NDSeq: Runtime Checking for Nondeterministic Sequential Specifications of Parallel Correctness

Jacob Burnim Tayfun Elmas George Necula Koushik Sen

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley

{jburnim,elmas,necula,ksen}@cs.berkeley.edu

Abstract

We propose to specify the correctness of a program's parallelism using a sequential version of the program with controlled nondeterminism. Such a *nondeterministic sequential specification* allows (1) the correctness of parallel interference to be verified independently of the program's functional correctness, and (2) the functional correctness of a program to be understood and verified on a sequential version of the program, one with controlled nondeterminism but no interleaving of parallel threads.

We identify a number of common patterns for writing nondeterministic sequential specifications. We apply these patterns to specify the parallelism correctness for a variety of parallel Java benchmarks, even in cases when the functional correctness is far too complex to feasibly specify.

We describe a sound runtime checking technique to validate that an execution of a parallel program conforms to its nondeterministic sequential specification. The technique uses a novel form of conflict-serializability checking to identify, for a given interleaved execution of a parallel program, an equivalent nondeterministic sequential execution. Our experiments show a significant reduction in the number of false positives versus traditional conflict-serializability in checking for parallelization bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Reliability, Verification

1. Introduction

The spread of multicore processors and the end of rapidly growing single-core performance is increasing the need for programmers to write parallel software. Yet writing correct parallel programs with explicit multithreading remains a difficult undertaking. A programmer must ensure not only that each part of his or her program computes the correct results in isolation, but also that the uncontrolled and nondeterministic interleaving of the program's parallel threads cannot cause harmful interference, leading to incorrect final results. This need to simultaneously reason about sequential functional correctness and the correctness of parallel interleavings poses a great challenge both for programmers writing, understanding, and testing their software and for tools analyzing and verifying such software.

We previously proposed *nondeterministic sequential (NDSeq) specifications* [9] as a means to separate the correctness of the parallelism of a program from its sequential functional correctness. The key idea is for a programmer to specify the *intended* or *algorithmic* nondeterminism in a program using annotations, and then the NDSeq specification is a version of the program that is sequential but includes the annotated nondeterministic behavior. The only valid parallel behaviors are those allowed by the NDSeq specification—any *additional* nondeterminism is an error, due to unintended interference between interleaved parallel threads, such as data races or atomicity violations. Thus, a program with such annotated nondeterminism serves as its own NDSeq specification for the correctness of its parallelism.

Showing that a parallel program conforms to its NDSeq specification is a strong statement that the program's use of parallelism is correct. The behavior of the program can be understood by considering only the NDSeq version of the program, as executing the program in parallel cannot produce any different results. Testing, debugging, and verification of functional correctness can be performed on this sequential version, with no need to deal with the uncontrolled interleaving and interference of parallel threads. We show in this work that NDSeq specifications for parallel applications can be both written and checked in a simple manner, independent of an application's complex functional correctness.

In this paper, we propose several patterns for writing NDSeq specifications, and we apply these patterns to specify the parallelism correctness of a number of Java benchmarks. We find that, with a few simple constructs for specifying intended nondeterministic behavior, adding such specifications to the program text was straightforward for a variety of applications. This is despite the fact that, for many of these applications, writing a traditional functional correctness specification would be extremely difficult. (Imagine, for example, specifying the correct output of an application to render a fractal, or to compute a likely phylogenetic tree given genetic sequence data.) For many of our benchmarks, verifying that the final output is correct even for a single known input is challenging.

We propose a novel sound runtime technique for checking that a structured parallel program conforms to its NDSeq specification. Given a parallel execution of such a program, we perform a conflict-serializability check to verify that the same behavior could have been produced by the NDSeq version of the program. But first, our technique combines a dynamic dependence analysis with a program's specified nondeterminism to show that conflicts involving certain operations in the trace can be soundly ignored when performing the conflict-serializability check. Our experimental results show that our runtime checking technique significantly reduces the number of false positives versus traditional conflict-serializability in checking parallel correctness.

```
 1:  coforeach (i in 1,...,N) {
 2:    b = lower_bound_cost(i);
 3:    t = lowest_cost;
 4:    if (b >= t)
 5:        continue;
 6:    c = expensive_compute_cost(i);
 7:    atomic {
 8:        t = lowest_cost
 9:        if (c < t) {
10:            lowest_cost = c;
11:            best_soln = i;
12: } }  }
```

(a) An example parallel search procedure.

```
 1:  nd-foreach (i in 1,...,N) {
 2:    b = lower_bound_cost(i);
 3:    t = lowest_cost;
 4:    if (* && (b >= t))
 5:        continue;
 6:    c = expensive_compute_cost(i);
 7:    atomic {
 8:        t = lowest_cost
 9:        if (c < t) {
10:            lowest_cost = c;
11:            best_soln = i;
12:  } }   }
```

(b) Nondeterministic sequential specification for parallel search procedure.

```
 1:  coforeach  (i in 1,...,N) {
 2:    b = lower_bound_cost(i);
 3:    t = lowest_cost;
 4:    if ( true* &&  (b >= t))
 5:        continue;
 6:    c = expensive_compute_cost(i);
 7:    atomic {
 8:        t = lowest_cost
 9:        if (c < t) {
10:            lowest_cost = c;
11:            best_soln = i;
12:  } }   }
```

(c) Parallel search with embedded nondeterministic sequential specification.

Figure 1. An example parallel search procedure (Figure 1(a)) and a nondeterministic sequential specification (NDSeq) for its parallel correctness (Figure 1(b)). Since a parallel program and its NDSeq specification look very similar, in practice, we do not write the NDSeq specification of a parallel program separately, but embed it in the parallel program itself. Figure 1(c) shows the parallel search procedure with its NDSeq specification embedded in the parallel code. The boxed constructs have different semantics when viewed as parallel code or as nondeterministic sequential specification.

2. Overview

In this section, we discuss an example program in detail to motivate NDSeq specifications and to informally describe our runtime checking that a parallel program is parallelized correctly with respect to its NDSeq specification. In Section 3, we then give a formal definition of parallelization correctness. In Section 4, we illustrate the generality of these specifications and of our checking approach on several examples highlighting different parallelization patterns. Section 5 describes the details of the runtime checking algorithm. We discuss the experimental results in Section 6 and conclude in Section 8 by pointing out significance and possible future applications of NDSeq specifications.

2.1 Motivating Example

Consider the simplified version of a generic branch-and-bound procedure given in Figure 1(a). This program takes as input a list of N possible solutions and computes `lowest_cost`, the minimum cost among the possible solutions, and `best_soln`, the index of a solution with minimum cost. Function `expensive_compute_cost(i)` computes the cost of solution `i`. Because this computation is expensive, the program first computes a lower bound for the cost of solution `i` with `lower_bound_cost(i)`. If this lower bound is no smaller than the lowest cost found so far (i.e. `lowest_cost`), then the program skips computing the exact cost for solution `i`.

The program is a *parallel* search—the **`coforeach`** loop allows different iterations to examine different potential solutions in parallel. Thus, updates to `lowest_cost` and `best_soln` at Lines 8–11 are enclosed in an **`atomic`** block, a synchronization mechanism that enforces that these lines be executed atomically—that is, all-at-once and without interruption by any other thread. Functions `expensive_compute_cost` and `lower_bound_cost` have no side-effects and do not read any mutable shared data (i.e., `lowest_cost` or `best_soln`), and thus require no synchronization.

2.2 Nondeterministic Sequential Specifications

We would like to formalize and specify that the search procedure in Figure 1(a) is *parallelized correctly*, and we would like some way to verify or test this parallel correctness.

If we could specify the full functional correctness of our example program—i.e. specify precisely which outputs are correct for each input—then this specification would clearly imply that the parallelization of the program was correct. But writing a full functional correctness specification is often a very difficult task. For our example search procedure, the cost of a possible solution may be a complex function whose behavior we are unable to specify, short of reimplementing the entire `expensive_compute_cost` in a specification/assertion language. Even if we could write such a specification, verifying the full functional correctness could similarly require very complex reasoning about the internals of the cost and lower bound computations.

We argue that we should seek to specify the correctness of the parallelism in our example program independently of the program's functional correctness. More generally, we aim to decompose our effort of verifying or checking the correctness of the program into two parts: (1) addressing the correctness of the parallelism, independent of the complex functional correctness and (2) addressing the functional correctness independent of any reasoning about the interleaving of parallel threads.

A natural approach to specifying parallel correctness would be to specify that the program in Figure 1(a) must produce the same results—i.e. compute the same `lowest_cost` and `best_soln`—as a version of the program with all parallelism removed. But if we simply replace the **`coforeach`** with a traditional **`foreach`**-loop that iterates `i` sequentially from 1 to `N`, we do not get an equivalent program. Rather, the parallel program has two "freedoms" the sequential program does not:

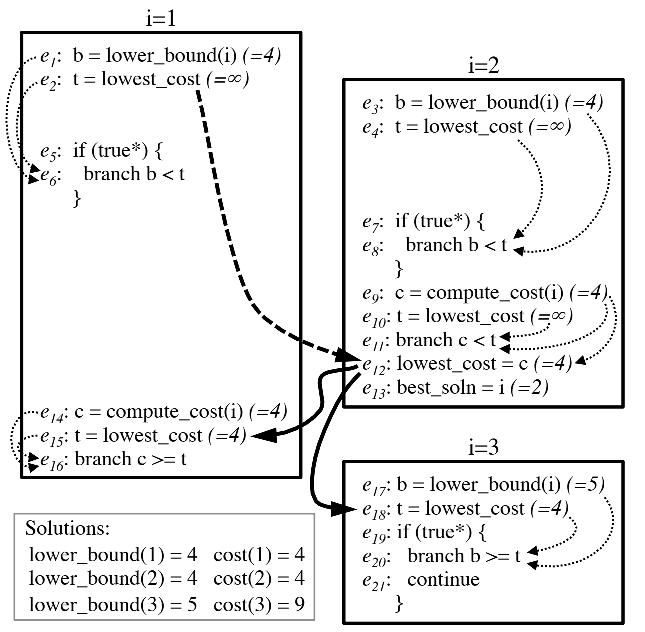
ND1 First, the parallel search procedure is free to execute the parallel loop iterations in any *nondeterministic* order. If there are multiple solutions of minimum cost, then different runs of the procedure may return different values for `best_soln`, depending on the order in which the loop iterations are scheduled.

