ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs

Felix Suchert
TU Dresden, Germany

Lisza Zeidler
Barkhausen Institut, Dresden, Germany

Jeronimo Castrillon
TU Dresden, Germany

Sebastian Ertel[1]
Barkhausen Institut, Dresden, Germany

Abstract

SAT/SMT-solvers and model checkers automate formal verification of sequential programs. Formal reasoning about scalable concurrent programs is still manual and requires expert knowledge. But scalability is a fundamental requirement of current and future programs.

Sequential imperative programs compose statements, function/method calls and control flow constructs. Concurrent programming models provide constructs for concurrent composition. Concurrency abstractions such as threads and synchronization primitives such as locks compose the individual parts of a concurrent program that are meant to execute in parallel. We propose to rather compose the individual parts again using sequential composition and compile this sequential composition into a concurrent one. The developer can use existing tools to formally verify the sequential program while the translated concurrent program provides the dearly requested scalability.

Following this insight, we present *ConDRust*, a new programming model and compiler for Rust programs. The *ConDRust* compiler translates sequential composition into a concurrent composition based on threads and message-passing channels. During compilation, the compiler preserves the semantics of the sequential program along with much desired properties such as determinism.

Our evaluation shows that our *ConDRust* compiler generates concurrent deterministic code that can outperform even non-deterministic programs by up to a factor of three for irregular algorithms that are particularly hard to parallelize.

2012 ACM Subject Classification Theory of computation → Parallel computing models; Software and its engineering → Parallel programming languages

Keywords and phrases concurrent programming, verification, scalability

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.33

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.16`

Funding *Felix Suchert*: was funded by the EU Horizon 2020 Programme under grant agreement No 957269 (EVEREST).
Lisza Zeidler: was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 469256231.

Acknowledgements The authors would like to thank the anonymous reviewers for their invaluable feedback in the submission process.

[1] Corresponding author

© Felix Suchert, Lisza Zeidler, Jeronimo Castrillon, and Sebastian Ertel;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 33; pp. 33:1–33:39

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

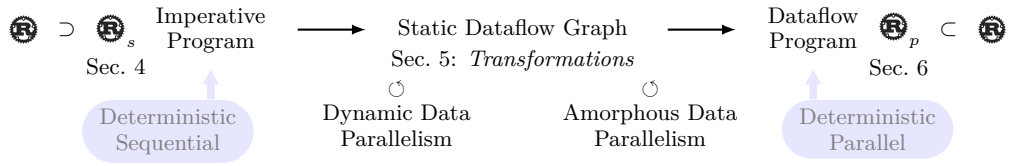


Figure 1 The *ConDRust* compiler translates imperative programs written in $_s$, a subset of Rust for **s**equential composition, into dataflow programs in $_p$, a subset of Rust for **p**arallel composition.

1 Introduction

Formal verification of sequential programs can be automated to a large extent which makes it ready for widespread adoption. Verification of concurrent multi-core shared-memory programs, instead, can only be automated to some extent and requires expert knowledge. This is a major hurdle for safe systems which must rely on scalable parallelism to overcome the physical boundary in current and future processors.

A formally verified program carries mathematical proof that certain properties of the program hold. Determinism is such an interesting property. Deterministic programs are straightforward to debug [12]. A deterministic execution increases the time predictability of IoT systems [50] and establishes latency boundaries of service-level agreements in the cloud [20, 4]. In recent database management systems with transaction support, determinism removes costly synchronization and challenging distributed failure scenarios [52, 40].

Formal verification of program properties, such as determinism, proceeds along two directions. Proof assistants such as Coq allow expressing properties in higher-order logic but require a manual proof from the developer. SAT/SMT-based verifiers such as Prusti and model checkers such as Kani for Rust programs are restricted to properties formulated in first-order logic but calculate the proof automatically [3, 54]. That is, the hard part of formal verification is automated which makes them particularly interesting for widespread adoption. But concurrent programs require separation logic to state and prove their properties [51, 43]. Encoding separation logic into first-order logic is still ongoing research [15, 14, 44, 43] and needs to sacrifice important (higher-order) parts. Expert knowledge in Coq is required to take full advantage of separation logic [30]. At the time of this writing, none of the formal verification tools for Rust programmers supports reasoning about concurrent programs.

Our insight is that two main steps are needed to translate a sequential program into a concurrent one. We call these steps *Decompose* and *Recompose*. The Decompose step breaks the sequential composition of a program to create independent parts that can execute in parallel. The Recompose step composes these parts again using concurrency abstractions such as threads and synchronization primitives such as locks or message-passing. We refer to this as *concurrent composition*. In programming models such as threads with locks, message-passing or software transactional memory (STM), the developer has to perform both steps manually. Automatic approaches to tackle both steps require a precise points-to analysis to create a concurrent program without concurrency hazards such as data races or deadlocks. This analysis is known to be undecidable in general [48]. Hence, researchers resort to speculative approaches [17] or language constraints for a precise dependence analysis [6].

The *ConDRust* approach

In this paper, we propose *ConDRust*, a new *sequential* programming model for the *concurrent composition* in the Recompose phase. Our *ConDRust* compiler translates a sequential composition into a concurrent one automatically. This allows for testing and formal verification to be performed on the sequential program, with guarantees that are carried into the concurrent

one. More specifically, our current prototype compiler supports a well-defined subset of *safe* Rust for sequential composition and generates concurrent dataflow composition in a well-defined subset of *safe* Rust with threads and message-passing. The dataflow execution model is the runtime representation for scalable parallelism in many domains such as embedded systems, database systems and machine learning frameworks [38, 23, 27, 58]. Our compiler design is based on rewriting steps that preserve the semantics of the sequential input program. This includes the semantics of control flow constructs such as loops but also properties such as determinism. Formally verifying the compiler is a larger effort [39] that is beyond the scope of this paper. This paper, instead, investigates to what extent the programming model is applicable to real-world programs and whether the compiler can generate scalable concurrent composition that is at least on par with existing concurrent programming models. Throughout the paper, we point to novel and interesting research directions that our approach introduces.

