists a reduction of B that is immediately followed by the reduction of a block C.*

Soundness and Completeness. Theorem 5.2 is now proved as follows.

Proof (Sketch[5]). For termination and race-freeness of $\phi(P)$, it suffices by Lemma 5.6 to consider $\pi(P)$ — the predicated version of P. Reason by contradiction and construct for a reduction of $\phi(P)$ which is either infinite or has data race, a reduction of $\pi(P)$ that also is either infinite or has a data race: Replace each statement and goto from the right-hand columns of Table 3 by a copy of the statement or goto in the left-hand column and reduce, where we introduce a copy for each thread. That SYNC can be replaced by BARRIER$_\text{S}$ follows as no statements from outside loops can be reduced while we are inside a loop (cf. the second requirement on sort order of blocks).

The remainder of the theorem follows by permuting steps of different threads so the reverse transformation from above can be applied. □

[5] A full proof of the theorem is provided at `http://multicore.doc.ic.ac.uk/tools/GPUVerify/ESOP2012/paper.pdf`.

6 Implementation and Experiments

Implementation in GPUVerify. We have implemented the predication technique described here in GPUVerify [7], a verification tool for OpenCL and CUDA kernels built on top of the Boogie verification engine [6] and Z3 SMT solver [20]. GPUVerify previously employed a predication technique for structured programs. Predication for CFGs has allowed us to build a new front-end for GPUVerify which takes LLVM intermediate representation (IR) as input; IR directly corresponds to a CFG. This allows us to compile OpenCL and CUDA kernels using the Clang/LLVM framework and perform analysis on the resulting IR. Hence, tricky syntactic features of C-like languages are taken care of by Clang/LLVM. Analysing kernels after compilation and optimisation also increases the probity of verification, opening up the opportunity to discover compiler-related bugs.

Experimental Evaluation. To assess the performance overhead in terms of verification time for our novel predication scheme and associated tool chain we compared our new implementation (GPUVerify II) with the original structured one (GPUVerify I).

We compared the tool versions using 163 OpenCL and CUDA kernels drawn from the AMD Accelerated Parallel Processing SDK v2.6 [4] (71 OpenCL kernels), the NVIDIA GPU Computing SDK v2.0 [21] (20 CUDA kernels), Microsoft C++ AMP Sample Projects [19] (20 kernels translated from C++ AMP to CUDA) and Rightware's Basemark CL v1.1 [23] suite (52 OpenCL kernels, provided to us under an academic license). These kernels were used for analysis of GPUVerify I in [7], where several of the kernels had to be manually modified before they could be subjected to analysis: 4 kernels exhibited unstructured control flow due to **switch**-statements, and one featured a **do-while**-loop which was beyond the scope of the predication scheme of [7]. Furthermore, unstructured control flow arising from short-circuit evaluation of logical operators had been overlooked in GPUVerify I, which affected 30 of our example kernels. In GPUVerify II all kernels are handled uniformly due to our novel predication scheme in combination with the use of Clang/LLVM, which removes short-circuit evaluation in favour of unstructured control flow.

All experiments were performed on a PC with a 3.6 GHz Intel i5 CPU, 8 GB RAM running Windows 7 (64-bit), using Z3 v4.1. All times reported are averages over 3 runs. Both tool versions and all our benchmarks, except the commercial Basemark CL kernels, are available online to make our results reproducible:

`http://multicore.doc.ic.ac.uk/tools/GPUVerify/ESOP2012`

The majority of our benchmark kernels could be automatically verified by both GPUVerify I and GPUVerify II; 22 kernels were beyond the scope of both tools and result in a failed proof attempt. Key to the usability of GPUVerify is its *response time*, the time the tool takes to either report successful verification vs. a failed proof attempt. Comparing GPUVerify I and GPUVerify II we found that across the entire benchmark set the analysis time taken by GPUVerify II was 2.25 times that of GPUVerify I. The average and longest analysis time across all kernels were 4 seconds and 157 seconds respectively for GPUVerify I, and 10 seconds and 300 seconds respectively for GPUVerify II. Thus overall the accurate handling of unstructured control flow afforded by GPUVerify II comes at the price of a moderate performance penalty.

On 124 of the 163 kernels (76%), GPUVerify II was marginally (though not significantly) faster than GPUVerify I. For a further 21 kernels (13%) GPUVerify II was up to 50% slower than GPUVerify I. The remaining 18 kernels (11%) caused the slow down on average. In each case the time it took to run the front-ends and predication engines of both tool chains was negligible; the difference lay in constraint solving times; the SMT queries generated by our CFG-based tool chain can be somewhat more complex than in the structured case. The most dramatic example is a kernel which was verified by GPUVerify I and GPUVerify II in 3 and 202 seconds, resp., a slow-down for GPUVerify II of 70 times. This kernel exhibits a large number of shared memory accesses. In the LLVM IR processed by GPUVerify II these accesses are expressed as many separate, contiguous loads and stores, requiring reasoning about race-freedom between many pairs of operations. The structured approach of GPUVerify I captures these accesses at the abstract syntax tree level, allowing a load/store from/to a contiguous region to be expressed as a single access, significantly simplifying reasoning. This illustrates that there are benefits to working at the higher level of abstract syntax trees, and suggests that we might implement optimisations in GPUVerify II to automatically identify and merge contiguous memory accesses.

7 Related Work and Conclusion

Related Work. Interleaving semantics for GPU kernels has been defined by [16,18,12]. These are similar to our semantics, except that [16,12] do not give a semantics for barriers. Contrary to our lock-step approach, [16,18] battle the state space explosion due to arbitrary interleavings of threads by considering one particular schedule.

In [11,12], a semantics of CUDA kernels is defined that tries to model NVIDIA hardware as faithfully as possible. The focus is, however, not on predicated execution (although it does figure briefly in [11]), but on so-called *immediate post-dominator re-convergence* [10], a method to continue lock-step execution of threads as soon as possible after branch divergence has occurred between threads.

In addition to the above and similar to us, [12] shows for terminating kernels that CUDA execution of kernels can be faithfully simulated by certain interleaving thread schedules. The reverse is not shown; our analysis is that such a result is difficult to establish due to data races that occur in their examples.

Conclusion. Our lock-step semantics for GPU kernels expressed as arbitrary reducible CFGs enables automated analysis of a wider class of GPU kernels than previous techniques for structured programs, and allows for the analysis of compiled kernel code, after optimisations have been applied. Our soundness and completeness result establishes an equivalence between our lock-step semantics and a traditional semantics based on interleaving, and our implementation in GPUVerify and associated experimental evaluation demonstrate that our approach is practical.

Because our kernel programming language supports non-deterministic choice and havocking of variables it can express an over-approximation of a concrete kernel. In future work we plan to exploit this, investigating the combination of source-level abstraction techniques such as predicate abstraction with GPUVerify's verification method.

The well-formedness restrictions of Definition 5.1 mean that our equivalence result does not apply to kernels that exhibiting "dead end" paths. This is relevant if such paths are introduced through under-approximation, e.g., unwinding a loop by a fixed number of iterations in the style of bounded model checking. We plan to investigate whether it is possible to relax these well-formedness conditions under certain circumstances.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Pearson Education, 2nd edn. (2007)
2. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL'83. pp. 177–189 (1983)
3. Alshawabkeh, M., Jang, B., Kaeli, D.: Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems. In: GPGPU-3. pp. 104–110 (2010)
4. AMD: AMD Accelerated Parallel Processing (APP) SDK, `http://developer.amd.com/sdks/amdappsdk/pages/default.aspx`
5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE'05. pp. 82–87 (2005)
6. Barnett, M., et al.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO 2005. LNCS, vol. 4111, pp. 364–387 (2005)
7. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA 2012 (2012)
8. Collingbourne, P., , Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: HVC 2011 (2012)
9. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)
10. Fung, W.W., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic warp formation and scheduling for efficient GPU control flow. In: MICRO 2007. pp. 407–418 (2007)
11. Habermaier, A.: The model of computation of CUDA and its formal semantics. Tech. Rep. 2011-14, University of Augsburg (2011)
12. Habermaier, A., Knapp, A.: On the correctness of the SIMT execution model of GPUs. In: ESOP 2012. LNCS, vol. 7211, pp. 316–335 (2012)
13. Hecht, M.S., Ullman, J.D.: Characterizations of reducibe flow graphs. Journal of the ACM 21(3), 367–375 (1974)
14. Khronos Group: The OpenCL specification, version 1.2 (2011)
15. Lamport, L.: What good is temporal logic? In: Information Processing 83. pp. 657–668 (1983)
16. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. In: PLDI 2012. pp. 383–394 (2012)
17. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: FSE 2010. pp. 187–196 (2010)
18. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: PPoPP 2012. pp. 215–224 (2012)
19. Microsoft Corporation: C++ AMP sample projects for download, `http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx`
20. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)
21. NVIDIA: CUDA Toolkit Release Archive, `http://developer.nvidia.com/cuda/cuda-toolkit-archive`

22. NVIDIA: NVIDIA CUDA C Programming Guide, Version 4.2 (2012)
23. Rightware Oy: Basemark CL, `http://www.rightware.com/en/Benchmarking+Software/Basemark\%99+CL`
24. Zhu, F., Chen, P., Yang, D., Zhang, W., Chen, H., Zang, B.: A GPU-based high-throughput image retrieval algorithm. In: GPGPU-5. pp. 30–37 (2012)

A Control Flow Graphs

The definition of *control flow graphs (CFGs)* is as usual [1]. Given a CFG G, a node d *dominates a node* n, if every path from the root of G to n goes through d. A *back edge* is an edge whose target dominates its source. A CFG G is *reducible* if the maximal acyclic CFG that is a subgraph of G is unique [13]. In a reducible CFG, the maximal direct acyclic subgraph is obtained by removing all back edges. A *loop with head* n in a reducible CFG is a maximal set of nodes and edges L with a node $n \in L$ such that (a) n dominates all other nodes in L and (b) for every node $m \in L$ there is a path to n. The *body* of a loop L is the set of all nodes and edges in L except for the back edges.

B Omitted Proofs

To fully prove Theorem 5.2, we first consider the one-threaded, sequential language embedded in our kernel language, i.e. the language whose rules occur in Figs. 4b and 4a. Then, for full generality we extend this language with procedures. Having finished the discussion of the sequential language, we prove Theorem 5.5. This is followed by the actual proof of Theorem 5.2.

Recall that any newly introduced variables or block identifiers are called *fresh*.

B.1 Stutter Equivalence of Sequential Programs

To allow us to prove results about statements or sequences of statements in isolation, without having to consider blocks, we introduce one additional rule on the level of statements:

$$\frac{P \vdash \langle \sigma, \sigma_t, s \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} (\tau, \tau_t)}{P \vdash \langle \sigma, \sigma_t, s \rangle \overset{(\sigma,\sigma_t)}{\rightarrow} \langle \tau, \tau_t, \mathit{End} \rangle} \textsc{SeqS}$$

Thus, termination occurs after we reduce a statement in complete isolation.

For the purpose of this section, we assume that the above rule and the rules in Figs. 4b and 4a are defined over a *single store*, where we write $\langle \sigma, b \rangle$ and where the $\rightarrow$-labels are simplified accordingly to σ. The proofs of this section carry over to the two-store setting, mutatis mutandis. We ignore the **barrier**-statement for the purposes of this section.

Remark B.1. Observe that $\sigma[v \mapsto \sigma(v)] = \sigma$, i.e. assigning a variable the value it already has does *not* modify the store. Observe also that the only rules that can modify a store are the ASSIGN- and HAVOC-rules, which only modify the value of the variable being assigned to or havocked; evaluation of expressions does *not* modify the store.

Denotational Semantics. The denotational semantics for the sequential language is defined as follows, cf. Definition 5.3:

Definition B.2. *Let ρ be a maximal reduction. The* denotation *or* execution trace $\mathcal{D}(\rho)$ of ρ *is the sequence of $\to$-labels of ρ together with the termination status of ρ in case ρ terminates. Let b be a statement, a sequence of statements, or a block. The* denotation $\mathcal{D}(b)$ of b *is the set of denotations of all maximal reductions of b for all initial stores* σ not *terminating as* infeasible. *Let P be a program. The* denotation $\mathcal{D}(P)$ of P *is* $\mathcal{D}(Start)$.