The hypothetical sequential version, on the other hand, would be deterministic—it would always first consider solution 1, then solution 2, ..., up to solution `N`.

ND2 Second, the parallel program is free, in a sense, to not perform the optimization in Lines 2–5, in which the rest of an iteration is skipped because the lower bound on the solution cost is larger than the minimum cost found so far.

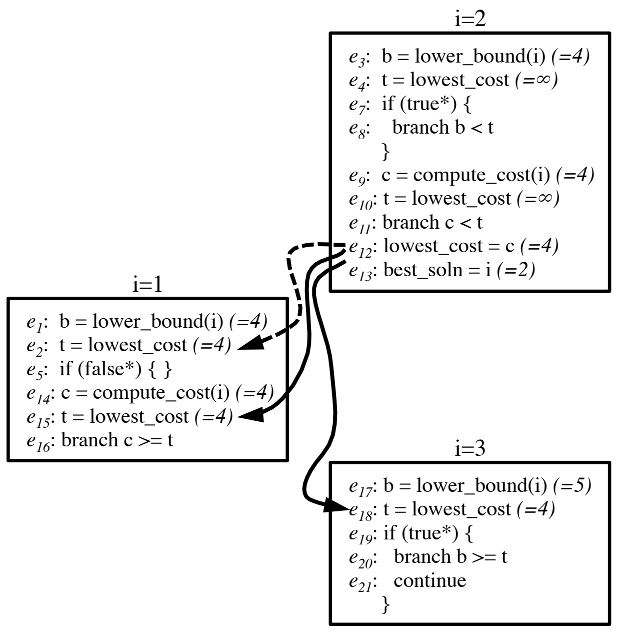
Consider two iterations with the same cost and with lower bounds equal to their costs. In the hypothetical sequential version, only one of the iterations would proceed to compute its cost. In the parallel code in Figure 1(a), however, both iterations may proceed past the check in Line 3 (as `lowest_cost` is initially ∞).

We propose to specify the correctness of the parallelism by comparing our example parallel program to a version that is sequential but contains the nondeterministic behaviors ND1 and ND2. Such



i=1
e_1: b = lower_bound(i) (=4)
e_2: t = lowest_cost (=∞)
e_5: if (true*) {
e_6: branch b < t
}
e_{14}: c = compute_cost(i) (=4)
e_{15}: t = lowest_cost (=4)
e_{16}: branch c >= t
i=2
e_3: b = lower_bound(i) (=4)
e_4: t = lowest_cost (=∞)
e_7: if (true*) {
e_8: branch b < t
}
e_9: c = compute_cost(i) (=4)
e_{10}: t = lowest_cost (=∞)
e_{11}: branch c < t
e_{12}: lowest_cost = c (=4)
e_{13}: best_soln = i (=2)
i=3
e_{17}: b = lower_bound(i) (=5)
e_{18}: t = lowest_cost (=4)
e_{19}: if (true*) {
e_{20}: branch b >= t
e_{21}: continue
}
Solutions:
lower_bound(1) = 4 cost(1) = 4
lower_bound(2) = 4 cost(2) = 4
lower_bound(3) = 5 cost(3) = 9

Figure 2. A parallel execution of three iterations (i=1,2,3) of the parallel search procedure. The vertical order of events shows the interleaving. Each assignment shows in parentheses the value being assigned. The thin dotted arrows denote data dependencies. The thick solid and thick dashed arrows denote transactional conflicts. Our analysis proves that the transactional conflict $e_2 \dashrightarrow e_{12}$ can be safely ignored for the serializability check.



i=2
e_3: b = lower_bound(i) (=4)
e_4: t = lowest_cost (=∞)
e_7: if (true*) {
e_8: branch b < t
}
e_9: c = compute_cost(i) (=4)
e_{10}: t = lowest_cost (=∞)
e_{11}: branch c < t
e_{12}: lowest_cost = c (=4)
e_{13}: best_soln = i (=2)
i=1
e_1: b = lower_bound(i) (=4)
e_2: t = lowest_cost (=4)
e_5: if (false*) { }
e_{14}: c = compute_cost(i) (=4)
e_{15}: t = lowest_cost (=4)
e_{16}: branch c >= t
i=3
e_{17}: b = lower_bound(i) (=5)
e_{18}: t = lowest_cost (=4)
e_{19}: if (true*) {
e_{20}: branch b >= t
e_{21}: continue
}

Figure 3. An execution of the nondeterministic sequential version of the search procedure. This execution is a serialization of the parallel execution in Figure 2, producing the same final result. The thick solid and thick dashed arrows denote transactional conflicts. Note that the order of conflicts $e_{12} \rightarrow e_{15}$ and $e_{12} \rightarrow e_{18}$ is the same as in Figure 2, while conflict $e_{12} \dashrightarrow e_2$, involving irrelevant event e_2, has been flipped.

a version of the program is a *nondeterministic sequential (NDSeq) specification* for the program's parallel correctness. For the program in Figure 1(a), our NDSeq specification is listed in Figure 1(b). The NDSeq specification differs from the parallel program in two ways:

1. The parallel **coforeach** loop at Line 1 is replaced with a sequential but nondeterministic **nd-foreach** loop, which can run its iterations in any order.
2. The "* &&" is added to the condition at Line 4. This expression "*" can nondeterministically evaluate to `true` or `false`, allowing the sequential specification to run the rest of a loop iteration even when `lower_bound_cost(i)` $\geq$ `lowest_cost`.

This specification is a completely sequential version of the program, executing its loop iterations one-at-a-time with no interleaving of different iterations. It contains only the controlled nondeterminism added at the two above points. We say that a parallel program *conforms* to its NDSeq specification when every final result of the parallel program can also be produced by an execution of the NDSeq specification. Section 3 elaborates the semantics of NDSeq specifications and our precise definition of parallelism correctness.

Note the close similarity between the parallel program in Figure 1(a) and its NDSeq specification in Figure 1(b). Rather than maintaining our parallel search procedure and its NDSeq specifications as separate artifacts, we embed the NDSeq specifications into the parallel code, as shown in Figure 1(c). Here we show in boxes the **coforeach** and "*true** &&" to indicate that these two constructs are interpreted differently when we consider Figure 1(c) as a parallel program or as nondeterministic sequential one. That is, in the parallel interpretation, **coforeach** is a standard parallel for-loop and "*true**" always evaluates to *true*, yielding the exact behavior of Figure 1(a). But when interpreted as nondeterministic sequential constructs, **coforeach** is treated as a **nd-foreach** and "*true**" can nondeterministically evaluate to *true* or *false*, yielding exactly the behavior of Figure 1(b). With these annotations, the example program in Figure 1(c) embeds its own NDSeq specification for the correctness of its parallelism.

2.3 Runtime Checking of Parallel Correctness

We now give an overview of our proposed algorithm for the runtime checking that a parallel program conforms to its NDSeq specification. We will present the algorithm in full formal detail in Section 5.

Figure 2 illustrates a possible parallel execution of our example program in Figure 1(c) on $\texttt{N} = 3$ possible solutions. The three iterations of the parallel for-loop are shown in separate boxes, with the $i = 1$ iteration running in parallel with the $i = 2$ iteration, followed by the $i = 3$ iteration. Although the $i = 1$ compute a lower bound and compare it against `lowest_cost`, iteration $i = 2$ is first to compute the full cost of its solution and to update `lowest_cost` = 4 and `best_soln` = 2.

We would like to verify that the parallelism in this execution is correct. That is, the final result produced is also possible in an execution of the NDSeq specification. The key will be to show that parallel loop iterations together are *serializable* [29]—i.e. there exist some order such that, if the iterations are executed sequentially in that order, then the same final result will be produced.

A common restriction of serializability that can be efficiently checked and is thus often used in practice is *conflict-serializability* [29]. Given a collection of transactions—in this case, we think of each parallel loop iteration as a transaction—we form the *conflict graph* whose vertices are the transactions and with a *conflict edge* from transaction tr to tr' if tr and tr' contain conflicting operations *op* and op' with *op* happening before *op'*. (Two operations from different threads are conflicting if they operate on the same shared global and at least one of them is a write; in our example the conflicts are shown with thick solid or thick dashed arrows.) It is a well-known result [29] that if there are no cycles in the conflict graph, then the transactions are serializable.

But conflict-serializability is too strict for our example in Figure 2. There are three pairs of conflicting operations in this execution: a read-write conflict between e_2 and e_{12}, a write-read conflict between e_{12} and e_{15}, and a write-read conflict between e_{12} and e_{18}. In particular, the $i = 1$ and $i = 2$ transactions are not conflict-serializable because the $i = 2$ transaction's write of `lowest_cost` at e_{12} comes after the read of `lowest_cost` at e_2 but before the read at e_{15}.

In this paper, we generalize conflict-serializability by determining, using a dynamic data dependence analysis, that the only use of the value read for `lowest_cost` at e_2 is in the branch condition at e_6. (The data dependence edges in each thread are shown in Figure 2 with the thin dashed arrows.) But because of our added nondeterminism at Line 4 of our example program, this branch condition is gated by the nondeterministic condition at e_5. Our data dependence analysis tells us that no operation performed inside the `if (true*)` opened at e_5 has any local or global side-effects—thus, in the equivalent sequential execution whose existence we are trying to show, we can choose this nondeterministic condition to be `false`, in which case the read of `lowest_cost` at e_2 will never be used. This shows that the read-write conflict involving e_2 and e_{12} is *irrelevant*, because, once we choose the nondeterministic condition to be false, the value read at e_2 has no effect on the execution.