Concretely, we make the following contributions:

- The main contribution of this paper is a new programming model and compiler for the compositional fragment, i.e., subset, of safe Rust (Sections 2 and 3).
- We define $\circledR_s$, a subset of Rust for **s**equential imperative composition with abstractions, calls, variables but without references, and formally specify its type system and operational semantics (in Section 4 and the Appendix).
- We formally specify $\circledR_p$, the subset of Rust for concurrent composition that the *ConDRust* compiler targets in Section 6. The appendix contains the operational semantics, type system and a proof sketch for the deterministic execution.
- The key part of our compiler, visualized in Figure 1, is a transformation of sequential imperative $\circledR_s$ composition into a functional representation based on the well-defined concept of state threads [35]. Our compiler lowers this functional representation into dataflow, a well-established abstraction for parallel execution [5, 2, 37, 29] (Section 5.1).
- Key to scalable concurrent composition are two transformations to exploit data parallelism even in stateful applications with (tail) recursion (Sections 5.2 and 5.3).
- Our current *ConDRust* prototype compiler consists of ca. 20K lines of Haskell code that can be lifted to Coq in future work to formally-verify semantic preservation (Section 7).
- To provide a fair baseline with the same guarantees, we re-implemented a deterministic STM (DSTM) algorithm [49] for an existing STM implementation in Rust. We ported 7 benchmarks from 3 different benchmark suites and provide implementations for sequential, threads/DSTM, threads/STM and *ConDRust* (ca. 12K lines of Rust code). Our evaluation in Section 8 shows that *ConDRust* produces programs that outperform all threads/DSTM and even some of the non-deterministic threads/STM programs by up to a factor of 3. We highlight directions for future work whenever *ConDRust* programs do not scale.

We review related work in Section 9 and conclude in Section 10.

2 Programming for Scalability: Decompose and Recompose

Before we introduce our programming model in more detail, we reflect on concurrent programming. We argue that concurrent programming and our *ConDRust* programming model contain the same *Decompose–Recompose* steps. That is, the reasoning for the developer to prepare a scalable concurrent program is the same as for writing a *ConDRust* program. But the implications are different. Figure 2 uses an inarguably contrived but easy to follow example for a side-by-side comparison. The left column shows the sequential program. The middle column lists the program with threads and atomics, i.e., transactions on a single variable. And the right column shows the sequential *ConDRust* program.