As before, that *infeasible* traces are *not* included in the denotations of b and P; these traces do not constitute actual program behaviour.

The following is immediate by the observation that the END- and BLOCK-rules do not modify the store:

Proposition B.3. *If ss and tt are sequences of statements, then $\mathcal{D}(ss\,;\,tt)$ is the set of all infinite and* error *traces of ss together with all traces $\Sigma\,\tau\,\Upsilon$ where $\Sigma\,\tau$ is a terminating trace of ss and $\tau\,\Upsilon \in \mathcal{D}(tt)$.*

If ss is a sequence of statements and $B_1, \ldots, B_n$ are blocks, then it holds that $\mathcal{D}(ss\;\mathbf{goto}\,B_1, \ldots, B_n\,;)$ is the set of all infinite and error *traces of ss together with all traces $\Sigma\,\tau\,\Upsilon$ where $\Sigma\,\tau$ is a terminating trace of ss and $\tau\,\Upsilon \in \mathcal{D}(B_i)$ for some $1 \le i \le n$.*

As the SKIP-rule has an empty $\to$-label, we also have:

Proposition B.4. *If ss is a sequence of statements, then*

$$\mathcal{D}(ss) = \mathcal{D}(\mathbf{skip}\,;\,ss) = \mathcal{D}(ss\;\mathbf{skip}\,;)\,.$$

As mentioned in Sect. 5, we compare programs based on their denotations by means of a stutter equivalence on subsets of variables. Recall that the *restriction* of a store σ to a set of variables V is denoted by $\sigma\restriction_V$. Stutter equivalence for our sequential language is defined as follows, cf. Definition 5.4:

Definition B.5. *Let V be a set of variables. Define the map δ_V over execution traces as the map that replaces every maximal consecutive subsequence $\sigma_1\,\sigma_2 \cdots \sigma_n \cdots$ where $\sigma_1\restriction_V = \sigma_2\restriction_V = \ldots = \sigma_n\restriction_V = \ldots$ by $\sigma_1\restriction_V$.*

Let Σ and T be execution traces. The traces are stutter equivalent *with respect to V, denoted $\Sigma \sim^V_{\mathrm{st}} T$, iff:*

- *Σ and T are both finite with equal termination statuses and $\delta_V(\Sigma) = \delta_V(T)$;*
- *Σ and T are both infinite and $\delta_V(\Sigma) = \delta_V(T)$.*

Let ss and tt be sequences of statements. The sequences are stutter equivalent *with respect to V, denoted $ss \sim^V_{\mathrm{st}} tt$, iff for every $\Sigma \in \mathcal{D}(ss)$ there exists a $T \in \mathcal{D}(tt)$ such that $\Sigma \sim^V_{\mathrm{st}} T$, and vice versa.*

Let B and C be blocks and let ss and tt be the statements of, resp., B and C. The blocks are block level stutter equivalent *with respect to V, denoted $B \sim^V_{\mathrm{st}} C$, iff $ss \sim^V_{\mathrm{st}} tt$.*

Let P and Q be programs. The programs are stutter equivalent *with respect to V, denoted $P \sim^V_{\mathrm{st}} Q$, iff for every $\Sigma \in \mathcal{D}(P)$ there exists a $T \in \mathcal{D}(Q)$ such that $\Sigma \sim^V_{\mathrm{st}} T$, and vice versa.*

Observe that block level stutter equivalence is only concerned with the statements in blocks and *disregards* gotos.

Stutter Equivalence of Statements. Denote predication of statements by π, we have:

Lemma B.6. *Let P be a sequential program with variables V. For every statement s of P in a block B*

1. *if $v_{\text{active}} = B$, then $\pi(s) \sim_{\text{st}}^{V} s$;*
2. *if $v_{\text{active}} \neq B$, then $\pi(s) \sim_{\text{st}}^{V}$ $\mathbf{skip}$.*

Proof. We consider $v_{\text{active}} = B$ and $v_{\text{active}} \neq B$ in turn. Suppose $v_{\text{active}} = B$. Given a store σ of P and a store $\hat{\sigma}$ of $\pi(P)$ with $\hat{\sigma}\restriction_V = \sigma$, we have:

– If s is $v := e$, then the only trace of s is the terminating trace $\sigma\,\tau$ with $\tau = \sigma[v \mapsto \sigma(e)]$. As $v_{\text{active}} = B$, the only trace of $\pi(s)$ is the terminating trace $\hat{\sigma}\,\hat{\tau}$ with $\hat{\tau} = \hat{\sigma}[v \mapsto \hat{\sigma}(e)]$. Hence, as e only depends on variables in V, for e originates from P, we have $\hat{\tau}\restriction_V = \tau\restriction_V$.
– If s is $\mathbf{havoc}\,v$, then the traces of s are the terminating traces $\sigma\,\upsilon_{val}$ with $\upsilon_{val} = \sigma[v \mapsto val]$ for $val \in D$. As $v_{\text{active}} = B$, the traces of $\pi(s)$ are the terminating traces $\hat{\sigma}\,\hat{\tau}_{val}\,\hat{\upsilon}_{val}$ with $\hat{\tau}_{val} = \hat{\sigma}[v_{\text{havoc}} \mapsto val]$ and $\hat{\upsilon}_{val} = \hat{\tau}_{val}[v \mapsto val]$. As v_{havoc} is fresh, $\hat{\sigma}\restriction_V = \hat{\tau}_{val}\restriction_V$ and $\hat{\upsilon}_{val}\restriction_V = \hat{\sigma}[v \mapsto val]\restriction_V$. Hence, $\hat{\tau}_{val}\restriction_V = \sigma\restriction_V$ and $\hat{\upsilon}_{val}\restriction_V = \upsilon_{val}\restriction_V$ for every $val \in D$.
– If s is $\mathbf{assert}\,e$, then the only trace of s is (a) the terminating trace σ in case $\sigma(e)$ holds, or (b) the *error* trace σ in case $\neg\sigma(e)$ holds. As $v_{\text{active}} = B$, the only trace of $\pi(s)$ is (a) the terminating trace $\hat{\sigma}$ in case $\hat{\sigma}(e)$ holds, and (b) the *error* trace $\hat{\sigma}$ in case $\neg\hat{\sigma}(e)$ holds. Hence, as e only depends on variables in V, the termination statuses of the traces σ and $\hat{\sigma}$ are equal.
– If s is $\mathbf{assume}\,e$, then the only trace of s is (a) the terminating trace σ in case $\sigma(e)$ holds, or (b) the *infeasible* trace σ in case $\neg\sigma(e)$ holds. As $v_{\text{active}} = B$, the only trace of $\pi(s)$ is (a) the terminating trace $\hat{\sigma}$ in case $\hat{\sigma}(e)$ holds, and (b) the *infeasible* trace $\hat{\sigma}$ in case $\neg\hat{\sigma}(e)$ holds. Hence, as e only depends on variables in V, the termination statuses of the traces σ and $\hat{\sigma}$ are equal.
– If s is $\mathbf{skip}$, then the only trace of s is the terminating trace σ and the only trace of $\pi(s)$ is the terminating trace $\hat{\sigma}$.

As σ is a store of P, we have $\sigma\restriction_V = \sigma$. Thus, as σ and $\hat{\sigma}$ are arbitrary with $\hat{\sigma}\restriction_V = \sigma$, it follows by the above that $\pi(s) \sim_{\text{st}}^{V} s$ in case $v_{\text{active}} = B$.

Suppose $v_{\text{active}} \neq B$. Given a store σ of P, the only trace of $\mathbf{skip}$ is the terminating trace σ. Moreover, given a store $\hat{\sigma}$ of $\pi(P)$ such that $\hat{\sigma}\restriction_V = \sigma$:

– Suppose s is $v := e$. As $v_{\text{active}} \neq B$, the only trace of $\pi(s)$ is the terminating trace $\hat{\sigma}\,\hat{\tau}$ with $\hat{\tau} = \hat{\sigma}[v \mapsto \hat{\sigma}(v)] = \hat{\sigma}$ (see also Remark B.1).
– Suppose s is $\mathbf{havoc}\,v$. As $v_{\text{active}} \neq B$, the traces of $\pi(s)$ are the terminating traces $\hat{\sigma}\,\hat{\tau}_{val}\,\hat{\upsilon}_{val}$ with $\hat{\tau}_{val} = \hat{\sigma}[v_{\text{havoc}} \mapsto val]$ and $\hat{\upsilon}_{val} = \hat{\tau}_{val}[v \mapsto \hat{\tau}_{val}(v)] = \hat{\tau}_{val}$. Hence, as v_{havoc} is fresh, $\hat{\sigma}\restriction_V = \hat{\tau}_{val}\restriction_V = \hat{\upsilon}_{val}\restriction_V$.
– Suppose s is $\mathbf{assert}\,e$. As $v_{\text{active}} \neq B$, the only trace of $\pi(s)$ is the terminating trace $\hat{\sigma}$.

– Suppose s is $\mathbf{assume}\, e$. As $v_{\text{active}} \neq B$, the only trace of $\pi(s)$ is the terminating trace $\hat{\sigma}$.
– Suppose s is $\mathbf{skip}$. The only terminating trace of $\pi(s)$ is the terminating trace $\hat{\sigma}$.

As σ is a store of P, we have $\sigma\!\restriction_V = \sigma$. Thus, as σ and $\hat{\sigma}$ are arbitrary with $\hat{\sigma}\!\restriction_V = \sigma$, it follows that $\pi(s) \sim^V_{\text{st}}$ **skip** in case $v_{\text{active}} \neq B$. □

Define $\pi(\varepsilon) = \varepsilon$ and $\pi(s\,;\,ss) = \pi(s)\,;\,\pi(ss)$. We have:

Lemma B.7. *Let P be a sequential program with variables V. For every sequence of statements ss of P in a block B*

1. *if $v_{\text{active}} = B$, then $\pi(ss) \sim^V_{\text{st}} ss$;*
2. *if $v_{\text{active}} \neq B$, then $\pi(ss) \sim^V_{\text{st}}$ $\mathbf{skip}$.*

Proof. By induction on the structure of ss using Proposition B.3 and employing (1) Lemma B.6(1) if $v_{\text{active}} = B$ and (2) Lemma B.6(2) and Proposition B.4 if $v_{\text{active}} \neq B$. □

Stutter Equivalence of Blocks. To achieve full generality with regard to stutter equivalence, we first drop the requirement that if a block ends with $\mathbf{goto}\, B_1, \ldots, B_n$, then at most one of $B_1, \ldots, B_n$ is a loop head. Predication of blocks is now defined as in Table 4. Where in the second row, B is assumed to be the last node, according to the sort order, of loops $L_1, \ldots L_n$ and where $B^{L_1}_{\text{head}}, \ldots, B^{L_n}_{\text{head}}$ are the heads of the loops. The identifiers $B^{L_1}_{\text{back}}, \ldots, B^{L_n}_{\text{back}}$ and B_{exit} are fresh in the second row of the table. Note that the head of a loop L is the block that is always sorted first among the blocks of L (as it dominates all other blocksin L).

We have the following:

Lemma B.8. *Let P be a sequential program with variables V. For every block B of P*

1. *if $v_{\text{active}} = B$, then $\pi(B) \sim^V_{\text{st}} B$;*
2. *if $v_{\text{active}} \neq B$, then $\pi(B) \sim^V_{\text{st}} B_{\text{skip}}$,*

where B_{skip} denotes any block with $\mathbf{skip}$ *as its only statement. Moreover,*

1. *if $v_{\text{active}} = B$ and B ends with $\mathbf{goto}\, B_1, \ldots, B_n$, then for every terminating trace of the statements in $\pi(B)$ it holds that $v_{\text{active}} \in \{B_1, \ldots B_n\}$ in the final store of the trace;*
2. *if $v_{\text{active}} \neq B$, then the value of v_{active} is the same for every store in every trace of the statements in $\pi(B)$.*

Proof. Let σ be a store of P and observe that $\sigma\!\restriction_V = \sigma$. Moreover, observe that the only trace of $\mathbf{skip}$ is the terminating trace σ.