With only the remaining two *relevant* conflicts, the conflict graph has no cycles, and thus we will conclude that the loop iterations are serializable. Figure 3 shows a serial execution whose existence we have inferred by verifying that there are no conflict cycles. Note that the two *relevant* conflicts, $e_{12} \rightarrow e_{15}$ and $e_{12} \rightarrow e_{18}$, are preserved—they both appear in the serial execution in the same order. But the irrelevant conflict has been flipped. The write of `lowest_cost` at e_{12} now happens before the read at e_2, but this change does not affect the final result of the execution, because the value read for `lowest_cost` does not affect the control-flow or any writes to global variables.

Now suppose that the execution in Figure 2 produces an incorrect result, i.e., violates a functional specification of the parallel program. Because we showed above that this execution is equivalent (with respect to the final relevant state) to the computed serial execution in Figure 3, then the nondeterministic sequential execution exhibits the same functional bugs as the parallel execution. Thus, we can simply debug the serial execution without worrying about thread interleavings.

3. Parallelism Correctness with Nondeterministic Sequential Specifications

In this section, we formally define our programming model, our nondeterministic sequential (NDSeq) specifications, and our notion of parallel correctness. As discussed in Section 2.2, we embed the NDSeq specifications for a parallel program in the program itself. We achieve this both by overloading parallel language constructs and by adding a couple of new constructs. The syntax for the language is shown in Figure 4.

To simplify the presentation we consider a program $\mathcal{P}$ to consist of a single procedure. We omit discussion of multiple procedures and object-oriented concepts, and we assume that each global variable refers to a distinct location on the shared heap and that each local variable refers to a distinct location on the stack of a thread. Handling these details in our dynamic analysis is straightforward.

For each program $\mathcal{P}$, we define two sets of executions *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$), described below. The correctness of a parallel program is then given by relating *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$).

$$\begin{array}{c}
g \in \mathit{Global} \quad l \in \mathit{Local} \quad \mathit{Var} = \mathit{Global} \cup \mathit{Local}
\end{array}$$

$$\begin{array}{rcl}
x \in \mathit{Var} & ::= & l \mid g \\
b & ::= & l \mid \boxed{\mathit{true}*} \mid \boxed{\mathit{false}*} \\
s \in \mathit{Stmt} & ::= & l = l\ \mathit{op}\ l \mid l = \text{constant} \mid l = l \mid g = l \mid l = g \\
 & \mid & s; s \mid \mathbf{if}(b)\ s \mid \mathbf{if}(l)\ s\ \mathbf{else}\ s \\
 & \mid & \mathbf{while}(l)\ s \mid \mathbf{for}\ (l\ \mathbf{in}\ l)\ s \\
 & \mid & \mathbf{continue} \mid \mathbf{break} \mid \mathbf{return} \mid \cdots \\
 & \mid & \mathbf{atomic}\ s \\
 & \mid & \boxed{\mathbf{coforeach}}\ (l\ \mathbf{in}\ l)\ s \mid \boxed{\mathbf{cobegin}}\ s; ...; s
\end{array}$$

Figure 4. Selected statements of our language. The constructs with a different semantics in the parallel program and the sequential specification are shown in boxes.

Parallel executions: *ParExecs*($\mathcal{P}$) contains the parallel executions of $\mathcal{P}$ where each **cobegin** and **coforeach** statement creates implicitly new threads to execute its body. **cobegin** $s_1; ...; s_n$ is evaluated by executing each of $s_1, ..., s_n$ on a separate, newly created thread. **coforeach** is evaluated by executing each iteration of the loop on a separate, newly created thread. Following structured fork/join parallelism, a parallel execution of a **cobegin** and **coforeach** statement terminates only after all the threads created on behalf of the statement terminate. Assignments, the evaluation of conditionals, and entire **atomic** statements, are executed as atomic steps without interruption by other threads. In the parallel semantics, *true*$*$ and *false*$*$ always evaluate to *true* and *false*, respectively.

Sequential executions: *NdSeqExecs*($\mathcal{P}$) contains the (nondeterministic) sequential executions of $\mathcal{P}$ where all statements are evaluated sequentially by a single thread. Under the sequential semantics, the statements other than **if** with $*$, **cobegin**, and **coforeach** are interpreted in the standard way. Each evaluation of **cobegin** $s_1; ...; s_n$ is equivalent to running a nondeterministic permutation of statements $s_1, ..., s_n$, where each $s_{i \in [1..n]}$ executes sequentially. A statement **coforeach** is evaluated similarly to its deterministic version (**for**) except that the elements of the collection being iterated over are processed in a nondeterministic order. This, in essence, abstracts the semantics of the collection to an unordered set. Keyword **atomic** has no effect in the sequential case, so **atomic** s is simply equivalent to s. Finally, *true*$*$ or *false*$*$ yield nondeterministic boolean values each time they are evaluated.

Parallelism Correctness. We describe executions of $\mathcal{P}$ using a standard notion of small-step operational semantics extended with nondeterministic evaluation of **cobegin** and **coforeach**, and nondeterministic branches (*true*$*$ and *false*$*$).

The parallelism correctness for $\mathcal{P}$ means that every final state reachable by a parallel execution of the program from a given initial state is also reachable by an NDSeq execution from the same initial state. Therefore, parallel executions have no unintended nondeterminism caused by thread interleavings: either the nondeterminism is prevented using synchronization, or it is expressed by the nondeterministic control flow in the sequential specification.

While defining parallel correctness, we distinguish a set of global variables as *focus* variables, which contain the final results of a program. Then, we reason about the equivalence of executions on the final values of the focus variables.

Definition 1 (Parallelism correctness) *A program* $\mathcal{P}$ *conforms to its NDSeq specification with respect to a set Focus* $\subseteq$ *Global iff for every parallel execution* $E \in$ *ParExecs*($\mathcal{P}$)*, there exists a nondeterministic sequential execution* $E' \in$ *NdSeqExecs*($\mathcal{P}$)*, such that the initial states of* E *and* E' *are the same and the final states agree on the values of all variables in Focus.*

4. Nondeterministic Specification Patterns

The use of nondeterministic sequential specifications is an attractive way to specify parallel correctness, yet it is not immediately clear where to introduce the nondeterministic constructs into the specification (1) without breaking the functional correctness while (2) capturing the nondeterminism due to thread interleavings.

Figure 5 shows the pseudo-code for three common patterns that we encountered repeatedly in our experiments. Each of these patterns considers parallel worker tasks where there is a potential for conflicting accesses to shared data. In contrast, applications where shared data is distributed strictly disjointly between tasks do not require use of **if**(*true*∗) specifications. Next, we discuss these patterns in detail.

4.1 Optimistic Concurrent Computation

This pattern is a manually implemented analogue of software transactional memory (STM) [37]. A parallel task performs its work optimistically in order to reduce the synchronization with other threads. It reads the shared data (shared) required for its work to a local variable (local) and performs the computation (do_work) without further access to shared data. Before committing the result of the computation back to the shared space, it checks if the input it read previously has been modified by another thread (is_conflict). In that case it retries the work. This pattern is used when the time spent for the local computation dominates the time for checking conflict and committing, and the contention on the same regions of shared memory is low.

Such fail-retry behaviors are not normally conflict-serializable when another thread updates shared during do_work. The update conflicts with the read of shared before and after do_work. However, in those situations we expect is_conflict to return true and the commit to be skipped. The $\boxed{true* \,\&\&}$ allows the NDSeq program to nondeterministically skip the conflict checking and the commit. This captures that (1) it is acceptable from a partial-correctness point of view to skip the commit nondeterministically even without checking is_conflict, and (2) if the commit is skipped then the read of shared data before and after do_work are irrelevant and can be ignored for conflict serializability purposes.

Examples: This pattern is used in non-blocking data structures, e.g., stacks and queues, to implement optimistic concurrent access to the data structure without locks. These data structures implement the atomic block in the pseudo code using a compare-and-swap (CAS) operation [18]. We have also encountered this pattern when parallelizing a mesh refinement program from the Lonestar benchmark suite (see Section 6). Our parallelization instantiates the pattern in Figure 5 as follows:

```
// Processing cavity of a node N:
while (true) {
  local_cavity = read cavity of N from shared mesh;
  refined_cavity = refine the cavity locally;
  atomic {
    if ( true* && mesh still contains all nodes in local_cavity) {
      replace old cavity in mesh with refined_cavity;
      break;
} } }
```

Appendix A gives more examples of NDSeq specifications for non-blocking implementations following the same pattern.

4.2 Redundant Computation Optimization

In contrast with optimistic computation where each parallel task must complete its work, in the redundant computation pattern, each thread may choose to skip its work when it detects that the work is no longer necessary (is_work_redundant). Here synchronizing the check for redundancy and the actual work may not be practical when the latter is a long running operation.

```
Optimistic Concurrent Computation Pattern
while (true) {
  local = shared;
  local' = do_work(local);
  atomic {
    if ( true* && !is_conflict(local,shared)) {
      shared = local';  // commit result
      break;
} } }

Redundant Computation Optimization Pattern
if ( true* && is_work_redundant(shared)) {
  // work is unnecessary; skip the work
} else {
  do_work(); // accesses shared
}

Irrelevant Computation Pattern
do_work(local, shared);
if ( true* ) {
  do_irrelevant_work();
}
```

Figure 5. Common concurrency patterns and use of *true*∗ and *false*∗ to express the nondeterminism.