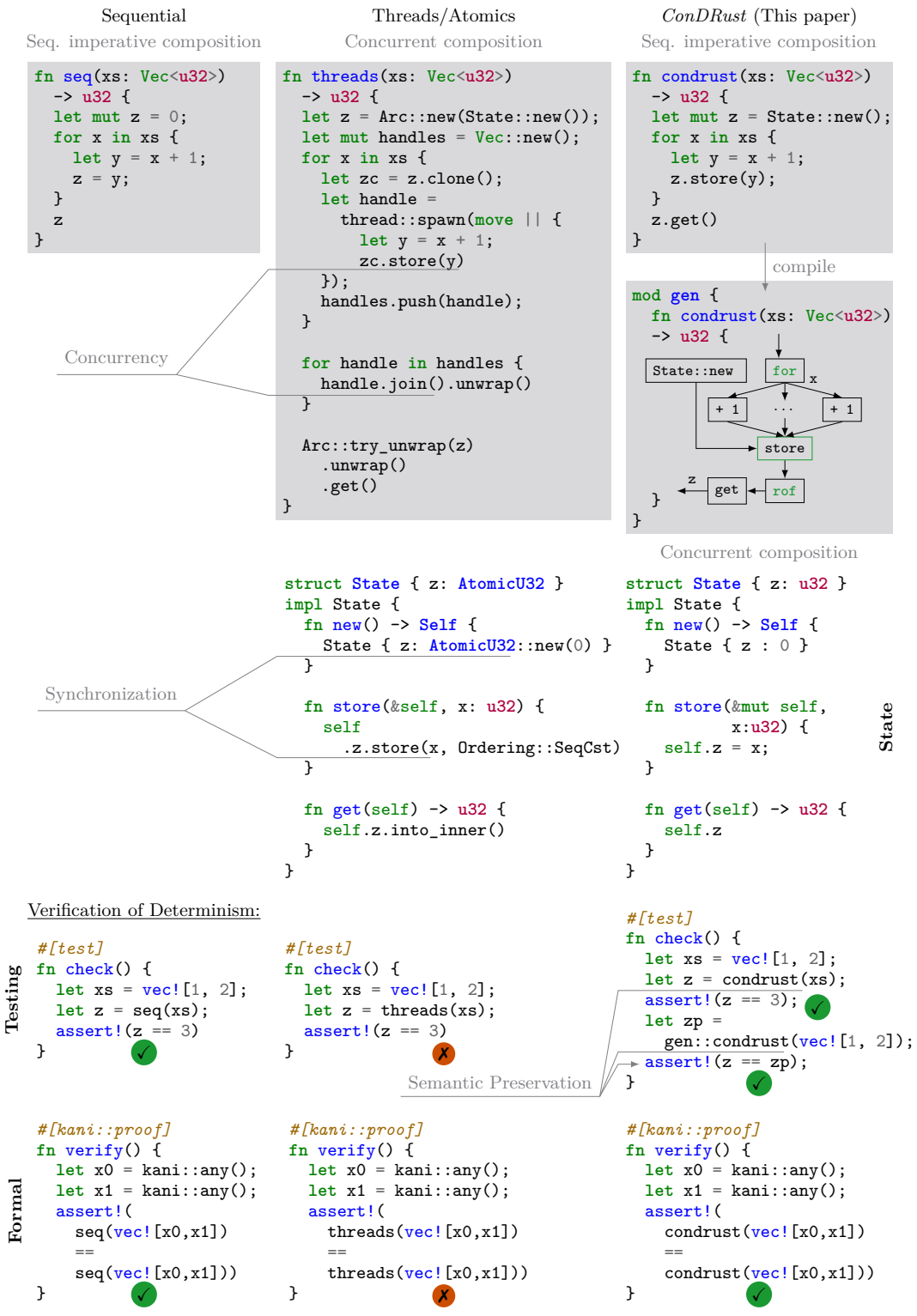


Figure 2 Comparison between sequential, concurrent and *ConDRust* programs and their properties. The sequential program executes deterministically and is amenable to verification. The concurrent program offers parallel speedup but compromises determinism and verifiability. The *ConDRust* approach preserves determinism and verifiability and compiles the program into a concurrent dataflow for scalable parallel execution.

The sequential program on the left iterates over a vector of numbers `xs`. For each number, the program computes its increment and assigns it to a shared state `z`. The resulting value of `z` is always the increment of the last number in `xs`. This is by definition of the sequential execution order of statements and loop iterations in Rust. As such, we can check deterministic execution with a simple test case for a vector with 2 elements and let Kani formally verify this property in a simple proof harness (see bottom of left column in Figure 2). Apart from detecting potential overflow, which is inconsequent for this example, Kani succeeds.

To arrive at a scalable concurrent version of the program, the developer needs two steps. In the *Decompose* step, the developer identifies the parts of the program that can/should be executed in parallel and the state that needs to be shared. In our example, the developer selected the body of the loop for parallel execution. Any approach, that seeks to automate this step, needs to solve the thread granularity problem [47, 1]. Tools now assist in identifying the state [21]. The focus of this paper is on the second step. In the *Recompose* step, the developer replaces the sequential composition with a *concurrent composition*, e.g., for the loop iterations. Prevalent concurrent programming models for imperative programs consist of two parts for composition: *concurrency abstractions* and *synchronization primitives* to access (shared) state. The concurrent version of the program spawns a thread for each computation on the elements of the input vector `xs`. Thereby, it removes the sequential execution order of the `for` loop. Accesses to the state variable `z` now need to be protected with atomic operations (or transactions) to prevent data races. But this protection cannot recover the deterministic update order on `z`, even when opting for the strongest and least performing memory ordering: sequential consistency (`SeqCst`). The test's post-condition now may see `z == 2`. That is, the determinism property of the program is lost. Even worse, Kani cannot even help to detect this flaw because verification of concurrent programs needs a higher-order (separation) logic which is (currently) out of reach for model checkers like Kani[2].

To construct a *ConDRust* version of the program, the developer has to follow the same two steps. The *Decompose* step is the same as in the construction of the concurrent version. The developer identifies independent parts and shared state. In the *Recompose* step, the developer uses the sequential composition constructs of the host language such as for example statements, `for` loops, function calls and (imperative) method calls. The *ConDRust* program defines the update to the state `z` with a method rather than an assignment. Assignments are not supported in this paper because we tried to keep 🐵_s minimal but can be added easily. The program is free of concurrency abstractions and synchronization primitives. The deterministic property on the result of the program can be tested and formally verified in the same way as for the sequential version. Note that formal verification is performed on the sequential `condrust` program. The *ConDRust* compiler is aware of the semantics of the composition constructs and preserves them during compilation. The generated dataflow graph (in `gen::condrust`) exhibits data parallelism for the computation of the increments and pipeline-parallelizes this with the update of the state `z`. The state update order is preserved. We can then dynamically verify that the generated concurrent program `gen::condrust` preserves the semantics, including the determinism property.

Concurrency vs. Parallelism. Throughout the paper, we use the words *concurrent* and *parallel* in the following well-established sense [53]. Concurrency means interleaved execution of computations. Parallelism exploits additional (multi-core) hardware to execute computations simultaneously. That is, concurrency does not necessarily imply parallelism. But in this

[2] `https://model-checking.github.io/kani/rust-feature-support.html#concurrency`

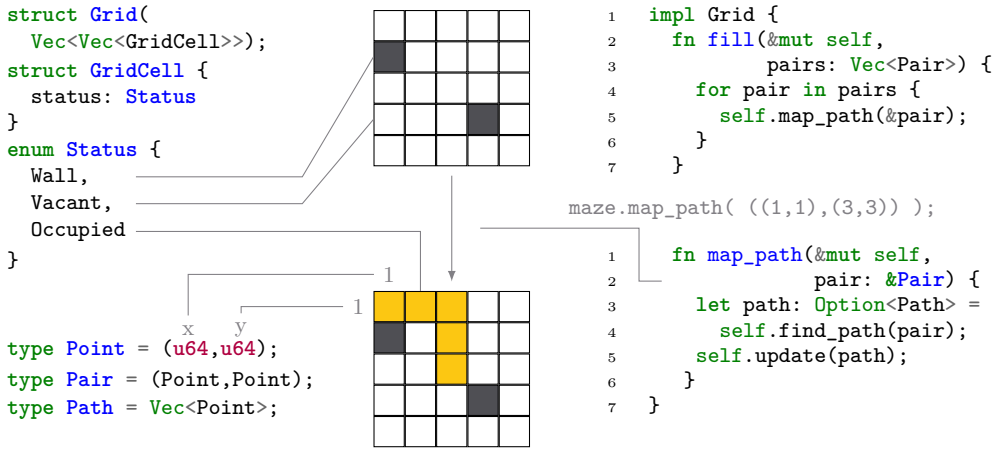


Figure 3 Illustration of the labyrinth benchmark on a 2D grid in imperative sequential Rust.