Assume ss is defined as

$$\begin{aligned}&v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,;\\ &v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;\end{aligned}$$

We consider $v_{\text{active}} = B$ and $v_{\text{active}} \neq B$ in turn, where we assume $\hat{\sigma}$ is a store of $\pi(P)$ with $\hat{\sigma}\!\restriction_V = \sigma$.

Original form	Predicated form
B : ss $\quad$ **goto** $B_1, \ldots, B_n$; (B *is not* the last node of any loop according to the sort order)	B : $\pi(ss)$ $v_{\text{next}} :\in \{B_1, \ldots, B_n\}$; $v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}$; **goto** $\mathit{next}(B)$;
B : ss $\quad$ **goto** $B_1, \ldots, B_n$; (B *is* the last node of some loops according to the sort order)	B : $\pi(ss)$ $v_{\text{next}} :\in \{B_1, \ldots, B_n\}$; $v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}$; **goto** $B^{L_1}_{\text{back}}, \ldots, B^{L_n}_{\text{back}}, B_{\text{exit}}$; $B^{L_1}_{\text{back}}$: **assume** $v_{\text{active}} = B^{L_1}_{\text{head}}$; **goto** $B^{L_1}_{\text{head}}$; $\vdots$ $B^{L_n}_{\text{back}}$: **assume** $v_{\text{active}} = B^{L_n}_{\text{head}}$; **goto** $B^{L_n}_{\text{head}}$; B_{exit} : **assume** $\bigwedge_{i=1}^{n}(v_{\text{active}} \neq B^{L_i}_{\text{head}})$; **goto** $\mathit{next}(B)$;

Table 4: Predication of blocks

1. Let $v_{\text{active}} = B$. The only maximal traces of ss not terminating as *infeasible* are the traces $\hat{\sigma}\,\hat{\tau}_{B_i}\hat{v}_{B_i}$ with $0 \leq i \leq n$ where $\hat{\tau}_{B_i} = \hat{\sigma}[v_{\text{next}} \mapsto B_i]$ and $\hat{v}_{B_i} = \hat{\tau}_{B_i}[v_{\text{active}} \mapsto B_i]$. As v_{active} does not occur in P and as v_{next} is fresh, $\hat{\sigma}\restriction_V = \hat{\tau}_{B_i}\restriction_V = \hat{v}_{B_i}\restriction_V$. Hence, as σ is arbitrary, $ss \sim^V_{\text{st}}$ **skip** and stutter equivalence follows by Lemma B.7(1) and Propositions B.3 and B.4. The complete result follows once we observe that $\hat{v}_{B_i}(v_{\text{active}}) = B_i$ for all $0 \leq i \leq n$.
2. Let $v_{\text{active}} \neq B$. The only maximal traces of ss not terminating as *infeasible* are the traces $\hat{\sigma}\,\hat{\tau}_{B_i}\hat{v}_{B_i}$ with $0 \leq i \leq n$ where $\hat{\tau}_{B_i} = \hat{\sigma}[v_{\text{next}} \mapsto B_i]$ and $\hat{v}_{B_i} = \hat{\tau}_{B_i}[v_{\text{active}} \mapsto \hat{\tau}_{B_i}(v_{\text{active}})] = \hat{\tau}_{B_i}$. As v_{next} is fresh, $\hat{\sigma}\restriction_V = \hat{\tau}_{B_i}\restriction_V = \hat{v}_{B_i}\restriction_V$. Hence, as σ is arbitrary, $ss \sim^V_{\text{st}}$ **skip** and stutter equivalence follows Lemma B.7(2) and Propositions B.3 and B.4. The complete result follows once we observe that the only assignment to v_{active} in $\pi(B)$ occurs in the last statement of ss and that $\hat{v}_{B_i}(v_{\text{active}}) = \hat{\sigma}(v_{\text{active}})$. □

We also have the following:

Lemma B.9. *Let P be a sequential program with variables V. For every block B of P that occurs last in the sort order for loops $L_1, \ldots, L_n$*

1. *if $v_{\text{active}} = B^{L_i}_{\text{head}}$ for some $0 \leq i \leq n$, then $B^{L_i}_{\text{back}} \sim^V_{\text{st}} B_{\text{skip}}$ and every trace of B_{exit} is* infeasible*;*
2. *if $v_{\text{active}} \neq B^{L_i}_{\text{head}}$ for all $0 \leq i \leq n$, then $B_{\text{exit}} \sim^V_{\text{st}} B_{\text{skip}}$ and every trace of $B^{L_i}_{\text{back}}$ is* infeasible*,*

where B_{skip} denotes any block with **skip** *as its only statement. Moreover, in neither case is the value of v_{active} changed.*

Proof. Let σ be a store of P and observe that $\sigma{\upharpoonright_V} = \sigma$. Moreover, observe that the only trace of **skip** is the terminating trace σ. We consider the two cases in turn, where $\hat\sigma$ is a store of $\pi(P)$ such that $\hat\sigma{\upharpoonright_V} = \sigma$ and

$$e \doteq \bigwedge\nolimits_{i=1}^{n}(v_{\text{active}} \neq B^{L_i}_{\text{head}})\,.$$

1. If $v_{\text{active}} = B^{L_i}_{\text{head}}$ for some $0 \le i \le n$, then $\hat\sigma(v_{\text{active}} = B^{L_i}_{\text{head}})$ holds and the only trace of $\mathbf{assume}\, v_{\text{active}} = B^{L_i}_{\text{head}}$ is the terminating trace $\hat\sigma$. The only trace of $\mathbf{assume}\, e$ is *infeasible*. As $\hat\sigma{\upharpoonright_V} = \sigma$, the result follows.
2. If $v_{\text{active}} \neq B^{L_i}_{\text{head}}$ for all $0 \le i \le n$, then $\hat\sigma(e)$ holds and the only trace of $\mathbf{assume}\, e$ is the terminating trace $\hat\sigma$. For all $1 \le i \le n$, the only trace of $\mathbf{assume}\, v_{\text{active}} = B^{L_i}_{\text{head}}$ is *infeasible*. As $\hat\sigma{\upharpoonright_V} = \sigma$, the result follows. □

Stutter Equivalence of Sequential Programs. Predication of a sequential program P proceeds by (1) sorting all blocks as described earlier, (2) predicating each block, and (3) inserting the assignment $v_{\text{active}} := \mathit{Start}$ at the beginning of $\pi(\mathit{Start})$.

The addition of $v_{\text{active}} := \mathit{Start}$ ensures that the statements from $\pi(\mathit{Start})$ are always reduced first. Observe that if $\pi(\mathit{Start})$ is part of a loop, it can only be the head and, hence, by the construction in the second row of Table 4, it is guaranteed that $v_{\text{active}} = \mathit{Start}$ in case execution returns to $\pi(\mathit{Start})$. Call $\pi(\mathit{Start})$ with $v_{\text{active}} := \mathit{Start}$ at the beginning simply $\pi(\mathit{Start})$ from here on. We have:

Lemma B.10. *Let P be a sequential program with variables V. Then, $\pi(\mathit{Start}) \sim^V_{\text{st}} \mathit{Start}$. Moreover, if Start ends with* $\mathbf{goto}\, B_1, \ldots, B_n$*, then for every terminating trace of the statements of $\pi(\mathit{Start})$ it holds that $v_{\text{active}} \in \{B_1, \ldots B_n\}$ in the final store of the trace.*

Proof. Let σ be a store of P and let $\hat\sigma$ be a store of $\pi(P)$ such that $\hat\sigma{\upharpoonright_V} = \sigma$. The only trace of $v_{\text{active}} := \mathit{Start}$ is the terminating trace $\hat\sigma\,\hat\tau$ with $\hat\tau = \hat\sigma[v_{\text{active}} \mapsto \mathit{Start}]$. Hence, as $\hat\sigma{\upharpoonright_V} = \hat\tau{\upharpoonright_V} = \sigma$ and σ is arbitrary, we have $(v_{\text{active}} := \mathit{Start}) \sim^V_{\text{st}} \mathbf{skip}$. Reduction of the remaining of $\pi(\mathit{Start})$ starts with $v_{\text{active}} = \mathit{Start}$ and is, thus, stutter equivalent to Start by Lemma B.8, where $v_{\text{active}} \in \{B_1, \ldots B_n\}$ in the final store of each terminating trace. The complete result now follows by Proposition B.3. □

We can now state the main theorem of this section:

Theorem B.11. *If P is a sequential program with variables V, then $\pi(P) \sim^V_{\text{st}} P$.*

To prove the theorem, we need the following observation, which is the essence of Lemma 5.6:

Lemma B.12. *Let P be a sequential program with variables V. For each block B, if σ is a store of P and $\hat\sigma$ is a store of $\pi(P)$ such that $\hat\sigma{\upharpoonright_V} = \sigma$ and $\hat\sigma(v_{\text{active}}) = B$, then*

1. *if the reduction of B is immediately followed by reduction of a block C, then there exists a reduction of $\pi(B)$ such that $v_{\text{active}} = C$ at the end of $\pi(B)$ and eventually $\pi(C)$ is reduced. Moreover, every other reduction of $\pi(B)$ with $v_{\text{active}} = C$ at the end of $\pi(B)$ is* infeasible*;*

2. *if the reduction of $\pi(B)$ ends with $v_{\text{active}} = C$, then there exists a reduction of B that is immediately followed by the reduction of a block C.*

Proof. We consider each of the cases in turn.

1. If the reduction of B is immediately followed by reduction of C, then B ends in $\mathbf{goto} \cdots C \cdots$ Hence, by Lemma B.8 (and Lemma B.10 in case $B = \mathit{Start}$), there are terminating traces of the statements of B and, hence, reductions of B such that $v_{\text{active}} = C$ at the end of $\pi(B)$.
   Consider the the CFG of P, clearly there is an edge from B to C. If the edge is *not* a back edge, then the first requirement on the sort order of blocks in P ensures that $B \leq C$. By construction of the $\mathbf{goto}$s in $\pi(P)$ and the observations regarding v_{active} in Lemmas B.8 and B.9 (Lemmas B.10 and B.9 in the case of $B = \mathit{Start}$), the result follows.
   If the edge from B to C is a back edge, say of a loop L_i, then it follows by the same reasoning as above that eventually the block of L_i that is sorted last is reduced. By definition of $B^{L_i}_{\text{back}}$ and Lemma B.9, the result now follows.
2. Immediate by Lemma B.8 (Lemma B.10 in case $B = \mathit{Start}$), reasoning by contradiction. □

We can now prove Theorem B.11:

Proof. By Lemma B.12 we have for every reduction of P reducing in order the blocks $B_1, B_2, \ldots$ that there exists a reduction of $\pi(P)$ such that the value of v_{active} is in turn equal to $B_1, B_2, \ldots$, and vice versa. Reasoning by contradiction, it follows by Proposition B.3 and Lemmas B.8, B.9, and B.10 that for every trace of P we can construct a trace of $\pi(P)$ that is stutter equivalent, and vice versa. □

B.2 Procedure Calls

Define the *main routine* of a sequential program as the the set of blocks that occur in the CFG that has Start as the root node. We extend our kernel programming language with procedures. Grammar-wise this means that $\mathit{Program}$s will be defined as follows [9]:

$$\begin{aligned}
\mathit{Program} &::= \mathit{Procedure}^* \ \mathit{Block}^+ \\
\mathit{Procedure} &::= \mathbf{proc}\, \mathit{ProcId}(\mathbf{in} : \mathit{Var}^*, \mathbf{out} : \mathit{Var}^*)\ \mathit{BlockId}\,; \\
\mathit{Block} &::= \mathit{BlockId} : \mathit{Stmts}\ \mathit{Transfer}\,; \\
\mathit{Transfer} &::= \mathbf{return} \mid \mathbf{goto}\, \mathit{BlockId}^+
\end{aligned}$$

where we assume that the CFGs of all procedures and of main routine are disjoint. That is, for two distinct procedures defined by $\mathit{BlockId}$s B and C, we have that the blocks reachable from B are disjoint from the blocks reachable from C and that both are disjoint from the blocks reachable from Start.

For each procedure, $\mathbf{in}$ specifies a list of input parameters and $\mathbf{out}$ specifies a list of output parameters, both lists should be without duplicates. Both $\mathbf{return}$ and $\mathbf{goto}$ *transfer* control to some other block, where it is assumed that $\mathbf{return}$ does not occur in the main routine.