Threads operating under this pattern are not conflict serializable if another thread updates the shared state while our thread calls is_work_redundant and finds that it returns false. Those updates conflict with the shared read before calling is_work_redundant and while executing do_work.

The $\boxed{true* \,\&\&}$ allows the NDSeq program to nondeterministically skip the call to is_work_redundant and do the work anyway. This expresses that (1) it is acceptable from a partial-correctness point of view to skip the redundancy check and to do the actual work, and also that (2) if we skip the redundancy check, then the initial read of shared state is not relevant to the computation and can be ignored for conflict-serializability purposes.

Examples: This pattern is often used when a solution space is examined by multiple threads to improve convergence to an optimal solution. Our running example in Section 2.1 follows this pattern: Lines 2–5 in Figure 1 test the lower bound of the current solution to decide if the computation at Line 6 can produce a better solution. The phylogeny benchmark from the Parallel Java Library follows a similar bound check to prune the search space for optimal phylogenetic trees. Programs using caches to avoid multiple computations of a function for the same input also use this pattern.

4.3 Irrelevant Computation

This pattern generalizes tasks that perform some computation (shown in the **if**(*true*∗) branch) that does not affect the rest of the execution path and does not produce a result that flows into shared state that is relevant to core functional correctness. One can ignore the irrelevant part of the program when reasoning about the program in the sequential semantics, since either (1) it does not affect the focus state, or (2) it is only necessary for the parallel executions of the program.

Examples: A prevalent instance of case (1) is when updating a statistic counter to profile the procedures of a program. For example, in the following we use **if**(*true*∗) to mark the increment of a statistic counter as an irrelevant operation. Updates to counter do not affect the final result (with respect to focus variables) of the program. However, without using the **if**(*true*∗), conflicts due to counter will not be ignored by our analysis. By surrounding the if statement with **if**(*true*∗), the programmer indicates that the if branch is not relevant for the final result, and conflicts due to the accesses to `counter` when executing the branch should not affect the parallel correctness of the program. In fact, one can easily prove statically that, given `counter` is not a focus variable, skipping the conditional at all is safe for the functionality.

```
do_work(shared, local); // access shared variables exclusively
 if ( true* ) {
   if (counter < MAX_INT) {
      counter = counter + 1;
   }
}
```

Moreover, rebalancing a tree, garbage collection and compaction, maintaining a cache, and load balancing are operations that are performed to improve the performance of a parallel program, but –when implemented correctly– do not affect the core functionality of the program and thus are considered irrelevant.

An instance of (2) is when using a locking library to ensure atomicity of the relevant computation. In contrast with the statistics counters, locks are essential for the correctness of the parallelism, though the locking-state is often irrelevant for reasoning about the core functionality of the program in a sequential run.

5. Runtime Checking of Parallel Correctness

Among the various possible techniques for checking parallelization correctness, we describe here a runtime checking algorithm. We use dynamic data-flow analysis to determine parts of the executions that are not relevant to the final valuation of the focus variables. At the same time the analysis determines appropriate assignments of boolean values to the **if(***true*∗**)** nondeterministic branches in the NDSeq execution, in order to eliminate as many serialization conflicts as possible. Therefore, data flow analysis and NDSeq specifications play key roles in improving the applicability of conflict serializability for reasoning about parallelism in real programs. In the rest of this section, we describe the details of the checking algorithm and we sketch its correctness.

In order to simplify the presentation of the runtime checking algorithm for parallelism correctness we make the following assumptions about the parallel program being checked:

A1. Branch predicates are either *true*∗ or a local variable.

A2. The body of an **if(***true*∗**)** does not contain unstructured control flow (e.g., **continue**, **break**, **return**), and there is no **else** clause.

These assumptions can be established using standard program refactoring. For example, Lines 4–5 from Figure 1(c) can be translated to:

```
bool cond = false;
if( true* ){ l = (b >= t); if(l){ cond = true; } }
if(cond) continue;
```

where `cond` and `l` are new local variables.

Our checking algorithm operates on an execution trace described as a sequence of execution *events*. Let τ denote a trace and e an event. For each event e we have the following information:

- *Type*(e) is the type of the event, defined as follows:

$$T \quad ::= \quad x = x' \ \mid \ \textit{branch}(l) \ \mid \ \textit{branch}(\textit{true}*)$$

  The "$x = x'$" event type corresponds to the assignment and binary operation statements in our language (shown in Figure 4; recall that metavariable x stands for both locals and globals). We use a simple assignment in our formal description to simplify the presentation; unary and binary operators do not pose notable difficulties. We assume that an event can read a global, or write a global, but not both. The "*branch*(l)" event marks the execution of a branch operation when the boolean condition denoted by local l evaluates to true. The case of a branch when the negation of a local is true is similar. Finally, the "*branch*(*true*∗)" marks the execution of an **if(***true*∗**)** branch, which in the parallel execution is always taken. Our algorithm does not require specific events to mark the start and end of procedures or atomic blocks.

  We write $e : T$ when e has type T.

- *Thread*(e) denotes the thread that generates the event e. Recall that new threads are created when executing **cobegin** and **coforeach** statements.

- *Guard*(e) denotes the event of type *branch*(*true*∗) that corresponds to the most recent invocation of the innermost **if(***true*∗**)** that encloses the statement generating e. For events outside any **if(***true*∗**)** this value is *nil*.

  For example, in the trace shown in Figure 2, *Guard*$(e_6) = e_5$, *Guard*$(e_8) = e_7$, *Guard*(e_{20}) = *Guard*$(e_{21}) = e_{19}$, and *Guard*(e) = *nil* for all other events e.

The checking algorithm operates in two stages, shown in Figure 6. The first stage computes a subset of the events in the trace that are relevant (Section 5.1), and the second stage determines whether the relevant part of the trace is conflict serializable (Section 5.2).

5.1 Selecting Relevant Events

A standard conflict-serializability algorithm [29] considers all events in a trace. We observed that in many concurrent programs it is common for partial work to be discarded when a conflict is later detected. In such cases, some of the computations based on previously read values of shared variables are discarded and are not relevant to the rest of the execution. If we can ignore such irrelevant reads of shared variables we can prove that more paths are conflict serializable. Similarly, we can ignore writes that do not affect a relevant control flow and do not flow into the final state of the focus variables. Our experiments show that this makes a difference for most benchmarks where traditional conflict serializability reports false alarms.

Informally, an assignment event is *relevant* if it targets a location that is eventually used in the computation of a final value of a focus variable, or in the computation of a *deterministic* branch. To track this relevance aspect we compute a dynamic data-dependence relation between events. For trace τ, we define the dependence relation $\dashrightarrow$ as follows:

D1. **(Intra-Thread Data Dependence).** For each local variable read $e_j : x = l$ or branch e_j : *branch*(l), we add a dependence $(e_i \dashrightarrow e_j)$ on the last $e_i : l = x'$ that comes before e_j in τ. This dependence represents an actual data flow through local l from e_i to e_j in the current trace. Both of these events are in the same thread (since they operate on the same local) and their order and dependence will be the same in any serialization of the trace. These dependence edges are shown as thin dashed arrows in Figure 2.

D2. **(Inter-Thread Dependence).** For each global variable read $e_j : l = g$ we add dependencies $(e_i \dashrightarrow e_j)$ on events $e_i : g = l'$ as follows. From each thread we pick the last write to g that comes before e_j in τ, and the first write to g that comes after e_j in τ. This conservative dependence is necessary because the relative order of reads and writes to the global from different threads may change in a serialization of the trace. Section 5.3 explains the importance of this detail for correctness. In the example shown in Figure 2, we have such dependence edges from e_{12} to all events that read `lowest_cost`: e_2, e_4, e_{10}, e_{18}.

Let $\dashrightarrow^*$ denote the transitive closure of $\dashrightarrow$.

```
Algorithm ComputeRelevant($\tau$, $Focus$)
    // Collecting relevant events for the serializability check
    // Add all writes to focus globals
1   $Relevant = \{e : g = l \in \tau \mid g \in Focus$
                                   $\wedge$ $e$ is last write to g in $Thread$ (e)$\}$
    // Start with all top-level deterministic branches
2   $Relevant = Relevant \cup \{e : branch(l) \in \tau \mid Guard(e) = nil\}$
3   Compute data dependency relation $\dashrightarrow$ from $\tau$
4   repeat
        // Add events that some relevant events data-depend on
5       $Relevant = Relevant \cup \{e \in \tau \mid \exists e' \in Relevant.\ e \dashrightarrow^{*} e'\}$
        // Add nondeterministic branches containing relevant events
6       $Relevant = Relevant \cup$
            $\{e : branch(true*) \in \tau \mid \exists e' \in Relevant.\ Guard(e') = e\}$
        // Add deterministic branches nested inside relevant if (true*)'s
7       $Relevant = Relevant \cup$
            $\{e : branch(l) \in \tau \mid Guard(e) \in Relevant\}$
8   until $Relevant$ does not change
9   return $Relevant$

Algorithm CheckCycle($\tau$, $Focus$)
    // Check serializability of trace
10  $Relevant =$ ComputeRelevant($\tau$, $Focus$)
11  Compute conflict relation $\rightsquigarrow$ between threads
12  if exists a cycle $t \rightsquigarrow t' \rightsquigarrow^{*} t$
13      Report unserializable thread $t$
14  else
15      Declare the execution serializable
16  end if
```

Figure 6. The algorithm for checking parallelism correctness.

Figure 6 lists the algorithm **ComputeRelevant** to compute the set of relevant events. We seed the set of relevant events with all the events that assign to the global focus variables (Line 1) and the deterministic branches outside any nondeterministic blocks (Line 2). In Line 5 we add the assignment events on which existing relevant events have data dependence on.