paper, we focus on scalable concurrency. Scalable concurrency assumes multi-core hardware to turn explicit concurrency in the program into implicit parallelism. Hence, whenever we refer to parallelism, we mean independent concurrent computations in the program. ◀

3 The *ConDRust* programming model

In this section, we present the *ConDRust* programming model that we formally specify and embed into the Rust programming language (see Section 4). We compare programming in *ConDRust* to concurrent programming with threads and software transactional memory (STM). As a running example, we use Labyrinth, an irregular application from the STAMP benchmark suite [41]. Irregular programs contain algorithms where concurrent programming models particularly shine because such algorithms are notoriously hard to parallelize at compile-time. We start from the sequential version of the labyrinth algorithm and then develop a version based on threads and STM to compare it against programming in *ConDRust*.

3.1 The Labyrinth benchmark

The labyrinth benchmark implements Lee's algorithm to find wire-paths between points on a multi-layer printed circuit board [36]. The challenge consists in finding paths that do not overlap across layers. The original STAMP implementation and our re-implementation in Rust execute on a 3D board (or 3D grid). In this section, we restrict ourselves to a 2D grid because it is easier to visualize while retaining the core principles required by our exposition.

Figure 3 illustrates the problem and shows a sequential imperative Rust implementation. The left-hand side defines a grid as a vector of vectors where a grid cell holds one of the three states: wall (in dark grey), vacant (in white) or occupied (in yellow). A point in the grid is a tuple of an x- and a y-coordinate. A pair is a tuple of two points and a path is a sequence of points represented as a vector. The algorithm to `fill` a given grid is given on the right-hand side of the figure. For each pair, the function `map_path` finds a path and declares the corresponding grid cells as occupied. Note that both steps `find_path` and `update` require access to the grid. That is what makes introducing concurrency particularly challenging for the developer.

3.2 Concurrent Labyrinth

Concurrently mapping paths onto the grid requires changes to the algorithm. Two candidate paths computed concurrently may overlap and thus one of them has to be re-computed. We visualize this effect in the middle column of Figure 4 and compare the threads/STM version in the left column with the *ConDRust* version in the right column. These alternative implementations are discussed in the following.

3.2.1 Threads/STM

We use threads for concurrency and synchronize grid access via STM. We chose STM over locks for two reasons. First, STM guarantees data-race freedom without creating deadlocks. Second, deterministic STM algorithms exist that provide the same deterministic execution properties as *ConDRust*. For this paper, we used `rust-stm`[3], an STM implementation in Rust, that follows the design of the STM in Haskell [28].

The threads/STM implementation in the left-hand column of Figure 4 starts with a re-definition of the grid data structure. Cells hold the state of the grid that the algorithm mutates and hence must be protected to prevent data races. Wrapping the cells into transactional variables (`TVars`) means that all grid methods need to be redefined for two reasons: First, accesses are now through the `TVar`. Second, each access needs a transaction which is an additional parameter to every grid method. That is the reason why all benchmarks in STAMP are implemented twice: with and without STM.

Concurrency is introduced in Line 7 of the Threads/STM implementation. We deliberately assume a higher-order function `spawn_join` to abstract from the verbose spawn-join pattern given in Figure 2. For every pair, the closure from Line 7–10 tries to map a path on the grid. The combination of the loop over the pairs and the closure's `move` semantics for threads demand a clone of the grid for every loop iteration. The `update` method of the new `StmGrid` then tries to occupy the corresponding cells on the grid for a computed path. When a path cell is already occupied, a retry error (Line 30) aborts the transaction and triggers a retry.

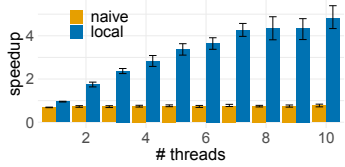
For this particular example, the `map_path` method is particularly important for scalability reasons. The general structure of `map_path` is straightforward. It consists of a closure with the two steps of the algorithm (Lines 39–46): finding a path (Line 44) and updating the grid (Line 45). Line 39 places the closure onto a transaction. The key to scalability lies in the updates to the shared state. A naive implementation for `find_path`, as shown in the comment on Line 42, would operate directly on the grid (`StmGrid`) through the `TVar` (see Line 24). As a result, the transaction's read set increases dramatically which leads to a much higher probability for collisions in `update`. A common pattern is thus to have the path-finding operate on a non-transactional clone of the grid (`local`). To illustrate the effect, the bar plot below the `map_path` function compares the scalability of the naive version to that of the optimized one.

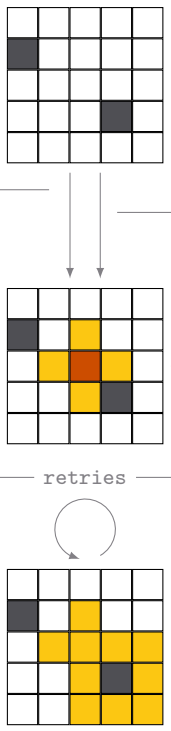
The function `map_path` is representative of a common pattern found in irregular applications. The optimization described above requires the developer to find the maximal set of accesses that still preserves data-race freedom. This, in turn, requires carefully distinguishing the parts of the program that have side-effects on the state from the ones that are pure. Making this distinction explicit is a core idea behind the *ConDRust* programming model.