Remark B.13. We disallow $\mathbf{return}$ from occurring in the main routine, as this avoids having to deal with a host of special cases. Any program that does include a $\mathbf{return}$ in the main routine is easily rewritten to one without: Simply replace the $\mathbf{return}$ by a $\mathbf{goto}\, End$.

Observe that, symmetrically to disallowing $\mathbf{return}$ in the main routine, a program cannot terminate from within a procedure: The End block is part of the main routine and, hence, cannot occur in the CFG of any procedure. Again, this avoids having to deal with a host of special cases. Any program which does allow termination from within a procedure can be rewritten into a program which does not exhibit this behaviour, albeit this is a harder than in the case of a $\mathbf{return}$ occurring in the main routine.

Statements are defined as before, except that we also have **call**-statements:

$$\mathit{Var}^* := \mathbf{call}\, \mathit{ProcId}(\mathit{Expr}^*)$$

where the number of expressions is equal to the number of input parameters of the procedure and where the number of variables on the left of the assignment is equal to the number of output parameters. The variables are assigned values upon return from the procedure; no duplicates may occur among the variables and all variables must be thread-private.

Semantics. We describe the operational and denotational semantics of programs with procedures.

Operational Semantics. To give an operational meaning to procedure calls, we replace the private stores in the operational semantics from Figs. 4a and 4b by *stacks*. A stack is defined as a list of *stack frames*, where $S \cdot F$ denotes a stack with the frame F on top. Denoting a finite sequence of variables $v_1, \dots, v_n$ by $\boldsymbol{v}$, a stack frame is a tuple $(\sigma, \boldsymbol{v}, \boldsymbol{v_{\mathrm{out}}}, b)$ with (a) σ a store that maps the *local variables* of a procedure to values, (b) $\boldsymbol{v}$ the names of variables that should be assigned values upon return from the procedure, (c) $\boldsymbol{v_{\mathrm{out}}}$ the names of output parameters of the procedure, and (d) b a program fragment, which defines how reduction must continue upon return from the current procedure. Observe that a stack frame does not explicitly map input parameters to values. Instead, input parameters are considered to be local variables of a procedure. Output parameters are also local variables. Properly returning the values of output parameters is achieved by retaining in the stack frame the names of the parameters and the variables they need to be assigned to.

As before reduction of threads and sequential programs starts from $Start$, but the initial private stores are replaced by stacks containing a single stack frame $(\sigma, \varepsilon, \varepsilon, \varepsilon)$ where σ is a store that maps to values the *global variables* of the thread/sequential program and where the first two occurrences of ε denote an empty sequence of variables and where the last occurrence of ε denotes an empty sequence of statements. The empty sequence of statements in the stack frame will never be reduced, as we assume that $\mathbf{return}$ does not occur in the main routine.

We denote the simultaneous update of multiple variables by $\sigma[\boldsymbol{v} \mapsto \boldsymbol{val}]$, where we assume that the variables in $\boldsymbol{v}$ are all distinct. Projecting a stack $S_{\boldsymbol{\sigma}}$ to the list of its stores

$$\frac{(\mathbf{proc}\, p(\mathbf{in} : \boldsymbol{v}_{\mathbf{in}},\ \mathbf{out} : \boldsymbol{v}_{\mathbf{out}})\ B_p) \in P \qquad \tau_t \text{ a store} \qquad F = (\boldsymbol{v}, \boldsymbol{v}_{\mathbf{out}}, b, \tau_t[\boldsymbol{v}_{\mathbf{in}} \mapsto \boldsymbol{\sigma}(\boldsymbol{e})])}{P \vdash \langle \sigma, S_{\boldsymbol{\sigma}_t}, \boldsymbol{v} := \mathbf{call}\ p(\boldsymbol{e})\ ;\ b\rangle \overset{(\sigma,\boldsymbol{\sigma}_t)}{\to} \langle \sigma, S_{\boldsymbol{\sigma}_t} \cdot F, B_p\rangle}\ \textsc{Call}$$

$$\frac{F = (\boldsymbol{v}, \boldsymbol{v}_{\mathbf{out}}, b, \tau_t) \qquad \boldsymbol{val} = (\sigma, \boldsymbol{\sigma}_t \cdot \tau_t)(\boldsymbol{v}_{\mathbf{out}})}{P \vdash \langle \sigma, S_{\boldsymbol{\sigma}_t} \cdot F, \mathbf{return}\ ;\rangle \overset{(\sigma,\boldsymbol{\sigma}_t \cdot \tau_t)}{\to} \langle \sigma, S_{\boldsymbol{\sigma}_t}[\boldsymbol{v} \mapsto \boldsymbol{val}], b\rangle}\ \textsc{Return}$$

Fig. 8: Operational semantics of procedure calls and returns

$\boldsymbol{\sigma} = \sigma_1 \cdot \ldots \cdot \sigma_n$, we denote by $\boldsymbol{\sigma}(\boldsymbol{e})$ evaluation of the expressions $\boldsymbol{e}$ under $\boldsymbol{\sigma}$, where the global variables are evaluated according to σ_1 and the local variables according to σ_n (all other stores on the stack are inaccessible during evaluation). By $S_{\boldsymbol{\sigma}}[\boldsymbol{v} \mapsto \boldsymbol{val}]$ with $\boldsymbol{\sigma} = \sigma_1 \cdot \ldots \cdot \sigma_n$ we denote the simultaneous update of the (thread-private) global variables in σ_1 and the (thread-private) local variables in σ_n as specified by $\boldsymbol{v}$. In case of evaluation and updates, we assume that, if v is the name of *both* a local and a global variable, then the local variable v *hides* the global variable v.

The operational semantics of the kernel language extended with procedures is now defined as in Figs. 4a and 4b, where each store σ replaced by a stack and where each non-empty $\to$-label becomes the projection to the list of stores of the stack that occurs on the left of the arrow. The additional rules for **call** and $\mathbf{return}$ are given in Fig. 8. Observe that CALL is defined on the block level and not on the (possibly expected) statement level, because we need to take into account the continuation after the return from the procedure.

Denotational Semantics. The denotational semantics of kernels and sequential programs with procedures is defined as earlier, where thread-private stores are replaced by lists of stores. Observe that Propositions B.3 and B.4 still hold under introduction of procedures and when replacing stores by lists of stores.

Given a list of stores $\boldsymbol{\sigma}$ and a set of variables V, we define the *restriction* of $\boldsymbol{\sigma}$ to V, denoted $\boldsymbol{\sigma}\!\upharpoonright_V$, by restricting each store in $\boldsymbol{\sigma} = \sigma_1 \cdot \ldots \cdot \sigma_n$ to V, i.e., $\boldsymbol{\sigma}\!\upharpoonright_V = \sigma_1\!\upharpoonright_V \cdot \ldots \cdot \sigma_n\!\upharpoonright_V$. Stutter equivalence is defined as before, mutatis mutandis.

Predication of Procedures. To predicate programs with procedures, we make the following two assumptions with regard to the programs we wish to predicate (to avoid a host of special cases):

1. each procedure call occurs in block of its own, i.e. the body of a block in which a call to a procedure p occurs always has the form $\boldsymbol{v} := \mathbf{call}\, p(\boldsymbol{e})\ ;\ t$, where t only transfers control.
2. Each procedure has a unique block in which $\mathbf{return}$ occurs and the sequence of statements in the body of this block is *empty*.

A program is easily transformed into one which satisfies the above requirements.

Blocks are sorted *per* procedure following the earlier requirements. We require that the unique block with $\mathbf{return}$ is always sorted last to avoid premature return from procedures. Moreover, we assume that blocks with **call**-statements are not sorted last among the blocks in a loop. If a block with a call does occur last, then we replace the $\mathbf{goto}\, B_1, \ldots, B_n$ of the block by $\mathbf{goto}\, B_{\text{last}}$ with $\mathbf{goto}\, B_{\text{last}}$ fresh and we define

$$B_{\text{last}} \;:\; \mathbf{goto}\, B_1, \ldots, B_n\,.$$

Observe that B_{last} will be part of the loop and sorted after the block with the **call**.

Predication replaces each procedure definition

$$\mathbf{proc}\, p(\mathbf{in} : \boldsymbol{v_{in}}, \mathbf{out} : \boldsymbol{v_{out}})\ B_p$$

with

$$\mathbf{proc}\, p(\mathbf{in} : \boldsymbol{v_{in}}, v_{\text{active}}, \mathbf{out} : \boldsymbol{v_{out}})\ B_p\,.$$

That is, we specify an input parameter for the initial value of v_{active} in p. Adding v_{active} as an input parameter avoids having to modify the initial block B_p of every procedure p in the same way we modified Start. Moreover, adding v_{active} as an input parameter will be of use in our treatment of our lock-step programs — it allows us to call a procedure as a no-op in case it does not need to be called.

Blocks with procedure calls are now predicated as in the top row of Table 5, where B_{call} and B_{nocall} are fresh. The block of a procedure in which the **return** occurs, which is unique and contains only a **return** by the restrictions from above, is simply copied during predication, as depicted in the bottom row of Table 5. Observe that reduction of the block is not restricted by the value of v_{active}. Hence, any sort order in which the **return** block does not occur last will result in premature return from the procedure.

Original form	Predicated form
$B \;:\; \boldsymbol{v} := \mathbf{call}\, p(\boldsymbol{e})\,;$ $\quad\mathbf{goto}\, B_1, \ldots, B_n\,;$	$B \;:\; \mathbf{goto}\, B_{\text{call}}, B_{\text{nocall}}\,;$ $B_{\text{call}} \;:\; \mathbf{assume}\, v_{\text{active}} = B\,;$ $\quad \boldsymbol{v} := \mathbf{call}\, p(\boldsymbol{e}, B_p)\,;$ $\quad v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,;$ $\quad v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$ $\quad \mathbf{goto}\, \mathit{next}(B)\,;$ $B_{\text{nocall}} \;:\; \mathbf{assume}\, v_{\text{active}} \neq B\,;$ $\quad \mathbf{goto}\, \mathit{next}(B)\,;$
$B \;:\; \mathbf{return}\,;$	$B \;:\; \mathbf{return}\,;$

Table 5: Predication of procedure calls and returns

Predication of a sequential program P with procedures now proceeds in five steps: (1) sort the blocks of the main routine, (2) sort the blocks of each procedure, with the block from the procedure definition sorted first and with the unique block with the **return**-statement sorted last, (3) change the definition of each procedure, adding v_{active} as input parameter, (4) predicate each block as in Tables 4 and 5, with statements predicated as in Table 1, and (5) insert the assignment $v_{\text{active}} := \mathit{Start}$ at the beginning of the predicated Start block.

Stutter Equivalence of Sequential Programs with Procedures. Ignoring again the shared store and observing that the results below go through if a shared store is present, mutatis mutandis, we next prove stutter equivalence for sequential programs with procedures.