The crucial factor that allows us to find a significant number of irrelevant events is that we can choose to skip the events (assignments and branches) corresponding to nondeterministic **if(***true∗***)** blocks, as long as those blocks are irrelevant in the trace. We extend the relevance notion from assignment events to branches as follows. A *branch*$(true*)$ event is relevant if and only if it corresponds to the execution of an **if(***true∗***)** block with at least one relevant assignment event (Line 6). For this step we use the previously introduced *Guard* function to relate events inside **if(***true∗***)** blocks with the corresponding *branch*$(true*)$ event. We also say that a *branch*(l) event is relevant if it represents control flow that must be preserved (it is not nested inside an irrelevant **if(***true∗***)** block). This is enforced in Lines 2 and 7. The computation in Lines 5–7 must be repeated to a fixed point since the additional relevant deterministic branches added in Line 7 can lead to new relevant assignments due to data dependencies.

For example, in the trace in Figure 2, relevant events are e_{9-13} from thread with $i = 2$, e_{14-16} from thread with $i = 1$, and e_{17-21} from thread with $i = 3$. Since the nondeterministic branch events e_5 and e_7 are irrelevant (no events in the rest of the trace data-depend on their bodies), the branch events e_6 and e_8 are not marked as relevant. Thus, events e_{1-2} from thread with $i = 1$ and e_{3-4} from thread with $i = 2$ have no data-dependents in $Relevant$ and they remain as irrelevant.

5.2 Checking Serializability of Transactions

The final stage in our runtime checking algorithm is a conflict serializability check implemented as a cycle detection problem similar to [12, 16]. The added element here is that we ignore the conflicts induced by irrelevant events, and we have the flexibility to alter the nondeterministic control flow in order to remove conflicts.

First we define the conflict relation between individual events.

Definition 2 (Conflicting events) *Two events $e, e' \in \tau$ are conflicting (written $e \rightsquigarrow e'$) iff (a) e occurs before e' in τ, and (b) both events operate on the same shared global variable, and at least one of them represents a write, and (c) both events are $Relevant$ in trace τ.*

Next we lift the conflict relation from events to threads. When comparing two threads for conflicts we need to consider their events and all the events of their descendant threads. Thus, for a thread t we define its *transaction* as the set of events *Trans* (t) that includes all the events of t and of the descendant threads of t.

Definition 3 (Conflicting threads) *Two threads t, t' are conflicting in trace τ (written $t \rightsquigarrow t'$) iff (a) their transaction sets are disjoint (i.e., one is not a descendant of the other), and (b) there exist two events $e \in$ Trans (t) and $e' \in$ Trans (t') that are conflicting ($e \rightsquigarrow e'$). The relation $t \rightsquigarrow^{*} t'$ is the transitive and reflexive closure of the thread conflict relation.*

The main runtime checking algorithm for parallelism correctness is shown in Lines 10–16 in Figure 6.

For the example trace shown in Figure 2 the event e_2 is not relevant, which allows the algorithm **CheckCycle** to ignore the conflict between e_2 and e_{12} (shown with thick dashed arrow in Figure 2). Without the dependence analysis we could not show that the trace is serializable.

5.3 Algorithm Correctness

The correctness of our runtime checking algorithm can be argued by showing that when the **CheckCycle** algorithm succeeds, the input trace $\tau \in$ *ParExecs*$(\mathcal{P})$ can be transformed incrementally into a trace $\tau' \in$ *NdSeqExecs*$(\mathcal{P})$ such that the final states in both traces agree on the values of the focus variables. Each incremental transformation preserves the validity of the trace and the final condition on focus variables. Some of the intermediate traces in this process will belong to the larger space of nondeterministic parallel executions *NdParExecs*$(\mathcal{P})$, which allow both interleavings (as in *ParExecs*$(\mathcal{P})$) and nondeterministic branches (as in *NdSeqExecs*$(\mathcal{P})$). For these executions the nondeterministic branches *true∗* and *false∗* are resolved at runtime nondeterministically to true or false.

The first trace transformation that we perform is to eliminate the events corresponding to **if(***true∗***)** blocks that were found irrelevant by **ComputeRelevant**. The second transformation is to commute adjacent events from different threads that are not in conflict, either because they do not operate on a shared global, or because one of them is irrelevant. The correctness of these steps, i.e., they preserve the validity of the trace and the final values of focus variables, is established by Lemma 1 and Lemma 2.

The rest of the correctness algorithm builds on a standard result from database theory: a trace is conflict serializable if it can be transformed into an equivalent serial trace by commuting adjacent, non-conflicting operations of different threads. This is possible if and only if the transactional conflict graph is acyclic [29].

Lemma 1 (Skip irrelevant nondeterministic blocks)
If $\tau \in$ ParExecs$(\mathcal{P})$, let τ' be the subtrace of τ obtained by eliminating all events e such that Guard$(e) \notin$ **ComputeRelevant**$(\tau, Focus)$. *Then τ' is a valid trace in NdParExecs$(\mathcal{P})$, meaning that τ' reflects the correct control flow of the program $\mathcal{P}$ with the corresponding irrelevant true∗ resolved to false, and τ' agrees with τ on the final values of Focus variables. Furthermore,* **ComputeRelevant**$(\tau', Focus)$ *returns the same answer as for trace τ.*

The proof of this lemma relies first on the assumption (A2) stated earlier that the body of any **if**(*true*$*$) has only normal exits and no **else** clause. This means that by removing all the events in any such body results in a trace where control flows properly to the statement after the skipped **if**(*true*$*$). All assignment events eliminated in this step are irrelevant since their guard is irrelevant (Line 6 in Figure 6). Therefore subsequent control flow and the final value of focus variables are preserved. The set of relevant events does not change through this transformation because its computation does not depend on irrelevant events.

Lemma 2 (Commutativity of irrelevant events) *Consider a trace $\tau \in NdParExecs(\mathcal{P})$ and two adjacent events e_1 and e_2 in τ, such that the events are from different threads, they operate on a shared global g, at least one is a write event, and at least one is irrelevant (not in* **ComputeRelevant**$(\tau, Focus)$*). Then the trace obtained by commuting e_1 and e_2 is still a valid trace in $NdParExecs(\mathcal{P})$ and it agrees with τ on the final value of Focus variables.*

Proof Sketch: Considering the types of events we can have in the trace and the conditions of the Lemma, we have three possible cases:

- Read-after-write: $e_1 : g = l$ and $e_2 : l' = g$. If e_2 were relevant then e_1 would also be relevant (Line 5 in the algorithm, with data-dependence rule D2). Thus it must be that e_2 is irrelevant, hence the value of l' does not affect the subsequent control flow or final values of *Focus* variables. Therefore we can commute the events and the trace remains in $NdParExecs(\mathcal{P})$. The relevant events computation does not change, since $l \neq l'$ (different threads), and the relative order of *relevant* reads and writes to global does not change.
- Write-after-read: $e_1 : l = g$ and $e_2 : g = l'$. If e_1 were relevant then e_2 would also be relevant (Line 5 in the algorithm, with data-dependence rule D2; this is a crucial part of the correctness argument that depends on the conservative form of the data-dependence rule D2). Thus, e_1 is irrelevant, and the rest of this case follows the same arguments as in the read-after-write case.
- Write-after-write: $e_1 : g = l$ and $e_2 : g = l'$. It must be that there is no nearby relevant read of the global g in the trace, or else both events would be relevant (again due to data-dependence rule D2). This means that it does not matter what we write to g. The relevant set does not change after the swap because we do not change the dependence relation $\dashrightarrow$. It is for this reason that we require the dependence rule D2 to consider the nearest write to a global from each thread.

□

With these results it is straightforward to prove our main correctness result given below using standard conflict-serializability results using our relaxed notion of conflicts, as proved adequate in Lemma 2.

Theorem 1 (Correctness) *Let τ be the trace generated by a parallel execution of $E \in ParExecs(\mathcal{P})$ of a program $\mathcal{P}$. If* **CheckCycle**$(\tau, Focus)$ *does not report any unserializable transaction, then there exists a nondeterministic sequential execution $E' \in NdSeqExecs(\mathcal{P})$, such that the initial states of E and E' are the same and the final states agree on the value of all variables in Focus.*

The theorem implies that if we explore, using a model checker, all parallel executions of the program and show that all these executions are serializable, then we can conclude that the parallel program conforms to its NDSeq specification.

6. Experimental Evaluation

In this section, we describe our efforts to experimentally evaluate our approach to specifying and checking parallel correctness using NDSeq specifications. We aim to evaluate two claims:

(1) That it is feasible to write NDSeq specifications for the parallel correctness of real Java benchmarks,

(2) Our runtime checking algorithm produces significantly fewer false positives than a traditional conflict-serializability analysis in checking parallel correctness of these benchmarks.

To evaluate these claims, we wrote NDSeq specifications for the parallel correctness of a number of Java benchmarks and then used our runtime checking technique on these specifications.

6.1 Benchmarks

We evaluate our technique on a number of benchmarks that have been used in previous research [8, 12, 16] on parallel correctness tools. Note that we focus on *parallel* applications, which use multithreading for performance but fundamentally are performing a single computation that can be understood sequentially. We do not consider *concurrent* benchmarks, such as reactive systems and stream-based systems, because it is not clear whether or not such programs can be understood sequentially.

The names, sizes, and brief descriptions of the benchmarks we used are listed in Table 1. Several benchmarks are from the Java Grande Forum (JGF) benchmark suite [38], the Parallel Java (PJ) Library [20]. We report on all benchmarks that we looked at except `tsp` [46], for which we have not yet found an easy way to write the NDSeq specification (see Section 6.5). We also applied our tool to two large benchmarks in the DaCapo benchmark suite [4]. Benchmark `meshrefine` is a sequential application from the Lonestar benchmark suite [22] that we have parallelized (by converting the application's main loop into a parallel loop). Benchmarks `stack` [39] and `queue` are non-blocking concurrent data structures. For each data structure, we construct a test harness that performs several insertions and removals in parallel (i.e., in a **`cobegin`**). The `queue` is similar to the Michael and Scott queue [26], but eagerly updates the queue's tail with a 4-word compare-and-swap. This change simplified significantly the NDSeq specification.