[3] `https://github.com/Marthog/rust-stm`

threads/STM

```
struct StmGrid(
  Vec<Vec<TVar<GridCell>>>);
fn fill(grid : StmGrid,
        pairs: Vec<Pair>
) -> StmGrid {
  spawn_join(
    | grid_clone,
      pair | {
      grid_clone.map_path(pair)
    },
    &grid,
    pairs
  );
  grid
}
impl StmGrid {
  fn update(&self,
    path: Option<Path>,
    tx: &mut Transaction
  ) -> StmResult<()> {
    if let Some(p) = path {
      for (x,y) in p {
        let gc =
          self.0[x][y].read(tx)?;
        if gc = Status::Vacant {
          self.0[x][y]
            .write(
              tx,
              Status::Occupied)?;
        } else {
          return Err(StmError::Retry);
        }
      }
      Ok(())
    } else {
      Ok(())
    }
  }
  fn map_path(&self, pair: Pair) {
    atomically(| tx | {
      /* naive:
      let path = self.find_path(&pair, tx); */
      let local: Grid = self.local_clone();
      let path = local.find_path(&pair);
      self.update(path, tx)
    })
  }
}
```





ConDRust

Sequential imperative composition

```
fn fill(grid : Grid,
        pairs: Vec<Pair>) {
  let retries = Vec::new();
  let shared =
    Arc::new(grid.clone());
  for pair in pairs {
    // map_path:
    let local = shared.clone();
    let path =
      find_path(local, pair);
    let retry =
      grid.update_c(path);
    retries.push(retry);
  }
  let not_empty =
    retries.filter_some();
  if not_empty {
    fill(grid, retries)
  } else {
    grid
  }
}
```

Stateless Function

```
fn find_path(
  grid : Arc<Grid>, pair: Pair
) -> Option<Path> {
  grid.find_path(&pair)
}
```

Stateful Function

```
impl Grid {
  fn update_c(&mut self,
    paths: &Vec<Option<Path>>
  ) -> Option<Pair> {
    for path in paths {
      if self.is_vacant(path) {
        self.update(path);
        None // success
      } else {
        Some( // retry
          ( path[0]
          , path[path.len() - 1]
          ))
      }
    }
  }
}
```

Figure 4 Introducing concurrency into the labyrinth benchmark adds collisions. The left column shows the implementation with threads and STM. The right column shows the imperative sequential *ConDRust* program.

3.2.2 *ConDRust*

The right-hand column in Figure 4 presents the *ConDRust* version of the labyrinth benchmark. We start with an informal introduction to the *ConDRust* programming model. We then explain how concurrency and synchronization arise naturally from a *ConDRust* program while preserving determinism and verifiability.

3.2.2.1 Programming model

A *ConDRust* program consists of three abstractions: *functions*, *stateless function calls* and *stateful function calls*.

▶ **Definition 1** (*ConDRust* Functions). *A* ConDRust *function is a top-level function or an anonymous (lambda) function definition in the host language that the* ConDRust *compiler translates into a concurrent dataflow graph.*

We highlight the *ConDRust* function for `fill` and `map_path` with a gray background. In the actual code base, the two functions would be located in a dedicated Rust module that is input to the *ConDRust* compiler. A *ConDRust* function may use control flow constructs of the host language, call other *ConDRust* functions or call stateless/stateful functions. We focus on loops because conditionals are rather unimportant when it comes to concurrency.

▶ **Definition 2** (Stateless Function Call). *A stateless function call is a (host language) term of the form* $f(t_1, \ldots, t_n)$ *where* $t_1, \ldots, t_n$ *are terms and* f *is a function symbol.*

▶ **Definition 3** (Stateful Function Call). *A stateful function is a (host language) term of the form* $t_s.f(t_1, \ldots, t_n)$ *where* $t_s, t_1, \ldots, t_n$ *are terms and* f *is a function symbol.*

Stateless functions such as `find_path` (Lines 23–27) represent pure computations to the *ConDRust* compiler. Stateful functions such as `update_c` (Lines 29–43) represent computations with side-effects to a particular state. The implementation of the stateless and stateful functions are outside the realm of the *ConDRust* compiler. That is, the developer may use the full set of the host language's, i.e., Rust's, features inside these functions without violating the stateless/stateful semantics.

The *ConDRust* programming model is general enough for embedding it into other languages such as Java or Python. But Rust is particularly well-suited because it allows to enforce state encapsulation via its type system. That is, the *ConDRust* compiler can reject programs with stateful function calls that return references to their state. Our Rust embedding does not support references as of now. Moving forward, we are interested in *ConDRust* in the context of share-nothing software architectures (e.g., serverless computing), where references do not exist. In the shared-memory context of this paper, the developer can either pass data by value or by reference using Rust's `Arc`'s. In Section 4, we present the details of the type system for *ConDRust*.

3.2.2.2 Concurrency and synchronization

The *ConDRust* compiler translates the input program into a concurrent dataflow graph. In this graph, pipeline and task-level parallelism arise naturally from the data and control flow dependencies in the program. Our transformation, defined in Section 5, introduces data parallelism for stateless function calls inside loops. In the labyrinth benchmark, the `find_path` computation is a good candidate. It is located inside the loop of `fill` function and actually does not have side effects on the grid. But in the sequential version of Figure 3,

`find_path` is a method on the grid. To tell the *ConDRust* compiler that this computation is stateless, the developer turns the method into a stateless function (Lines 23–27). At Line 26, the stateless `find_path` function just wraps the `find_path` method on the `Grid` data type.