Let the variables of a program be precisely those variables that either occur (a) in the main routine, (b) as input or output parameters of a procedure, or (c) in the blocks of procedures. Replaying the proofs of Lemmas B.6, B.7, and B.8, with the stores in the proofs replaced by stacks, we obtain:

Lemma B.14. *Let P be a sequential program with variables V. For every block B of P* without *any* $\mathbf{call}$-*statement*

1. *if* $v_{\text{active}} = B$*, then* $\pi(B) \sim_{\text{st}}^{V} B$*;*
2. *if* $v_{\text{active}} \neq B$*, then* $\pi(B) \sim_{\text{st}}^{V} B_{\text{skip}}$*,*

where B_{skip} *denotes any block consisting of a single* $\mathbf{skip}$ *statement. Moreover,*

1. *if* $v_{\text{active}} = B$ *and* B *ends with* $\mathbf{goto}\, B_1, \dots, B_n$*, then for every terminating trace of the statements in* $\pi(B)$ *it holds that* $v_{\text{active}} \in \{B_1, \dots B_n\}$ *in the final list of stores of the trace;*
2. *if* $v_{\text{active}} \neq B$*, then the value of* v_{active} *is the same for every list of stores in every trace of the statements in* $\pi(B)$*.*

In similar vein, we can replay Lemma B.9, where we should consider loops in both the main routine and in all procedures. We also have:

Lemma B.15. *Let P be a sequential program with variables V. For every block B of P* with *any* $\mathbf{call}$-*statement such that* B *ends in* $\mathbf{goto}\, B_1, \dots, B_n$ *and where* s *is defined as* $\mathbf{assume}\, v_{\text{active}} = B$; *and where* b *is defined as*

$$\begin{array}{l} v_{\text{next}} :\in \{B_1, \dots, B_n\} \,; \\ v_{\text{active}} := (v_{\text{active}} = B) \;?\; v_{\text{next}} : v_{\text{active}} \,; \\ \mathbf{goto}\, \mathit{next}(B) \,; \end{array}$$

1. *if* $v_{\text{active}} = B$*, then* $s \sim_{\text{st}}^{V} \mathbf{skip}$ *and* $b \sim_{\text{st}}^{V} B_{\text{skip}}$ *and every trace of* B_{nocall} *is* infeasible*;*
2. *if* $v_{\text{active}} \neq B$*, then* $B_{\text{nocall}} \sim_{\text{st}}^{V} B_{\text{skip}}$ *and every trace of* s *is* infeasible*,*

where B_{skip} *denotes any block with* $\mathbf{skip}$ *as its only statement. Moreover,*

1. *if* $v_{\text{active}} = B$*, then for every terminating trace of the statement in* b *it holds that* $v_{\text{active}} \in \{B_1, \dots B_n\}$ *in the final list of stores of the trace;*
2. *if* $v_{\text{active}} \neq B$*, then the value of* v_{active} *is the same for every list of stores in every trace of the statements in* B_{nocall}*.*

Proof. Let $\boldsymbol{\sigma}$ be a list of stores of P and observe that $\boldsymbol{\sigma}\restriction_V = \boldsymbol{\sigma}$. Observe also that the only trace of $\mathbf{skip}$ is the terminating trace $\boldsymbol{\sigma}$. We consider the two cases in turn, where we assume that $\hat{\boldsymbol{\sigma}}$ is a list of stores of $\pi(P)$ with $\hat{\boldsymbol{\sigma}}\restriction_V = \boldsymbol{\sigma}$:

1. If $v_{\text{active}} = B$, then $\hat{\boldsymbol{\sigma}}(v_{\text{active}} = B)$ holds. Hence, (a) the only trace of s is the terminating trace $\hat{\boldsymbol{\sigma}}$, (b) the only maximal traces of the statement in b that are not *infeasible* are the terminating traces $\hat{\boldsymbol{\sigma}}\,\hat{\boldsymbol{\tau}}_{B_i}\,\hat{\boldsymbol{v}}_{B_i}$ with $1 \leq i \leq n$ where $\hat{\boldsymbol{\tau}}_{B_i} = \hat{\boldsymbol{\sigma}}[v_{\text{next}} \mapsto B_i]$ and $\hat{\boldsymbol{v}}_{B_i} = \hat{\boldsymbol{\tau}}[v_{\text{active}} \mapsto B_i]$, and (c) the only trace of $\mathbf{assume}\, v_{\text{active}} \neq B$ is *infeasible*. As v_{next} and v_{active} do not occur in P, we have $\hat{\boldsymbol{\sigma}}\restriction_V = \hat{\boldsymbol{\tau}}_{B_i}\restriction_V = \hat{\boldsymbol{v}}_{B_i}\restriction_V$. Thus, as $\hat{\boldsymbol{\sigma}}\restriction_V = \boldsymbol{\sigma}$, we also have $s \sim^V_{\text{st}}$ $\mathbf{skip}$ and $b \sim^V_{\text{st}} B_{\text{skip}}$. The result follows, once we observe that for all $0 \leq i \leq n$ we have $\hat{\boldsymbol{v}}_{B_i}(v_{\text{active}}) = B_i$.
2. If $v_{\text{active}} \neq B$, then $\hat{\boldsymbol{\sigma}}(v_{\text{active}} \neq B)$ holds. Hence, (a) the only trace of the statement $\mathbf{assume}\, v_{\text{active}} \neq B$ is the terminating trace $\hat{\boldsymbol{\sigma}}$ and (b) the only trace of s is *infeasible*. Thus, as $\hat{\boldsymbol{\sigma}}\restriction_V = \boldsymbol{\sigma}$, we have $B_{\text{nocall}} \sim^V_{\text{st}} B_{\text{skip}}$. The result follows, once we observe that v_{active} is not assigned to in B_{nocall}. □

Before we arrive at the main theorem of this section, observe that we can replay Lemma B.10 in the current setting; by our earlier assumptions, no $\mathbf{call}$-statement occurs in Start. Moreover, we have the following analogue of Lemma B.12, where π is inductively extended to lists of block bodies and where $S_{\boldsymbol{\sigma},\boldsymbol{\gamma}}$ denotes a stack which when projected to its stores and continuations yields, resp., $\boldsymbol{\sigma}$ and $\boldsymbol{\gamma}$.

Lemma B.16. *Let P be a sequential program with variables V. For every block B, if $S_{\boldsymbol{\sigma},\boldsymbol{\gamma}}$ is a stack of P and $S_{\hat{\boldsymbol{\sigma}},\hat{\boldsymbol{\gamma}}}$ is a stack of $\pi(P)$ such that $\hat{\boldsymbol{\sigma}}\restriction_V = \boldsymbol{\sigma}$, $\hat{\boldsymbol{\gamma}} = \pi(\hat{\boldsymbol{\gamma}})$, and $\hat{\boldsymbol{\sigma}}(v_{\text{active}}) = B$, then*

1. *if the reduction of B is immediately followed by reduction of a block C, then there exists a reduction of $\pi(B)$ such that $v_{\text{active}} = C$ at the end of $\pi(B)$ and eventually $\pi(C)$ is reduced. Moreover, every other reduction of $\pi(B)$ with $v_{\text{active}} = C$ at the end of $\pi(B)$ is* infeasible*;*
2. *if the reduction of $\pi(B)$ ends with $v_{\text{active}} = C$, then there exists a reduction of B that is immediately followed by the reduction of a block C.*

Proof. We consider each of the cases in turn.

1. Let the reduction of B immediately be followed by reduction of C. There are three cases to consider depending on the way in which control is transferred:
   - Let control be transferred by a $\mathbf{goto}$. In this case, we can replay the proof of Lemma B.12 per procedure, mutatis mutandis.
   - Let control be transferred by a $\mathbf{call}$ to a procedure p which first reduces a block C. In this case, $\pi(B)$ is either followed by reduction of B_{call} or B_{nocall}.
     If B_{call} is reduced, then the result follows by Lemma B.15 and the observation that $\pi(p)$ is called with v_{active} set to C.
     If B_{nocall} is reduced, then reduction ends in *infeasible*, again by Lemma B.15.
   - Let control be transferred by a $\mathbf{return}$. In this case, we have by definition of predication that the reduction of B is followed by a continuation of the form $\mathbf{goto}\, B_1, \ldots, B_n$. Hence, by Lemma B.15 and the assumption that $\hat{\boldsymbol{\gamma}} = \pi(\hat{\boldsymbol{\gamma}})$, we have that $v_{\text{active}} = C$ at the end of the continuation in predicated form.
     Consider the CFG of the caller of the procedure p we are returning from, clearly there must be an edge to C from the block, say B', with the call to p. Hence, the sort order on blocks in procedures ensures that $B' \leq C$. By construction

of the **goto**s, the assumption that no back edges have B' as source, and the observation regarding v_{active} in Lemma B.14, the result now follows.

2. Let the reduction of $\pi(B)$ end with $v_{\text{active}} = C$. There are again three cases to consider:
   – Let control be transferred by a **goto**. In this case, we can replay the proof of Lemma B.12 per procedure, mutatis mutandis.
   – Let control be transferred by a **call** to a procedure $\pi(p)$ with v_{active} set to C. In this case, it follows by definition of predication that p first reduces a block C and, hence, the result is immediate.
   – Let control be transferred by a **return**. In this case, we have by definition of predication that reduction of B is followed by a continuation of the form $\mathbf{goto}\, B_1, \ldots, B_n$. The result now follows by Lemma B.15 and the assumption that $\hat{\gamma} = \pi(\hat{\gamma})$. □

The main theorem of this section now follows by replaying the proof of Theorem B.11, mutatis mutandis, where we employ Lemma B.16 instead of Lemma B.12.

Theorem B.17. *If P is a sequential program with variables V, then $\pi(P) \sim^V_{\text{st}} P$.*

B.3 Stutter Equivalence of Interleaving Kernels

To ensure the correct operation of procedures in combination with barriers, we introduce an additional set of barrier variables:

– Every thread has a local variable v_{call} in each procedure (and the main routine). Each variable v_{call} is initialised as $(-,-)$. When a procedure p is called from a block B, v_{call} is set to (p, B), and it is reset to $(-,-)$ upon return from p.

Moreover, every barrier variable introduced previously is assumed to be local. The additional constraint ensures that different threads follow the same sequence of calls to reach a barrier. Introducing these variables avoids having to inline all procedures, as is usually done by compilers of GPU kernels. Moreover, it allows us to handle recursion properly, which is not easily done with inlining.

In addition to the above we replace the BARRIER$_{\text{F}}$ rule to properly handle non-terminating kernels:

$$\frac{\begin{array}{c}\exists\, t : T_{\vec{\sigma}}|_t = \langle \beta_t, \sigma_t, \mathbf{barrier}\, e_t;\, b_t\rangle \wedge (\sigma, \sigma_t)(e_t) \\ \neg\forall \text{ maximal } (P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \to^* s) : s = \bot \\ \neg\exists\, P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \to^* \langle \sigma, T'_{\vec{\sigma}}\rangle : \forall\, t : T'_{\vec{\sigma}}|_t = \langle \beta, \sigma_t, \mathbf{barrier}\, e_t;\, b_t\rangle \wedge (\sigma, \sigma_t)(e_t)\end{array}}{P \vdash \langle \sigma, T_{\vec{\sigma}}\rangle \overset{(\sigma,\vec{\sigma})}{\to} \mathcal{E}}\ \textsc{Barrier}_{\text{F}}$$

Hence, a kernel terminate with *error* in case (a) one thread has reached a barrier, (b) not all traces end in *infeasible* irrespective of the barrier behaviour, and (c) it is impossible to reach a barrier where all barrier variables agree. In the rule, $\to^*$ denotes the transitive, reflexive closure of $\to$. Note that the premise of BARRIER$_{\text{F}}$ codifies an instance of the halting problem and, hence, is in general undecidable. However, in case we have a kernel that (a) is terminating, (b) has at its last statement a barrier, and (c) is well-formed, it follows easily that the above rule can be replaced by our original rule from Fig. 4d, albeit some extra reduction steps might be needed to have all threads reach a barrier.

Stutter Equivalence of Interleaving Kernels. To prove stutter equivalence of a kernel P with variables V (which are *not* barrier variables) and the predicated kernel $\pi(P)$, it is essential to observe that stutter equivalence of sequential programs, i.e. Theorem B.17, continues to hold under introduction of the rule

$$\frac{\boldsymbol{v} \in V^* \qquad \boldsymbol{val} \in D^*}{P \vdash \langle S_{\boldsymbol{\sigma}}, b\rangle \overset{\sigma}{\to} \langle S_{\boldsymbol{\sigma}}[\boldsymbol{v} \mapsto \boldsymbol{val}], b\rangle} \text{ UPDATE}$$

where the number of variables in $\boldsymbol{v}$ is equal to the number of values in $\boldsymbol{val}$ and where V consists solely of variables from P (no barrier variables or fresh variables from $\pi(P)$ occur in V);

The above follows easily by replaying the proof of Theorem B.17. The UPDATE-steps in reductions are copied verbatim to the corresponding reductions of $\pi(P)$, and vice versa; no duplication or removal of such steps occurs. Observe that a reduction may now be infinite, because an infinite number of UPDATE-steps may occur at the end of the reduction. This should be taken into account when replaying the proof of Theorem B.17.