6.2 Implementation

Although these benchmarks are written in a structured parallel style, Java does not provide structured parallelism constructs such as **`coforeach`** or **`cobegin`**. Thus, we must annotate in these benchmarks the regions of code corresponding to the bodies of parallel loops and **`cobegin`**'s. Typically, these regions are the bodies of `run` methods of subclasses of `java.lang.Thread`. Similarly, some of these benchmarks use *barrier synchronization*. As barriers have no sequential equivalent, we treat these programs as if they used a series of parallel **`coforeach`** constructs, ending one parallel loop and beginning another at each barrier. (This is a standard transformation [45] for such code.) We similarly treat each PJ benchmark, which employ sequential loops inside each of a fixed number of worker threads, as instead consisting of structured parallel loops.

In order to write NDSeq specifications, we implemented a simple library for annotating in Java programs the beginning and end of the bodies of **`if`**$(*)$, **`coforeach`**, and **`cobegin`** constructs, as well as which locations (fields, array elements, etc.) are *focus variables*. Columns 4, 5, and 6 of Table 1 list, for each benchmark, the number of such annotated parallel constructs, annotated **`if`**$(*)$, and statements added to mark focus variables.

We implemented our checking technique in a prototype tool for Java, which uses bytecode instrumentation via Soot [42]. In addition to the details described in Section 5, for Java it is necessary

Benchmark		Benchmark Description	Approximate Lines of Code (App + Library)	# of Parallel Constructs	Size of Spec		Size of Trace		Distinct Serializability Warnings	
					# of `if(*)`	# of focus stmts	All Events	Irrelevant Events	Conflict-Serializability	Our Technique
JGF	`sor`	successive over-relaxation	300	1	0	1	1,600k	112	0	0
	`matmult`	sparse matrix-vector multiplication	700	1	0	1	962k	8k	0	0
	`series`	coefficients of Fourier series	800	1	0	5	11k	140	0	0
	`crypt`	encryption and decryption	1100	2	0	3	504k	236	0	0
	`moldyn`	molecular dynamics simulation	1300	4	0	1	4,131k	79k	0	0
	`lufact`	LU factorization	1500	1	0	1	1,778k	6k	0	0
	`raytracer`	ray tracing	1900	1	0	1	6,170k	44k	1	1 **(bug)**
	`montecarlo`	Monte Carlo derivative pricing	3600	1	0	1	1,897k	534k	2	0
PJ	`pi3`	Monte Carlo approximation of π	150 + 15k	1	0	1	1,062k	141	0	0
	`keysearch3`	cryptographic key cracking	200 + 15k	2	0	4	2,059k	91k	0	0
	`mandelbrot`	fractal (Mandelbrot set) rendering	250 + 15k	1	0	6	1,707k	954	0	0
	`phylogeny`	branch-and-bound search	4400 + 15k	2	3	8	470k	5k	6	6 **(bug)**
DaCapo	`sunflow`	image rendering using ray tracing	24k	4	4	3	24,250k	2,264k	28	3 **(no bugs)**
	`xalan`	XML to HTML transformation	302k	1	3	4	16,540k	887k	6	2 **(no bugs)**
`stack`		Treiber non-blocking stack [39]	40	1	2	8	1,744	536	5	0
`queue`		non-blocking queue [26]	60	1	2	8	846	229	9	0
`meshrefine`		Delaunay mesh refinement	1000	1	2	50	747k	302k	30	0

Table 1. Experimental results. Note that the six warnings reported for `phylogeny` are all true violations caused by a single bug.

to handle language features such as objects, exceptions, casts, etc. Any Java bytecode instruction that can throw an exception—e.g., a field dereference, an array look-up, a cast, or a division—must be treated as an implicit branch instruction. That is, changing the values flowing into such an instruction can change the control-flow by causing or preventing an exception from being thrown.

Limitations. While our implementation supports many intricacies of the Java language, it has a couple of limitations:

First, our implementation tracks neither the shared reads and writes nor the flow of data dependence through uninstrumented native code. Thus, we may report an execution as having correct parallelism despite unserializable conflicts in calls to native code.

Second, our tool does not instrument all of the Java standard libraries. This may cause our tool to miss data dependencies carried through the data structures in these libraries, as well as shared reads and writes inside such data structures. To address this limitation, for certain shared data structure objects we introduced fake shared variables and inserted reads or writes of those variables whenever their corresponding objects were accessed. This allows us to conservatively approximate the conflicts and data dependencies for certain critical standard Java data structures.

6.3 Results: Feasibility of Writing Specifications

Writing an NDSeq specification for each benchmark program consisted of two steps: (1) adding code to mark which parts of the program's memory were *in focus*—i.e. storage locations whose values are relevant in the final program state, and (2) adding **if**($*$) constructs to specify intended or expected nondeterminism.

For all of our benchmarks besides `tsp`, it was possible to write an NDSeq specification for the benchmark's parallel correctness. The "Size of Spec" columns of Table 1 lists the number of **if**($*$) constructs added to each benchmark and the number of statements added to mark storage locations as in focus. These numbers show that, overall, the size of the specification written for each benchmark was reasonably small. We further found adding nondeterminism via **if**($*$) to be fairly straightforward, as all necessary nondeterminism fell under one of the NDSeq specification patterns discussed in Section 4. Identifying which storage locations were relevant to the final program result was similarly straightforward. As an example, we show in Appendix A.2 the complete NDSeq specification for the `stack` benchmark.

Though further work is needed to evaluate the general applicability of NDSeq specifications for parallel correctness, we believe it is promising preliminary evidence that we were able to easily write such specifications for a range of parallel applications.

6.4 Results: Runtime Checking

For each benchmark, we generated five parallel executions on a single test input using a simple form of race-directed parallel fuzzing [36]. On each such execution, we checked our NDSeq specification both using our technique and using a traditional, strict conflict-serializability analysis [16].

We report in Column "Size of Trace; All Events" of Table 1 the size of a representative execution trace of each benchmark. The size is the number of reads and writes of shared variables and the number of branches executed during the run. Note that most of our benchmarks generate a few hundred thousand or a few million events during a typical execution on a small test input. For a dynamic analysis, this is a more relevant measure of benchmark size than static lines of code. Column "Size of Trace; Irrelevant Events" reports the number of these events found to be *irrelevant* by our algorithm **ComputeRelevant** in Figure 6. The fraction of events found to be irrelevant, and therefore not considered during our algorithm's serializability checking, range from 0-40%.

The "Distinct Serializability Warnings" columns of Table 1 report the number of serializability warnings produced by a traditional conflict-serializability check and by our technique. Note that, in a trace of a benchmark, a conflict involving a few particular lines of code may generate many cycles among the dynamic events of the trace. We report only the number of *distinct* cycles corresponding to different sets of lines of code.

Both techniques find the two real parallelism bugs—a data race in `raytracer` due to the use of the wrong lock to protect a shared checksum, and an atomicity violation in the `phylogeny` branch-and-bound search involving the global list of min-cost solutions found. (The six warnings for `phylogeny` are all real violations caused by this single bug.) Because of these bugs, neither `raytracer` nor `phylogeny` is equivalent to its NDSeq spec.

A traditional conflict-serializability analysis also gives *false warnings* for six benchmarks—incidents where there are cycles of conflicting reads and writes, but the parallel code is still equivalent to its NDSeq specification. In four of these cases, our algorithm leverages its dynamic dependence analysis and specified nondeterminism and focus variables to verify that these conflicts involve reads and writes *irrelevant* to the final program result. In this way, our algorithm eliminates all false warnings for benchmarks `montecarlo`, `stack`, `queue`, and `meshrefine`.

For benchmarks `sunflow` and `xalan`, our checking algorithm eliminates 25 and 4 false warnings, respectively, produced by strict conflict-serializability checking, but generates 3 of the same false warnings for `sunflow` and 2 for `xalan`. We discuss in greater detail below these false warnings that our analysis was and was not able to eliminate.

Note that previous work on atomicity checking, such as [16], typically evaluate on such benchmarks by checking whether or not each individual method is atomic. Thus, a single cycle of conflicting reads and writes may lead to multiple warnings, as every method containing the cycle is reported to be non-atomic. (Conversely, multiple cycles may be reported as a single warning if they all occur inside a single method.) Our numbers of reported violations are not directly comparable, as we are interested only in whether each execution of an entire parallel construct is serializable and thus equivalent to an execution of its sequential counterpart.

`montecarlo` Benchmark. Each parallel loop iteration of the `montecarlo` benchmark contains several conflicting reads and writes on shared static fields. (The reads and writes occur inside the constructor of a temporary object created in each iteration.) To a naïve, traditional conflict-serializability analysis, these reads and writes make it appear that no equivalent serial execution exists. However, it turns out that the values written to and read from these static fields are never used—they affect neither the control flow nor the final program result. Thus, our analysis determines that these events are irrelevant and need not be considered during serializability checking. The remaining, relevant events *are* serializable, and thus our technique reports that the observed executions of `montecarlo` conform to its nondeterministic sequential specification.

`stack`, `queue`, and `meshrefine` Benchmarks. Benchmark `meshrefine` employs the Optimistic Concurrent Computation pattern described in Section 4. Each parallel iteration reads from the shared triangular mesh that is being refined, and then optimistically computes a re-triangulation of a region of the mesh. It then atomically checks that no conflicting modifications have been made to the mesh during its computation and either: (1) commits its changes to the mesh if there are no conflicts, or (2) discards its optimistic computation and tries again. When conflicting modifications occur, an execution of `meshrefine` is clearly not strictly conflict-serializable. However, when we specify with an **`if`**$(*)$ that the sequential execution is free to nondeterministically discard its optimistic computation and retry, even when there are no conflicts, our analysis can verify that conflicts involving these shared reads and optimistic computation are not relevant when the optimistic work is discarded. And the remaining relevant events are serializable, so our analysis reports no false warnings in this case.