Note that the *ConDRust* version of `map_path` (right column, Lines 7–12) is almost identical to the STM/threads version (left column, Lines 38–46). At Line 8, the *ConDRust* compiler requires the developer to create a clone of the grid. State, in this case the `grid`, may only be used once inside a loop, i.e., the loop in `fill` at Lines 6–11. The *ConDRust* compiler forces the developer into the optimization that the STM/threads version needed to scale. As a consequence, the new `update_c` method on the `Grid` now needs to take collisions and retries into account. This is almost identical to the `StmGrid::update` method but is not based on a transaction for retrying. Instead, it returns the colliding pair at Lines 37–40.

After the loop, the remaining code in `fill` first filters the successfully mapped paths (Line 16) and then uses a (tail) recursion to retry the collisions. That is, the `fill` function explicitly defines the recursion that is implicit in the concept of a transaction. But the execution semantics is different.

3.2.2.3 Determinism

Transaction execution order in STM is non-deterministic but the generated dataflow program executes deterministically. The *ConDRust* program presented in the right-hand column of Figure 4 is a valid, i.e., well-typed, sequential Rust program. The Rust compiler translates this into a deterministically executing binary. The *ConDRust* compiler preserves the semantics of the program including the deterministic execution property. For the labyrinth benchmark, this boils down to the sequential order in which the stateful function `update_c` is called within the loop in `fill`. The *ConDRust* compiler preserves this order even though the paths are computed concurrently (in a data-parallel fashion).

In Section 5, we define a transformation to take advantage of amorphous data parallelism in irregular algorithms. This transformation produces code that limits the number of collisions, and thereby re-computations, for a single recursion round. To activate this transformation, the developer has to change the type of the input `pairs` from an ordered vector `Vec<Pair>` into a `HashSet` with deterministic iteration order. All of the transformations that we introduce in this paper preserve determinism and the semantics of the input program.

3.2.2.4 Verification

Semantic preservation allows the developer to apply and even formally verify further optimizations to the algorithm. The *ConDRust* version of the labyrinth benchmark is free of concurrency constructs. Kani fully supports `Arc`'s[4]. As such, the developer can formally verify properties of the *ConDRust* labyrinth implementation.

Zero-clone concurrent labyrinth. Note that both the threads/STM and the previously discussed *ConDRust* versions required cloning the state for scalability reasons. By cleverly using *ConDRust* and the underlying Rust semantics it is possible to avoid having to clone in the first place, while retaining determinism and verifiability. This is again a pattern that extends to other irregular applications.

The zero-clone implementation for concurrent path-finding is shown in Figure 5. For the reader's reference, the comments in the code contain the previous implementation from Figure 4. On the left-hand side, we restate the *ConDRust* version of the `fill` function from

[4] `https://model-checking.github.io/kani/rust-feature-support.html`

```
1  fn fill(grid : Grid, pairs: Vec<Pair>) {
2    let /* retries */ paths = Vec::new();
3    let shared =
4      Arc::new(grid/*.clone()*/);
5    for pair in pairs {
6      let local = shared.clone();
7      let path = find_path(local, pair);
8      /* let retry =
9           grid.update_c(path);
10        retries.push(retry); */
11     paths.push(path);
12   }
13   /* let not_empty =
14     retries.filter_some(); */
15   let (paths, grid) = unarc(paths, shared);
16   let (not_empty, retries) =
17     grid.updates(paths);
18   if not_empty {
19     fill(grid, retries)
20   } else {
21     grid
22   }
23 }
```

```
1  impl Grid {
2    fn updates(
3      &mut self,
4      paths: Vec<Option<Path>>
5    ) -> (bool, Vec<Pair>) {
6      let mut retries = Vec::new();
7      for path in paths {
8        let r = self.update_c(path);
9        retries.push(r)
10     }
11     let not_empty =
12       retries.filter_some();
13     (not_empty, retries)
14   }
15 }
16 fn unarc<S,T>(
17   s:S, t: Arc<T>
18 ) -> (S,T) {
19   match Arc::<T>::try_unwrap(t) {
20     Ok(t) => (s,t),
21     _ => panic!("Failed to unarc.")
22   }
23 }
```

Figure 5 The `unarc` optimization provides a *ConDRust* version of the labyrinth benchmark that does not have to clone the grid.

Figure 4. We extracted the updates to the grid from Lines 8–9 to Lines 16–17 after the loop. This requires defining a new method (/stateful function) `updates` on the `Grid` that performs the loop over the computed paths. The corresponding code in the upper right part of Figure 5 also directly filters the retries. At this point, the key observation is that we can safely reuse the grid after the loop. At Line 4, `Arc::new` takes ownership of the grid. We had to clone the `grid` to use it inside the loop for updates and after the loop for the recursion. In the new version, the updates happen after the loop and the `find_path` actually takes ownership of the cloned `Arc` from Line 6. That is, when all path computations are done, we can safely take the `grid` out of the `Arc` again. This is what we specify at Line 15 and in the *ConDRust* version of Figure 4 at Line 24.