Theorem B.18. *If P is a kernel with variables V (not including barrier variables), then $\pi(P) \sim_{\text{st}}^{V} P$.*

Proof. Observe that adding a **skip**-statement before a **barrier**-statement in either a predicated and unpredicated kernel yields a kernel that is stutter equivalent to the original (predicated or unpredicated) kernel. Hence, it suffices to consider programs in which this additional **skip** is always present.

Let v_{state} be a fresh, shared variable and define proj_t for $1 \leq t \leq TS$ to be a map from reductions in (interleaving) kernels to reductions in sequential programs:

- proj_t maps $\langle \sigma, T_{\vec{\sigma}}\rangle$ to $\langle \sigma', S_{\boldsymbol{\sigma}_t}, b'_t\rangle$ where $T_{\vec{\sigma}}|_t = \langle S_{\boldsymbol{\sigma}_t}, b_t\rangle$ and with
  1. $\sigma'(v) = \sigma(v)$ for all $v \in V$,
  2. $\sigma'(v_{\text{state}}) = \langle \sigma, T_{\vec{\sigma}}\rangle[\bullet]_t$ for some constant $\bullet$,
  3. and b'_t identical to b_t but with (a) each **skip** preceding a barrier replaced by an **assume** that states that the premise of one of the BARRIER-rules is satisfied and (b) each **barrier** replaced by an **assert** that states that the premise of either the BARRIER$_{\text{SKIP}}$- or BARRIER$_{\text{S}}$-rule is satisfied (all appropriately adapted to the introduction of v_{state});
- proj_t maps a step in a reduction to:
  1. an application of the ASSUME$_{\text{T}}$-rule in case the THREAD$_{\text{B}}$-rule was employed to reduce a **skip**-statement before a barrier and t was responsible for the step;
  2. an application of an ASSERT-rule in case a BARRIER$_{\text{B}}$-rule was employed;
  3. the step in the premise of the applied rule in case either the THREAD$_{\text{B}}$- or the THREAD$_{\text{E,I}}$-rule was employed and t was responsible for the step;
  4. an application of the UPDATE-rule with $\boldsymbol{v}$ empty in case the THREAD$_{\text{T}}$-rule was employed and t was responsible for the step;
  5. an application of the END-rule in case the TERMINATION-rule was employed;
  6. an appropriate application of the UPDATE-rule, otherwise;

  in each case, σ is replaced by σ' as defined above.

Let $\Sigma \in \mathcal{D}(P)$ and $\mathcal{D}(\rho) = \Sigma$. If $1 \leq t \leq TS$, then, by the extension of Theorem B.17 with the SKIP-rule, we have $\mathcal{D}(\mathrm{proj}_t(\rho)) \sim_{\mathrm{st}}^{V \cup \{v_{\mathrm{state}}\}} \mathcal{D}(\rho')$ for some reduction ρ' of the *sequential* program $\pi(P)$, where barrier variables are not included in V and where the **skip** ; **barrier** is replaced as explained above for proj_t (all other statements are predicated as usual).

As UPDATE-steps are copied verbatim from $\mathrm{proj}_t(\rho)$ to ρ', the inverse of proj_t exists for ρ' and, hence, there exists a trace Σ' that is stutter equivalent to Σ for V and that corresponds to a reduction that reduces $\pi(\mathit{Start})$ for t and that reduces Start for all $t' \neq t$. By induction on the number of threads, applying the previous construction to each thread in turn, moving an increasing number of threads over to reduction of $\pi(\mathit{Start})$, it follows for every $\Sigma \in \mathcal{D}(P)$ that there exists a $\Sigma' \in \mathcal{D}(\pi(P))$ such that $\Sigma \sim_{\mathrm{st}}^{V} \Sigma'$.

By a symmetric argument, there exists a $\Sigma' \in \mathcal{D}(P)$ for every $\Sigma \in \mathcal{D}(\pi(P))$ such that $\Sigma \sim_{\mathrm{st}}^{V} \Sigma'$. Hence, $\pi(P) \sim_{\mathrm{st}}^{V} P$. $\square$

We have the following:

Theorem B.19. *Let P be kernel. A data race occurs in P iff a data race occurs in $\pi(P)$ where during reduction of neither of the two statements causing the data race it holds that $v_{\mathrm{active}} \neq B$ where B is the block in which the statement occurs.*

Proof. To start, observe that every shared variable occurring in P also occurs in $\pi(P)$, and vice versa (all variables freshly introduced during the predication are thread-private). Hence, any data race that may occur in P must involve variables that also occur in $\pi(P)$, and vice versa.

Suppose that P has a data race. Then, by definition, a reduction ρ of P exists with a data race. Employ the construction from the proof of Theorem B.18 to obtain a reduction ρ' of $\pi(P)$ with $\rho' \sim_{\mathrm{st}}^{V} \rho$ and V the non-barrier variables from P. By construction, we have for each block B that the variables accessed during reduction of B and $\pi(B)$ are identical in case $v_{\mathrm{active}} = B$. Moreover, no additional BARRIER$_{\mathrm{S}}$ is inserted in between accesses. Hence, ρ' exhibits the same conflict as ρ.

Suppose that $\pi(P)$ has a data race and consider the predicated form of the statements in Table 1. In case $v_{\mathrm{active}} \neq B$, observe for **call**-statements that the only reductions of B_{call}-blocks immediately terminate as *infeasible*. For any other block freshly introduced during the predication, we have that only thread-private variables occur and, hence, reduction of these blocks cannot introduce any data races. By a similar argument as for data races in P, it now follows that for every reduction ρ in $\pi(P)$ exhibiting some conflict, there exists a reduction ρ' in P that exhibits the same conflict. $\square$

We have thus proved Theorem 5.5, as this combines Theorems B.18 and B.19.

B.4 Lock-Step Execution

In this section, we prove our soundness and completeness result. Before we can do so, we need to extend the lock-step construction for blocks, as we extended the handling of loops above. Moreover, we need to explain how we handle procedures in the lock-step construction.

Lock-Step Construction for Blocks. The lock-step construction for blocks is presented in Table 6a, where $\phi(ss)$ denotes the lock-step construction for a sequence of statements ss, which is defined as $\phi(\varepsilon) = \varepsilon$ and $\phi(s \,;\, ss) = \phi(s) \,;\, \phi(ss)$, with $\phi(s)$ the lock-step construction for a statement s.

In case a block *is not* sorted last among the blocks in a loop (see the top row of Table 6a) we simply duplicate updating v_{active}.

In case a block *is* sorted last among blocks in loops $L_1, \ldots, L_k, \ldots, L_n$, where we may assume without loss of generality that $L_1 \subsetneq \cdots \subsetneq L_k \subsetneq \cdots \subsetneq L_n$, we need to check whether:

1. we need to jump back to the head of a loop L_i for one of the threads (otherwise that thread will exit the loop prematurely), and whether
2. if we do need to jump back, that we do not need to jump back to the head of a loop L_j with $1 \leq j < i$ for any thread (which would be a loop nested inside L_i).

In addition, we only continue to reduce the next block in the sort order in case all loops have been exited by all threads. The case is presented in the bottom row of Table 6a.

Lock-Step Construction for Procedures. For lock-step execution of procedures, we need to ensure that if one thread calls a procedure then all threads do so, even if not all threads would have normally called the procedure. Of course, if a thread would not have normally called a procedure, we must enforce that the behaviour of the thread in the lock-step program is stutter equivalent to skipping the procedure.

Assuming that the block containing the unique $\mathbf{return}$ from a procedure p is End_p (cf. Sect. B.2), the above can be achieved by changing the B_{call} block from Table 5 to:

$$\begin{aligned} B_{\text{call}} :\ & \boldsymbol{v}_{\text{out}} := \mathbf{call}\, p(\boldsymbol{e}, (v_{\text{active}} = B)\ ?\ B_p : \mathit{End}_p)\,; \\ & \boldsymbol{v} := (v_{\text{active}} = B)\ ?\ \boldsymbol{v}_{\text{out}} : \boldsymbol{v}\,; \\ & v_{\text{next}} :\in \{B_1, \ldots, B_n\}\,; \\ & v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,; \\ & \mathbf{goto}\, C\,; \end{aligned}$$

where $\boldsymbol{v}_{\text{out}}$ is a fresh variable. Observe that all statements in the called procedure p will effectively become no-ops, as the local v_{active} of the procedure is set to End_p and as the End_p block only returns.

Unfortunately, the above proposal for B_{call} is not stutter equivalent to the version from Table 5: If a recursive call to a procedure p occurs, then infinite recursion now occurs (calling p with End_p each time), while this might not have been the case in the original program. To avoid this problem, we ensure below that a procedure is only called in case it must be called by *at least* one thread.

The lock-step construction replaces each definition

$$\mathbf{proc}\, p(\mathbf{in} : \boldsymbol{v_{in}}, v_{\text{active}}, \mathbf{out} : \boldsymbol{v_{out}})\ B_p\,,$$

with

$$\mathbf{proc}\, p(\mathbf{in} : \langle \boldsymbol{v_{in}}_{,t}, v_{\text{active},t} \rangle_{t=1}^{TS}, \mathbf{out} : \langle \boldsymbol{v_{out}}_{,t} \rangle_{t=1}^{TS})\ B_p\,.$$

Thus, similarly to the lock-step construction for assignment statements, we introduce a copy v_t of each parameter v for each thread t.

Predicated form	Lock-step form
$B\ :\ ss$ $v_{\text{next}} :\in \{B_1,\ldots,B_n\}\,;$ $v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$	$B\ :\ \phi(ss)$ $\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle\{B_1,\ldots,B_n\}\rangle_{t=1}^{TS}\,;$ $\langle v_{\text{active},t}\rangle_{t=1}^{TS} := \langle(v_{\text{active},t} = B)\ ?\ v_{\text{next},t} : v_{\text{active},t}\rangle_{t=1}^{TS}\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$
$B\ :\ ss$ $v_{\text{next}} :\in \{B_1,\ldots,B_n\}\,;$ $v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$ $\textbf{goto}\ B^{L_1}_{\text{back}},\ldots,B^{L_n}_{\text{back}},B_{\text{exit}}\,;$	$B\ :\ \phi(ss)$ $\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle\{B_1,\ldots,B_n\}\rangle_{t=1}^{TS}\,;$ $\langle v_{\text{active},t}\rangle_{t=1}^{TS} := \langle(v_{\text{active},t} = B)\ ?\ v_{\text{next},t} : v_{\text{active},t}\rangle_{t=1}^{TS}\,;$ $\textbf{goto}\ B^{L_1}_{\text{back}},\ldots,B^{L_n}_{\text{back}},B_{\text{exit}}\,;$
$B^{L_1}_{\text{back}}\ :\ \textbf{assume}\ v_{\text{active}} = B^{L_1}_{\text{head}}\,;$ $\textbf{goto}\ B^{L_1}_{\text{head}}\,;$	$B^{L_1}_{\text{back}}\ :\ \textbf{assume}\ \bigvee_{t=1}^{TS}(v_{\text{active},t} = B^{L_1}_{\text{head}})\,;$ $\textbf{goto}\ B^{L_1}_{\text{head}}\,;$
$\vdots$	$\vdots$
$B^{L_k}_{\text{back}}\ :\ \textbf{assume}\ v_{\text{active}} = B^{L_k}_{\text{head}}\,;$ $\textbf{goto}\ B^{L_2}_{\text{head}}\,;$	$B^{L_k}_{\text{back}}\ :\ \textbf{assume}\ (\bigvee_{t=1}^{TS}(v_{\text{active},t} = B^{L_k}_{\text{head}})) \wedge (\bigwedge_{t=1}^{TS}\bigwedge_{i=1}^{k-1} v_{\text{active},t} \neq B^{L_i}_{\text{head}})\,;$ $\textbf{goto}\ B^{L_2}_{\text{head}}\,;$
$\vdots$	$\vdots$
$B^{L_n}_{\text{back}}\ :\ \textbf{assume}\ v_{\text{active}} = B^{L_n}_{\text{head}}\,;$ $\textbf{goto}\ B^{L_n}_{\text{head}}\,;$	$B^{L_n}_{\text{back}}\ :\ \textbf{assume}\ (\bigvee_{t=1}^{TS}(v_{\text{active},t} = B^{L_n}_{\text{head}})) \wedge (\bigwedge_{t=1}^{TS}\bigwedge_{i=1}^{n-1} v_{\text{active},t} \neq B^{L_i}_{\text{head}})\,;$ $\textbf{goto}\ B^{L_n}_{\text{head}}\,;$
$B_{\text{exit}}\ :\ \textbf{assume}\ \bigwedge_{i=1}^{n}(v_{\text{active}} \neq B^{L_i}_{\text{head}})\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$	$B_{\text{exit}}\ :\ \textbf{assume}\ \bigwedge_{t=1}^{TS}\bigwedge_{i=1}^{n}(v_{\text{active},t} \neq B^{L_i}_{\text{head}})\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$