The `stack` and `queue` benchmarks are also instances of the Optimistic Concurrent Computation pattern, where shared reads are performed, but these reads are only relevant when a later compare-and-swap operation succeeds. Conflicts leading to failing CAS operations in these benchmarks lead to false positives for a strict conflict-serializability analysis, but our technique determines that these conflicts are not relevant to the final program result.

`sunflow` and `xalan` Benchmarks. Benchmarks `sunflow` and `xalan` cause false alarms due to the following lazy initialization pattern:

```
1: if ( true* && flag == true) {
2:   // skip initialization
3: } else {
4:    atomic {
5:     if (flag == false) {
6:       initialize shared object
7:       flag = true;
   } } }
```

In the above pattern, each thread checks if some shared object has been initialized and, if not, initializes the object itself. The `flag` variable, which is initially `false`, indicates whether the object has been initialized.

This pattern has two potential sources of false alarms:

1. One thread may read that `flag` is `false` at Line 1, and then another thread may initialize the object and set `flag` to `true`, so that the first thread then reads that `flag` is `true` at Line 5. This is a violation of conflict-serializability.

   This is an instance of the Redundant Computation Optimization Pattern described in Section 4, as a thread can always choose to skip the check at Line 1, since `flag` will be checked again at Line 5. By annotating the first check with *true*$*$ &&, our analysis ignores irrelevant conflicts involving threads reading that `flag` is `false` at Line 1 and eliminates this kind of false warning.

2. When one thread initializes the shared object and sets `flag` to `true`, and then other threads read both `flag` and the shared object, our analysis sees conflict edges from the initializing thread to the other threads. These can lead to conflict cycles if the initializing thread later performs any relevant reads of data written by other threads. Our technique will report these cycles as violations.

   But such reports are false warnings, because it does not matter which thread performs this kind of lazy initialization, and it is possible to serialize such executions despite outgoing conflicts from the initialization code. Future work is needed to handle this pattern in our NDSeq specifications and dynamic checking.

6.5 Caveats

While we could easily write the specifications for our benchmarks, NDSeq specifications must be used with care. First, one must be careful to not introduce so much nondeterminism that the resulting NDSeq program exhibits undesired behavior. A catalog of specification patterns, along the lines of those presented in Section 4 can guide programmers to use this technique without breaking functional correctness. Second, we note that when introducing **`if(`***true*$*$**`)`** one can easily introduce nontermination (as shown in several examples in this paper). This is safe as long as we consider only the partial correctness properties of the NDSeq specification.

Our runtime algorithm reduces the number of false positives compared to a standard notion of conflict serializability. However, it can still give false positives for the following reasons:

- We do not apply semantic-level analyses, such as commutativity analysis [33]. However, there are parallel patterns that rely on commutativity, in which it is necessary to ignore low-level conflicts that are part of larger, commutative operations (such as reductions). Two commutative updates on a focus variable (e.g., addition) can have conflicts yet still be serializable when the updates are considered at a semantic level. We are exploring ways to address this limitation.

- We only search for NDSeq executions for which the control-flow path outside **`if(`***true*$*$**`)`** branches is the same as in the parallel execution. Thus, we might miss an equivalent NDSeq path, and falsely report lack of conflict-serializability. One can eliminate this source of imprecision by exploring executions of the program with different control flows.

`tsp` Benchmark. For reasons stemming from the limitations given above, we have not found an easy way to write and check the NDSeq specification for the `tsp` [46] benchmark. Figure 7(a) gives the original form of the main search routine. Each thread is implemented as a loop (Lines 2–10): At each iteration it obtains a work `w` from the shared queue `Q` (Line 4), searches the region represented by `w` (Line 6) and updates the shared variable `MinTourLen`

```
 1: cobegin <1,...,N> {
 2:  while (!isEmpty(Q)) {
 3:    atomic {
 4:      w = get_work(Q);
 5:    }
 6:    s = recursive_solve(w);
 7:    atomic {
 8:      if (s < MinTourLen)
 9:         MinTourLen = s;
10:} } }
```

(a) Original search routine

```
1: cowhile (!isEmpty(Q)) {
2:   atomic {
3:     w = get_work(Q);
4:   }
5:   s = recursive_solve(w);
6:   atomic {
7:     if (s < MinTourLen)
8:        MinTourLen = s;
9:} } }
```

(b) Rewritten form of the search

Figure 7. TSP benchmark.

(Lines 8–9). In order to show that each thread as implemented in Figure 7(a) is serializable, one needs to prove that executions of `get_work` are commutative. This requires a nontrivial reasoning because procedure `get_work` may split the work items in `Q` and submit new work items to `Q`, which creates a dependency from a thread processing a work item created by a call to `get_work` by another thread. We found that Figure 7(a) has the equivalent functionality to the rewritten form of the search in Figure 7(b), where each thread performs only one iteration of the `while` loop. In this case, one can show that each iteration at Lines 2–9 in Figure 7(b) is serializable, as procedure `recursive_solve` is a thread-local operation and the (atomic) update of `MinTourLen` at Lines 7–8 is commutative.

7. Related Work

Several generic parallel correctness criteria have been studied for shared memory parallel programs that separates the concerns about functionality and parallelism at different granularities of execution. These criteria include data-race freedom [27, 46], atomicity [14], linearizability [19]. All these criteria provides the separation between parallel and functional correctness partially, as the restriction on thread interleavings is limited, for example, to atomic block boundaries. NDSeq develops this idea up to a complete separation between parallelism and functionality so that the programmer can reason about the intended functionality by examining a sequential or nearly sequential program. NDSeq specification differs from determinism specification and checking [5, 8, 34] in that NDSeq not only allows to specify that some part of the final state is independent of the thread schedule, but also allows to specify that the part of the final state that depends on thread schedule is equivalent to the state arising due to nondeterministic choices in the NDSeq.

We formulate the checking of parallelism correctness to a general notion of atomicity. Various static [10, 14, 40, 41] and dynamic [7, 12, 16, 25, 44, 48, 49] techniques for checking atomicity and linearizability has been investigated in the literature. The main challenge in these techniques is to reason about conflicting accesses that are simultaneously enabled but ineffective on the rest of the execution. In the Purity work [15] Flanagan et al. provide a static analysis to rule out spurious warnings due to such conflicts by abstracting these operations to no-op's. Elmas et al. generalize this idea in a static proof system called QED [11]. They progressively transform a parallel program to an equivalent sequential program with respect to functional specifications expressed using assertions. They abstract reads and writes of shared variables; however, they need to consider functional specification when applying the abstractions to guarantee that the abstraction does not introduce functional bug in the new program. In addition, both Purity and QED are based on Lipton's reduction theory [24], whereas we apply the idea to relax the checking of conflict serializability [2] for nondeterministic specifications.

Atomic-set serializability [43] is an weaker notion of atomicity, which groups storage locations into *atomic sets* and requires, for each atomic set, all atomic blocks are conflict-serializable with respect to the locations in the set. Dynamic techniques for detecting violations of atomic-set serializability has been proposed [17, 23].

Recently, several platforms and languages have been developed to guarantee that parallel programs give deterministic results, i.e. there is no bug due to parallelization. Kendo [28] enforces a deterministic interleaving of parallel tasks by controlling synchronization operations (particularly locks) in the meantime targeting to achieve a load-balancing of parallel tasks close to the nondeterministic case. In Deterministic Parallel Java (DPJ) [5] Bocchino et al. allow programmers to write parallel programs that are deterministic by design (ensured at compile time). DPJ also allows programmers to explicitly mark parallel constructs with nondeterministic sequential semantics and compose them safely with other deterministic constructs without breaking the determinism-by-default guarantees [6]. On the other hand the Galois project [21], Praun et al. [47], and Prabhu et al. [30] aim to exploit the opportunities in parallelizing irregular, inherently sequential programs. The sequential model ensured by these systems allows nondeterministic ordering of parallel loops and pipelines. Praun et al. [47] propose the programming model IPOT that allows programmers to explicitly mark portions of the program for speculative multithreaded and transactional execution. Its `tryasync` construct resembles our `cobegin` construct. IPOT allows internal nondeterminism in that intermediate states may differ from the corresponding sequential execution, but guarantees external determinism where the final state only depends on the inputs, not the thread interleavings. Their runtime technique aims to preserve, rather than checking, sequential semantics. Prabhu et al. [30] propose speculative composition and iteration as programming constructs to parallelize parts of the program with explicit dependencies. They guarantee the obedience to sequential semantics by running a sequential version of the program that verifies the speculated values of each parallel part. Saltz et al. [35], and Rauchwerger et al. [32] present runtime checks for parallelizing executions of loops. Their runtime techniques complement static transformations by tracking at runtime data dependencies across parallel loop iterations similarly to our runtime algorithm does to identify true conflicts between threads.

8. Conclusion

We proposed the use of nondeterministic sequential specifications to separate functional correctness from parallelism correctness of parallel programs. Our proposal has several advantages.

First, NDSeq specifications are lightweight. Unlike traditional mechanisms for functional specification, e.g., invariants and pre/post-conditions, NDSeq specifications do not require one to learn a complex logic or language. The original parallel program along with a few **if**(*true∗*) serves as the specification and can be used alone to detect various parallelism-related bugs.