Figure 5 defines the stateless function `unarc` in the lower right part. The `unarc` function unpacks the `Arc`, its first argument, but leaves the second unchanged. We have to add a `panic` for the case where the `Arc` is still held elsewhere. But this is impossible by definition of Line 15. We verified this property and among the 422 reachable checks Kani reports:

```
Check 321: <std::vec::Vec<std::option::Option<benchs::Path>>
           as benchs::Unarc>::unarc.assertion.1
- Status: SUCCESS
- Description: "Failed to unarc."
- Location: src/benchs.rs:269:18
   in function <std::vec::Vec<std::option::Option<benchs::Path>>
   as benchs::Unarc>::unarc
```

Due to the semantic preserving transformations in *ConDRust*, this property also holds for the generated concurrent dataflow code.

$$
\begin{array}{lll}
\text{Terms} & t & ::= \; x \mid v \mid \texttt{|}x : T\texttt{|->}\ T\ \{\ t\ \} \mid t(t) \mid \textbf{let}\ x : T = t; t \mid \textbf{let mut}\ x : T = t; t \mid \\
 & & \quad \mathrm{f}_{\mathrm{SL}}(t_1) \mid t_s.\mathrm{f}_{\mathrm{SF}}(t_1) \mid \textbf{for}\ x\ \textbf{in}\ t\ \{\ t\ \} \mid \textbf{trfix}\ t\ t \\
\text{Values} & v & ::= \; l \mid v_{\textcircled{R}_s} \mid \texttt{|}x : T\texttt{|->}\ T\ \{\ t\ \}
\end{array}
$$

Figure 6 Syntactical constructs of $\textcircled{R}_s$.

Note that this `unarc` optimization is only necessary because the compiler presented in this paper does not support references. Support for references is future work but will remove such optimizations from the code and move them into the compiler. In the meantime, the developer can formally verify such optimizations with existing tools such as Kani.

4 $\textcircled{R}_s$– A subset of Rust for sequential composition

In this section, we formally specify $\textcircled{R}_s$, the subset of the Rust language that embeds the *ConDRust* programming model. Since $\textcircled{R}_s$ is a subset of Rust, the operational semantics are the same as for Rust. Therefore, we restrict the presentation to the syntactical constructs. The appendix defines the type system and the operational semantics for the interested reader.

ConDRust supports the subset of Rust's syntax that is necessary to compose calls to stateless and stateful functions (also called methods). We define this subset in Figure 6 as $\textcircled{R}_s$– a subset of Rust for **s**equential composition. For this paper, we restrict the terms of the language to variables x, abstractions (closures in Rust) $\texttt{|}x : T\texttt{|->}\ T\ \{\ t\ \}$, algorithm application $t(t)$, immutable and mutable bindings, **for**-loops and tail-recursion (**trfix**). We restrict the definition of $\textcircled{R}_s$ in the following (common) ways:

1. Abstractions and calls may only have a single parameter. The extension to support multiple parameters is straightforward.
2. We desugar top-level algorithm definitions into `let`-bound closures such that a top-level defined function can be used in multiple locations of succeeding function definitions.

The key ingredient in *ConDRust*'s programming model are *stateless function calls* $\mathrm{f}_{\mathrm{SL}}(t_1)$ and *stateful function calls* $t_s.\mathrm{f}_{\mathrm{SF}}(t_1)$.[5] The definition of stateless and stateful functions is not part of $\textcircled{R}_s$, as discussed before. Inside these functions, developers re-gain the full features of Rust. We further restrict control flow to loops and tail recursion leaving out other forms such as conditionals that play only a minor role in the parallel execution of a program. In Section 5 and in our implementation, loops may in fact iterate over all data types that implement Rust's `Iterator` trait which for instance includes `HashSet`. This allows the developer to specify that the loop does not depend on a particular order and enables our second transformation that extracts amorphous data parallelism. To model this in $\textcircled{R}_s$, we assume a stateless function that uses the iterator to collect the items into a list before looping over them. In fact, `collect` is a standard function of Rust's `Iterator`. We allow loops with an unknown iteration count via tail recursion. Tail recursion is a derived form. Precise definitions with their restrictions on variable usage are in the appendix. In the context of this paper, we are particularly interested in the case where the arguments to the recursion are a state to be updated and a worklist that triggers these updates.

[5] We allow *ConDRust* algorithms to be called from anywhere in a Rust program. Such a call may have arguments.

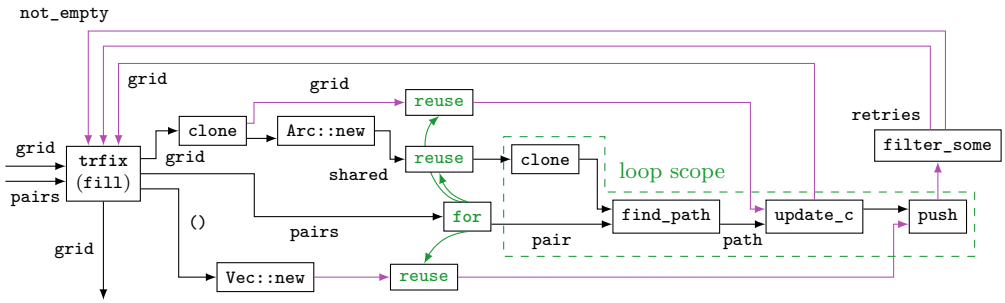


Figure 7 *ConDRust* dataflow graph for the (unoptimized) labyrinth code listed in Figure 4. The graph contains state arcs that transfer state into and out of the loop scope. The for and reuse nodes make sure that processing inside the loop is well-balanced, i.e., does not get stuck.