(a) Blocks

Predicated form	Lock-step form
$B\ :\ \textbf{goto}\ B_{\text{call}}, B_{\text{nocall}}\,;$	$B\ :\ \textbf{goto}\ B_{\text{call}}, B_{\text{nocall}}\,;$
$B_{\text{call}}\ :\ \textbf{assume}\ v_{\text{active}} = B\,;$ $\boldsymbol{v} := \textbf{call}\ p(\boldsymbol{e}, B_p)\,;$ $v_{\text{next}} :\in \{B_1,\ldots,B_n\}\,;$ $v_{\text{active}} := (v_{\text{active}} = B)\ ?\ v_{\text{next}} : v_{\text{active}}\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$	$B_{\text{call}}\ :\ \textbf{assume}\ \bigvee_{t=1}^{TS}(v_{\text{active},t} = B)\,;$ $\langle \boldsymbol{v}_{\text{out},t}\rangle_{t=1}^{TS} := \textbf{call}\ p(\langle \phi(\boldsymbol{e})_t, (v_{\text{active},t} = B)\ ?\ B_p : \mathit{End}_p\rangle_{t=1}^{TS})\,;$ $\langle \boldsymbol{v}_t\rangle_{t=1}^{TS} = \langle (v_{\text{active},t} = B)\ ?\ \boldsymbol{v}_{\text{out},t} : \boldsymbol{v}_t\rangle_{t=1}^{TS}$ $\langle v_{\text{next},t}\rangle_{t=1}^{TS} :\in \langle\{B_1,\ldots,B_n\}\rangle_{t=1}^{TS}\,;$ $\langle v_{\text{active},t}\rangle_{t=1}^{TS} := \langle(v_{\text{active},t} = B)\ ?\ v_{\text{next},t} : v_{\text{active},t}\rangle_{t=1}^{TS}\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$
$B_{\text{nocall}}\ :\ \textbf{assume}\ v_{\text{active}} \neq B\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$	$B_{\text{nocall}}\ :\ \textbf{assume}\ \bigwedge_{t=1}^{TS}(v_{\text{active},t} \neq B)\,;$ $\textbf{goto}\ \mathit{next}(B)\,;$
$B\ :\ \textbf{return}\,;$	$B\ :\ \textbf{return}\,;$

(b) Procedures

Table 6: Lock-step construction for blocks and procedures

The lock-step construction for **call**-statements now combines the above proposal for B_{call} with a check that *at least* one thread needs to call the procedure. The construction is presented in the top row of Table 6b, where all thread-private variables are again duplicated for all threads (recall that all variables assigned to in the assignment of a call statement are thread-private).

As shown in the bottom row of Table 6b, a block with a **return** statement is left unchanged, as its only effect is to return from a procedure.

Soundness and Completeness. To prove soundness, we need to make one trivial change to our well-formed kernels. To see why a change is necessary, consider the following kernel:

```
$Start$ : **goto** $B_1, B_2$ ;
$B_1$    : **assume** *true* ;
           **barrier** ;
           **goto** $End$ ;
$B_2$    : **assume** *false* ;
           **goto** $End$ ;
```

If we apply our lock-step construction to this program, where we pick $B_1 \leq B_2$, then the condition in the **sync** that replaces the barrier will not evaluate to $true$ in case we assign B_2 to $v_{\text{active},t}$ for some thread t. Although this assignment actually only yields infeasible traces in the interleaving semantics (due to the second condition in the premise of the BARRIER$_{\text{F}}$-rule), this behaviour is not captured by our lock-step construction, due to the sort order of blocks.

The above problem can be dealt with by "inserting relevant assumes" immediately before each goto. Suppose we have a block B in a *well-formed* kernel that ends in **goto** $B_1, \ldots B_n$, where $e_1, \ldots, e_n$ are the guards of the assumes of blocks $B_1, \ldots B_n$ (a guard is equal to *true* in case no assume occurs). By predication, the **goto**-statement of B will be replaced by:

```
$v_{\text{next}} :\in \{B_1, \ldots, B_n\}$ ;
$v_{\text{active}} := (v_{\text{active}}) \; ? \; v_{\text{next}} : v_{\text{active}}$ ;
**goto** $C$ ;
```

Consider the following the replacement for the above code fragment:

```
$v_{\text{next}} :\in \{B_1, \ldots, B_n\}$ ;
**assume** $(v_{\text{active}} = B) \Rightarrow \bigwedge_{i=1}^{n} (v_{\text{next}} = B_i \Rightarrow e_i)$ ;
$v_{\text{active}} := (v_{\text{active}}) \; ? \; v_{\text{next}} : v_{\text{active}}$ ;
**goto** $C$ ;
```

Here, we insert an early check for guards of the blocks whose reduction can follow reduction of B. Observe that this only reduces the length of infeasible traces, but does not affect the predicated kernel in any other way. Also observe that by well-formedness any reduction terminating as *infeasible* still can be transformed into one that does not terminate as *infeasible*.

The lock-step construction for the above assume is as for any other assume:

$$\textbf{assume} \bigwedge_{t=1}^{TS} ((v_{\text{active},t} = B) \Rightarrow \bigwedge_{i=1}^{n} (v_{\text{next},t} = B_i \Rightarrow e_i)) \; ;$$

With the above transformation, the observed problem with barriers can no longer occur, as the check for infeasibility is effectively moved in front of the barrier.

Assuming from here on that our lock-step programs include an $\mathbf{assume}$-statement of the above kind between every assignment to v_{next} and v_{active}, we are now ready to prove soundness and completeness.

Soundness. We have the following with respect to data races:

Theorem B.20. *Let P be a well-formed kernel and let $\phi(P)$ be the program obtained by applying the lock-step construction to P. The occurrence of a data race in $\phi(P)$ implies the occurrence of a data race P.*

Proof. By Theorem B.19, it suffices to construct a reduction for $\pi(P)$ — the predicated version of the kernel P — with an appropriate data race.

Let ρ be a reduction of $\phi(P)$ and call a thread t *semi-active* in a step of ρ, if

1. whenever the step involves reduction of a statement from within a procedure p, then p was called with $v_{\text{active},t} \neq \mathit{End}_p$, and
2. whenever the step involves reduction a statement from a loop L_i or a block $B^{L_i}_{\text{back}}$, then $v_{\text{active},t} \in L_i$.

Call t *semi-inactive*, otherwise.

By an easy induction it follows that we can obtain a reduction of $\pi(P)$ by replacing each reduction step in ρ by a sequence of steps which, instead of reducing one of the statements or transfers of control from the right-hand columns of Tables 3a, 6a, and 6b, reduces the corresponding statement or transfer of control from the left-hand column for *each* semi-active thread.

Care should be taken of the following:

– For an assignment to a shared variable, the application of the ASSIGN-step of a thread t should be last among all active threads in case the existential quantifier in the premise of ψ_{T} was instantiated with t.
– For a thread that should not call a procedure p, the transfer of control

$$\mathbf{goto}\, B_{\text{call}}, B_{\text{nocall}}$$

should reduce to B_{nocall} instead of B_{call}. In addition, instead of reducing the statement $\mathbf{assume}\, v_{\text{active}} = B$, the statement $\mathbf{assume}\, v_{\text{active}} \neq B$ should be reduced.
– Reduction within B_{call}-blocks should be simplified in the obvious way.
– Each reduction of a $\mathbf{sync}$ statement should be appropriately replaced by either a BARRIER$_{\text{SKIP}}$- or BARRIER$_{\text{S}}$-step in case the expression of the $\mathbf{sync}$ evaluates to true and it should be replaced by a BARRIER$_{\text{F}}$-step in case the expression evaluates to false. That the replacement is possible follows by following claim:

Claim. For the expression an $\mathbf{sync}$-statement the following holds:
• if the expression evaluates to true, then the premise of either the BARRIER$_{\text{SKIP}}$- or the BARRIER$_{\text{S}}$-rule is satisfied, and

- if the expression evaluates to false, then the premise of BARRIER$_\text{F}$ is satisfied.

Proof. We prove the two parts of the claim in turn.

- Suppose the expression evaluates to true. In this case, observe that a thread cannot progress from a semi-inactive state to a semi-active state while reduction is taking place inside a loop, as no statements from outside the loop occurs in between the statements from the loop (cf. the block sort order). The same holds while a thread is inside a procedure that was called in a semi-inactive state. Hence, in case the expression of a **sync**-statement evaluates to true, either no thread has reached a barrier, or it holds that the loop counters and v_call variables are identical across threads and, hence, that all threads have reached the same barrier by the lock-step execution. Whence, the premise of either the BARRIER$_\text{S}$- or the BARRIER$_\text{S}$-rule is satisfied.
- Suppose the expression evaluates to false. In this case, reason by contradiction and, thus, also suppose that for the the reduction constructed up to the **sync**-statement either (a) every maximal extension ends in *infeasible*, or (b) can be extended to one that satisfies the premise of the BARRIER$_\text{S}$-rule. Consider a thread t with $v_{\text{active},t} \neq B$ upon reaching the **sync**-statement. We consider the two cases in turn.
  a. In case every maximal extension of the reduction is *infeasible*, the thread t must fail on the guard of the block that is chosen as alternative to the block with the barrier. However, due to the **assume**-statements introduced on page 38, it follows that the assumed interleaving reduction does not exist.
  b. In case we can extend the reduction to one that satisfies the premise of the BARRIER$_\text{S}$-rule, the thread t can only reach the barrier if either the thread loops or calls an appropriate procedure later. Hence, the values of some loop variables or v_call variables will differ from those of the threads that already reached the barrier during lock-step execution. Whence, the premise of the BARRIER$_\text{S}$-rule can never be satisfied and we can apply the BARRIER$_\text{F}$-rule. □

It now follows immediately by the above construction and the definition of data races for kernels that each occurrence of a data race in $\phi(P)$ implies the occurrence of a data race in $\pi(P)$ such that during reduction of neither of the two statements causing the data race it holds that (a) $v_\text{active} \neq B$ where B is the block in which the statement occurs and that (b) a BARRIER$_\text{S}$-step occurs in between the two statements. □

We immediately have the following corollary.

Corollary B.21. *Let P be a well-formed kernel and let $\phi(P)$ be the program obtained by applying the the lock-step construction to P. If P is race free, then $\phi(P)$ is race free.*

The reverse of the above corollary, i.e., completeness, does in general not hold. To see this, assume that each thread t has a private variable tid whose value is t and consider the following program, where v is a shared variable:

```
Start : goto $B_1, B_2$ ;
$B_1$    : assume $tid = 1$ ;
          goto $Start$ ;
$B_2$    : assume $tid \neq 1$ ;
          $v := 1$ ;
          goto $End$ ;
```

Thread 1 loops forever, while every other thread assigns 1 to v. In case of the interleaving semantics, reductions exist in which thread 1 is ignored initially and in which every other thread is reduced until termination. Obviously, any such reduction has a data race in case there are at least three threads, as multiple threads assign 1 to the shared variable v. In the case of the lock-step semantics, every thread must first complete all loop iterations. As thread 1 never completes all loop iterations, the assignment to v is never reached and, hence, no data race occurs.

The above counterexample can be considered rather pathological due to its non-terminating nature (see also [11,12]). Hence, we next consider terminating programs.