Second, once we verify parallelism correctness, proving the functional partial correctness of the parallel program amounts to checking the functional correctness of the NDSeq program. Threads being absent, this can be done using well-developed techniques for verifying sequential programs. Note that verification of sequential programs (even with nondeterminism) is much simpler than verification of parallel programs. For example, model checking of Boolean multithreaded programs is undecidable [31], whereas model checking of Boolean nondeterministic sequential programs is decidable [13]. The latter fact has been exploited by several well-known model checkers for nondeterministic sequential programs [1, 3]. Similarly, NDSeq specifications also simplify the reasoning about other concurrency-related properties such as determinism and linearizability.

Third, NDSeq specifications can simplify debugging of functional correctness bugs. When investigating a parallel execution that exhibits a bug, the programmer can be presented with the equivalent, hence similarly buggy, NDSeq execution. This allows the programmer to analyze the bug by examining a sequential behavior of the program, which is much easier to debug than its parallel counterpart.

We proposed a runtime checking algorithm for parallelism correctness. Our algorithm is based on a combination of simple dynamic dataflow analysis and conflict serializability checking. The NDSeq specification is the key factor that improves the precision of conflict serializability by indicating the conflicts that can be safely be ignored by the analysis. We believe that a similar verification can be done statically; such an extension remains a future work. A key aspect of our checking algorithm (unlike static proof systems [11] and type systems [14]) is that it does not need to refer to functional invariants which often complicates the verification process.

Acknowledgments

We would like to Nicholas Jalbert, Pallavi Joshi, and our anonymous reviewers for their valuable comments on this paper. This research supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906, CCF-101781, CCF-0747390, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Oracle (formerly Sun Microsystems), from a gift from Intel, and from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, and Samsung.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 268–283, 2001.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9:505–525, October 2007.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 97–116, 2009.

[6] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Principles of Programming Languages (POPL)*, pages 535–548, 2011.

[7] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *Programming Language Design and Implementation (PLDI)*, pages 330–340, 2010.

[8] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *Foundations of Software Engineering (FSE)*, 2009.

[9] J. Burnim, G. Necula, and K. Sen. Separating functional and parallel correctness using nondeterministic sequential specifications. In *Hot Topics in Parallelism (HOTPAR)*, 2010. *Position paper*.

[10] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer Aided Verification (CAV)*, 2006.

[11] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Principles of Programming Languages (POPL)*, pages 2–15, 2009.

[12] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, pages 52–65, 2008.

[13] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Workshop on Verification of Infinite State Systems (INFINITY)*, 1997.

[14] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation (PLDI)*, 2003.

[15] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 221–231, 2004.

[16] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Programming Language Design and Implementation (PLDI)*, pages 293–303, 2008.

[17] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering (ICSE)*, pages 231–240, 2008.

[18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[19] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12:463–492, July 1990.

[20] A. Kaminsky. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In *Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Programming Language Design and Implementation (PLDI)*, 2007.

[22] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *International Symposium on Performance Analysis of Systems and Software, (ISPASS)*, April 2009.

[23] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *International Conference on Software Engineering (ICSE)*, pages 235–244, 2010.

[24] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[25] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[26] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Principles of Distributed Computing (PDOC)*, 1996.

[27] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Prog. Lang. Syst.*, 1(1):74–88, 1992.

[28] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, 2009.

[29] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., 1986.

[30] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *Programming Language Design and Implementation (PLDI)*, pages 50–61, 2010.

[31] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Prog. Lang. Syst.*, 22(2):416–430, 2000.

```
class Stack :
  class Node {int value; Node next;}
  Node TOP;  int TOP_version;

  void push(int x) {
    Node local_top, new_top;
    int local_version;
    boolean done = false;

    new_top = new Node();
    new_top.value = x;
    while (!done) {
      atomic {
        local_top = TOP;
        local_version = TOP_version;
      }
      new_top.next = local_top;
      if ( true* ) {
        if (CAS2(TOP, TOP_version,
                 local_top, local_version,
                 new_top, local_version+1))
          done = true;
  } } }
```

```
int pop() {
  Node local_top, new_top;
  int local_version;
  boolean done = false;
  int value = EMPTY;

  while (!done) {
    atomic {
      local_top = TOP;
      local_version = TOP_version;
    }
    if ( true* ) {
      if (local_top == null) {
        done = true;
      } else {
        new_top = local_top.next;
        if (CAS2(TOP, TOP_version,
                 local_top, local_version,
                 new_top, local_version+1)) {
          done = true;
          value = local_top.value;
  } } } }
  return value; }
```

```
void harness() {
  Stack stack = new Stack();
  coforeach (i = 1 .. 30) {
    // make a call to stack
    if(randomBoolean()) {
      stack.push(randomInt());
    } else {
      int t = stack.pop();
      mark_focus(t) ;
  } }

  // traverse the stack and mark
  // the final contents relevant
  mark_focus(stack.TOP) ;
  Node node = stack.TOP;
  while(node != null) {
     mark_focus(node.next) ;
     mark_focus(node.value) ;
     node = node.next;
} }
```

Figure 8. A non-blocking stack implementation with its nondeterministic sequential specification embedded.

[32] L. Rauchwerger and D. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Programming Language Design and Implementation (PLDI)*, pages 218–232, 1995.

[33] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Programming Language Design and Implementation (PLDI)*, pages 54–67, 1996.

[34] C. Sadowski, S. Freund, and C. Flanagan. SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In *European Symposium on Programming (ESOP)*, 2009.

[35] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *Computers, IEEE Transactions on*, 40(5):603 –612, 1991.

[36] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[37] N. Shavit and D. Touitou. Software transactional memory. In *Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[38] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Supercomputing (SC)*, 2001.

[39] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

[40] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–348, 2009.

[41] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Principles and Practice of Parallel Programming (PPOPP)*, 2006.

[42] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

[43] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Principles of Programming Languages (POPL)*, pages 334–345, 2006.

[44] M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN Workshop on Model Checking Software*, pages 261–278, 2009.

[45] M. Vechev, E. Yahav, R. Raman, and V. Sarkar. Verifying determinism of structured parallel programs. In *Static Analysis Symposium (SAS)*, 2010.

[46] C. von Praun and T. R. Gross. Object race detection. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 70–82, 2001.

[47] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 79–89, 2007.

[48] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32:93–110, 2006.

[49] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2):164–182, 1993.

A. NDSeq Specification Examples

A.1 Non-blocking Concurrent Reduction

Consider the simple parallel program in Figure 9. The program consists of a parallel for-loop, denoted `coforeach`—each iteration of this loop attempts to perform a computation (Line 6) based on the shared variable `x`, which is initially 0. In particular, each iteration uses an atomic compare-and-swap (`CAS`) operation to update the shared variable `x`. If multiple iterations try to concurrently update `x`, some of these `CAS`'s will fail and those parallel loop iterations will recompute their updates to `x` and then will try again.

The NDSeq specification of the program, which is embedded in the parallel program in Figure 9, indicates two nondeterministic aspects. First, the box around the **coforeach** construct in Line 1 specifies that the loop iterations can run in any permutation of the set `1,...,N`. This part of the specification captures the intended nondeterministic behavior of the parallel program: `x` can be updated by threads in an arbitrary order due to nondeterministic scheduling of threads. Second, the `if(`*true*∗`)` annotation in Line 4 specifies that

```
 1:  coforeach  (i in 1,...,N) {
 2:    bool done = false;
 3:    while (!done) {
 4:        if ( true* ) {
 5:            int prev = x;
 6:            int curr = i*prev + i;
 7:            bool c = CAS(x,prev,curr);
 8:            if (c) {
 9:                done = true;
10:            }
11:  } }   }
12:  mark_focus(x) ;
```

Figure 9. Simple parallel program to perform the reduction in Line 6 for the integers {1,...,N}, in some arbitrary order.

the iteration body may be skipped nondeterministically, at least from a partial correctness point of view; this is acceptable, since the `while` loop in this program fragment is already prepared to deal with the case when the effects of an iteration are ignored following a failed CAS statement.

The $\boxed{\texttt{mark\_focus}}$ annotation in Line 12 indicates that `x` is the only focus variable. That is, the functional correctness of the program depends only on the final value of `x` after all the threads created by `coforeach` terminate.

A.2 Non-blocking Concurrent Stack

In Figure 8, we give the Java-like code for the NDSeq specification of our non-blocking **`stack`** benchmark. The stack is represented as a `null`-terminating, singly-linked list of `Node` objects. The head of the list is pointed by the `TOP` field of the stack. In order avoid the ABA problem, we use a version number (`TOP_version`), which is increased whenever `TOP` is updated.

The `push` and `pop` methods implement the Optimistic Concurrent Computation pattern in Section 4 using a loop that iterates until the operation succeeds. Each method first reads the `TOP` of the stack without any synchronization and then uses an atomic `CAS2` (double compare-and-swap) operation to check for conflicts by comparing `TOP` and `TOP_version` with `local_top` and `local_version`. If `TOP=local_top` and `TOP_version=local_version` then `CAS2` commits the operation by writing `new_top` to `TOP` and incrementing `TOP_version` and returns *true*. Otherwise, `CAS2` returns *false* and the operation retries.

In the specification, an **`if(`***true∗***`)`** statement is placed around the critical `CAS2` in `push`, and around the check whether or not `local_top` is `null` in `pop`. These **`if(`***true∗***`)`**'s indicate that the sequential version of the program is free to nondeterministically retry as if there had been a conflict. This added nondeterminism enables our dynamic analysis to mark as irrelevant the reads of `TOP` and `TOP_version` by loop iterations in which the `CAS2` fails.

The `harness` method creates a number of threads, each of which calls `push` with a random input or calls `pop`. In the code, we mark the focus variables using $\boxed{\texttt{mark\_focus}}$. The focus variables are: (1) the values popped by the harness threads (marked after each call to `pop`), and (2) the last contents of the stack (marked at the end of the harness).