Soundness and Termination. Our main soundness result is as follows:

Theorem B.22 (Soundness). *Let P be a well-formed kernel and let $\phi(P)$ be the program obtained by applying the lock-step construction to P. If P is race free and terminating, then $\phi(P)$ is race free and terminating. Moreover, if no data races occur, then for every terminating reduction of P there exists a terminating reduction of $\phi(P)$ such that every shared variable v of P has the same value at the end of both reductions.*

To prove the theorem, we need the following observation:

Proposition B.23. *Let P be a kernel. If*

$$\langle \sigma, T_{\vec{\sigma}} \rangle \overset{(\sigma,\vec{\sigma})}{\rightarrow} \langle \tau, T_{\vec{\sigma}}[\langle S_{\boldsymbol{\tau}}, c\rangle]_t \rangle \overset{(\tau,\vec{\tau})}{\rightarrow} \langle \upsilon, T_{\vec{\tau}}[\langle S_{\boldsymbol{\upsilon}}, d\rangle]_{t'} \rangle$$

is race free, with $T_{\vec{\tau}} = T_{\vec{\sigma}}[\langle S_{\boldsymbol{\tau}}, c\rangle]_t$ and $1 \leq t \neq t' \leq TS$, and the second step is either a THREAD$_{\mathrm{B}}$-, THREAD$_{\mathrm{T}}$-, *or* BARRIER$_{\mathrm{SKIP}}$*-step, then there exists a race-free reduction*

$$\langle \sigma, T_{\vec{\sigma}} \rangle \overset{(\sigma,\vec{\sigma})}{\rightarrow} \langle \tau', T_{\vec{\sigma}}[\langle S_{\boldsymbol{\upsilon}}, d\rangle]_{t'} \rangle \overset{(\tau',\vec{\tau}')}{\rightarrow} \langle \upsilon, T_{\vec{\chi}}[\langle S_{\boldsymbol{\tau}}, c\rangle]_t \rangle$$

with $T_{\vec{\chi}} = T_{\vec{\sigma}}[\langle S_{\boldsymbol{\upsilon}}, d\rangle]_{t'}$ and $T_{\vec{\chi}}[\langle S_{\boldsymbol{\tau}}, c\rangle]_t = T_{\vec{\tau}}[\langle S_{\boldsymbol{\upsilon}}, d\rangle]_{t'}$.

Proof. Observe that only the THREAD$_{\mathrm{B}}$-, THREAD$_{\mathrm{T}}$-, and BARRIER$_{\mathrm{SKIP}}$-rules may have been employed as the first step in the two-step reduction. As the two steps do not race and as $t \neq t'$, the result is immediate by the fact that two threads cannot access each others' stacks. Observe by race-freeness that the premise of the rule applied in the second step of the original reduction are already satisfied in $\langle \sigma, T_{\vec{\sigma}} \rangle$ and that the premise of the rule applied in the first step of the original reduction are still satisfied after performing $\langle \sigma, T_{\vec{\sigma}} \rangle \rightarrow \langle \tau', T_{\vec{\sigma}}[\langle S_{\boldsymbol{\upsilon}}, d\rangle]_{t'} \rangle$. □

We can now prove Theorem B.22.

Proof. That $\phi(P)$ is race free follows by Corollary B.21. That $\phi(P)$ terminates follows by contradiction: If there exists an infinite and race free reduction for $\phi(P)$, then an infinite and race free reduction also exists for P by the construction in the proof of Theorem B.20.

For the second half of the theorem, let ρ be a terminating and race free reduction of P. Observe that before each $\textsc{Barrier}_{\textsc{s}}$-step in ρ only $\textsc{Thread}_{\textsc{b}}$- and $\textsc{Barrier}_{\textsc{skip}}$-steps occur and that after the last $\textsc{Barrier}_{\textsc{s}}$-step only $\textsc{Thread}_{\textsc{t}}$-steps and a $\textsc{termination}$-step occur. By the above proposition we may re-order the steps of ρ before each $\textsc{Barrier}_{\textsc{s}}$-step such that the eventual values of the shared variables are as in ρ. In fact, it is possible re-order the steps such that we obtain a reduction as the one constructed in the proof of Theorem B.20 — grouping reduction of the same statements and transfers of control from different threads. For suppose we cannot re-order in this way, then for one or more threads an insufficient number of steps occur before a certain barrier. However, in this case either (a) different threads have reached different barriers, (b) the number of loop iterations differs between threads, or (c) the sequence of procedure calls used to reach the barrier differs between threads. In each of these cases the premise of $\textsc{Barrier}_{\textsc{s}}$ is not satisfied, contradiction. The complete result now follows by induction applying the construction from Theorem B.20 in reverse. That the $\textsc{Barrier}_{\textsc{skip}}$- and $\textsc{Barrier}_{\textsc{s}}$-steps can be replaced by $\textsc{Sync}$-steps follows by similar reasoning as in the proof of the claim on page 39. □

Completeness. We have the following:

Theorem B.24 (Completeness). *Let P be a well-formed kernel and let $\phi(P)$ be the program obtained by applying the lock-step construction of P. If $\phi(P)$ is race free and terminating, then P is race free and terminating. Moreover, if no data races occur, then for every terminating reduction of $\phi(P)$ there exists a terminating reduction of P such that every shared variable v of P has the same value at the end of both reductions.*

To prove the theorem, we need the next observation, where we say that a data race with respect to a variable v is called:

- a *write-write conflict*, in case both threads update v;
- a *read-write conflict*, in case one thread uses v to evaluate an expression and later along ρ another thread updates v;
- a *write-read conflict*, in case one thread updates v and later along ρ another thread uses v to evaluate an expression.

Proposition B.25. *Let P be a well-formed kernel. If $\rho \cdot \tau$ is a (partial) reduction of P where τ consists of two steps and has a data race, then there exists a (partial) reduction $\rho \cdot \tau'$ with τ' consisting of at most two steps such that the first step of τ' reduces the same statement in the same thread as the second step of τ and either*

1. *τ' consists of a single step and terminates with* error*, or*
2. *τ' has a data race and the second step of τ' reduces the same statement in the same thread as the first step of ρ.*

Proof. Observe that the two steps in τ originate from different threads, otherwise no data race occurs in τ.

Suppose the data race in τ is a write-write conflict. In this case, the statements reduced in τ are either assignments, havocs, returns from a procedures, or a combination of two thereof. As the steps originate from different threads, it is easily seen that the statements can be reduced in reverse order. No *error* can be introduced, as neither of the statements is an **assert** statement. A write-write conflict still occurs after reversal, as the updated variables do not change.

Suppose the data race in τ is either a read-write or a write-read conflict. In this case, the write part of the conflict originates with either an assignment, a havoc, or a return from a procedure; the read part of the conflict originates with either an assignment, a call of a procedure, or an **assert** statement. In case the read part originates with an assignment or a call of a procedure, the reduction of the two statements can be reversed and a conflict still occurs, as no other variable is updated during the write part of the conflict. In case the read part originates with an **assert**, either the same reasoning as for the assignment applies, or the **assert** becomes the first statement that is being reduced and we terminate with *error*.

Note that **assume** statements do not need to be considered, as we have by well-formedness of P that no global variables occur in such statements. □

We can now prove Theorem B.24.

Proof. We reason by contradiction. Thus, suppose P is not terminating or race free. Then, either (a) P is race free and there exists a reduction ρ that is infinite or terminates with status *error*, or (b) there exists a reduction ρ of P with a data race.

In case of (a), employ Proposition B.23 to re-order the steps in ρ to obtain a reduction where the steps have the same order as those of the reduction constructed in the proof of Theorem B.20 — grouping reduction of the same statements and transfers of control from different threads. Before each BARRIER$_S$-step, it suffices to re-order steps, by reasoning as in the proof of Theorem B.22. Following the last BARRIER$_S$-step, steps may need to be added for some threads (or removed when dealing with the BARRIER$_F$-rule) to ensure we conform to the grouping from the theorem (this may turn an infinite reduction into an *error* reduction or lead to an earlier *error*, but this is, of course, of no consequence). By assumption, the additional steps cannot introduce any data races. By applying the construction from Theorem B.20 in reverse (see also the proof of Theorem B.22), we obtain for $\phi(P)$ an infinite reduction or an *error* reduction, contradiction.

In case of (b), employ Propositions B.23 and B.25 to re-order the steps of ρ to construct a (partial) reduction which either terminates with *error* or has a data race, and in which the steps have the same order as those of the reduction constructed in the proof of Theorem B.20 — grouping reduction of the same statements and transfers of control from different threads. Following the last BARRIER$_S$-step, steps may need to be added for some threads (or removed when dealing with the BARRIER$_F$-rule) to ensure we conform to the grouping from the theorem (this may turn an infinite reduction into an *error* reduction or lead to an earlier *error*, but this is, of course, of no consequence). By applying the construction from Theorem B.20 in reverse, we now obtain for $\phi(P)$ a (partial) reduction which either terminates with *error* or has a data race. By well-formedness of P, this partial reduction can be extended to a maximal reduction of

$\phi(P)$ which is not *infeasible* and which either terminates with *error* or has a data race, contradiction. □

Combining Theorems B.22 and B.24 we obtain Theorem 5.2.

C 2-Threaded Abstraction

Although considering lock-step programs instead of kernels offers the potential to significantly reduce the verification state space, due to the removal of interleavings, the space may still be too large to make verification feasible, due to the large number of threads which usually execute a GPU kernel. Observe, however, that a data race always involves an interaction between *two* threads, which leads to the idea of abstracting away all threads of a kernel except for two [7].

Definition C.1 (2-Threaded Abstraction). *Let P be a kernel. If $1 \leq t_1 < t_2 \leq TS$, then the* 2-threaded abstraction of P with respect to t_1 and t_2, *denoted P_{t_1,t_2} is the kernel P where each barrier havocs every shared variable and with all reductions starting from*

$$\langle \sigma, \langle S_{\boldsymbol{\sigma}_{t_1}}, \mathit{Start}\rangle, \langle S_{\boldsymbol{\sigma}_{t_2}}, \mathit{Start}\rangle\rangle$$

instead of from

$$\langle \sigma, \langle S_{\boldsymbol{\sigma}_1}, \mathit{Start}\rangle, \ldots, \langle S_{\boldsymbol{\sigma}_{TS}}, \mathit{Start}\rangle\rangle \, ,$$

where $S_{\boldsymbol{\sigma}_t} = (\sigma_t, \varepsilon, \varepsilon, \varepsilon)$ with σ_t a store for $1 \leq t \leq TS$.

We have the following:

Theorem C.2. *Let P be a well-formed kernel. If for all $1 \leq t_1 < t_2 \leq TS$ the program P_{t_1,t_2} is race free and terminating, then P is race free and terminating.*

Proof. Suppose P is either not terminating or not race free. Then, either (a) P is race free and there exists a reduction ρ that is infinite or terminates with *error*, or (b) there exists a reduction ρ of P with at least one data race.

In the case of (a), let t be a thread that is responsible for an infinite number steps of ρ or that is responsible for the termination with *error*. As P is assumed to be race free, we can remove from ρ all steps for which t is not responsible and that are not the responsibility of some arbitrary thread $t' \neq t$. The havocs performed at barriers ensure that the reduction ρ' so obtained is a reduction of $P_{t,t'}$, contradiction.

In the case of (b), consider the shortest (partial) reduction ρ' such that ρ' has a data race and is a prefix of ρ. Consider one of the data races in ρ' (there may be multiple, but all involve the last step of ρ') and suppose the two threads responsible for the data race are t and t'. Remove from ρ' all steps for which t and t' are not responsible. The havocking at at barriers ensures that the reduction ρ' so obtained is a reduction of $P_{t,t'}$, which also has a data race (otherwise there was a shorter reduction with a data race) and which can be extended to a maximal reduction not terminating with *infeasible* by well-formedness of P, contradiction. □

Observe that the lock-step construction also applies to our abstracted programs P_{t_1,t_2}; we simply need to replace every construct of the form $C_{t=1}^{TS}$ with $C_{t\in\{t_1,t_2\}}$, where C is either $\langle\rangle$, $\bigwedge$, or $\bigvee$. All facts established in the previous section continue to hold, mutatis mutandis. In particular, Theorem 5.2 continues to hold. Hence, we also have:

Theorem C.3. *Let P be a well-formed kernel. If for all $1 \leq t_1 < t_2 \leq TS$ the lock-step program $\phi(P_{t_1,t_2})$ is race free and terminating, then P is race free and terminating.*

Observe that the reverse of the above theorem does not hold due to the havocking introduced at each barrier.