$$
\begin{array}{lll}
\downarrow_{\mathrm{ST}} \ \textbf{let}\ x = x_s.f(x_1); t & := & \textbf{let}\ (x'_s, x) = x_s.f(x_1);\ \downarrow_{\mathrm{ST}}\ [x_s \mapsto x'_s]t \\
\downarrow_{\mathrm{ST}} \ \textbf{let}\ x = f(x_1); t & := & \textbf{let}\ x = f(x_1);\ \downarrow_{\mathrm{ST}}\ t \\
\downarrow_{\mathrm{ST}} \ \textbf{let}\ \_ = \textbf{for}\ x_1\ \textbf{in}\ x_2\ \{\ t_3\ \};\ t_4 & := & \textbf{let}\ \_ = \textbf{for}\ x_1\ \textbf{in}\ x_2\ \{\downarrow_{\mathrm{ST}} t_3\};\ \downarrow_{\mathrm{ST}} t_4 \\
\downarrow_{\mathrm{ST}} \ |x : T|\text{->}\ T\ \{\ t\ \} & := & |x : T|\text{->}\ T\ \{\downarrow_{\mathrm{ST}} t\ \} \\
\downarrow_{\mathrm{ST}} \ t & := & t
\end{array}
$$

$$
\begin{array}{lll}
\uparrow^{\mathrm{STL}} & \textbf{let}\ \_ = \ \textbf{for}\ x_1\ \textbf{in}\ x_2\ \{\uparrow^{\mathrm{STL}} & \\
& \quad t_3;\ \textbf{let}\ (x'_s, x_3) = x_s.f(x_1);\ t_4 & := \\
& \}\ ;\ \uparrow^{\mathrm{STL}} t & \\
\uparrow^{\mathrm{STL}} & (\uparrow^{\mathrm{STL}} t) & := \ (\uparrow^{\mathrm{STL}} t)
\end{array}
\qquad
\begin{array}{l}
\textbf{let}\ (x'_s) = \ \textbf{for}^{\star}\ x_1\ \textbf{in}\ x_2\ \{\uparrow^{\mathrm{STL}} \\
\quad t_3;\ \textbf{let}\ (x'_s, x_3) = x_s.f(x_1);\ t_4;\ (x'_s) \\
\}\ ;\ [x_s \mapsto x'_s](\uparrow^{\mathrm{STL}} t) \\
\end{array}
$$

Figure 8 Transformation of an imperative program into a functional one based on state threads.

Limitations. Currently, RS_s does not include include references and in particular borrowing. The support for references and their translation into dataflow is interesting future work.

5 Compiling *ConDRust* algorithms to dataflow

In this section, we describe the main steps in *ConDRust* compilation from an imperative algorithm to a dataflow graph. The dataflow representation of the program is not the usual program dependence graph that is used in classic compilers such as LLVM for dataflow analyses. The dataflow graph that *ConDRust* targets is a runtime representation and parallel execution model of a program. Dataflow runtimes are the foundation for scalable database engines, data streaming for embedded systems and data analytics [38, 23, 27, 58]. This dataflow model is a perfect fit for such systems because it makes parallelism explicit in the graph. In this section, we focus primarily on the 3 forms: task-level, pipeline and data parallelism. We start with the translation of algorithms into dataflow and present the dataflow representation informally to define our transformations for data parallelism. In Section 6, we formally specify the semantics of the dataflow graph construction and execution as part of *ConDRust*'s code generation process.

5.1 From sequential-imperative to parallel-functional dataflow

ConDRust's programming model with its restrictions on variable usage enable the compiler to translate a RS_s program into a dataflow graph that exposes pipeline and task-level parallelism while preserving the program's semantics. This translation encompasses two steps that we

define in Figure 8. In the first step, *ConDRust* translates an algorithm in @_s into applicative normal (ANF) form. In $\downarrow_{\text{ST}}$, every call to a stateful function becomes a *state thread (ST)*: **let** $(x'_s, x) = x_s.f(x_1)$. The state x'_s is the updated state after the call. To make sure that succeeding stateful function calls on x_s operate on the new state x'_s, we substitute x_s with x'_s in t. In the second step, the compiler removes the global notion of imperative state and effectively transforms the program into a functional one. This transformation relies on the notion of state threads [35, 56, 25]. $\uparrow^{\text{STL}}$ turns every loop into a state thread, i.e., a *state-threading loop (STL)*. The resulting states of a loop are all states of the state threads inside the loop. We restrict the definition to a single state x'_s for brevity. $\downarrow_{\text{ST}}$ rewrites the term before recursing into the subterms. To handle nested loops, $\uparrow^{\text{STL}}$ recurses into the subterms first and rewrites the current term with the already rewritten subterms.

From this functional program representation, the *ConDRust* compiler translates stateless and stateful calls into nodes. Data dependencies become arcs that transfer data values in FIFO order. We denote the different types of nodes in a *ConDRust* dataflow graph as follows:

$$n \ ::= \boxed{f_{\text{SL}}} \mid \boxed{f_{\text{SF}}} \mid \boxed{\texttt{for}} \mid \boxed{\texttt{reuse}} \mid \boxed{\textbf{trfix}}$$

The first two node types execute calls to stateless and stateful functions respectively. In order to perform a call, a node needs to retrieve a data value from each of its incoming arcs and emits the result of the call to its outgoing arc before the next call is constructed. Stateful nodes additionally emit their updated state via a dedicated outgoing arc.

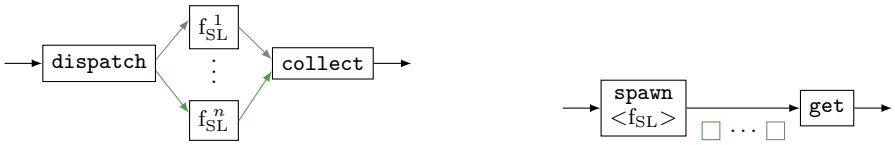
ConDRust translates loops and tail recursions directly into dedicated dataflow nodes. The `for` node streams the elements of the vector into its outgoing arc. The `trfix` node ties the knot of the recursion. Both language constructs, loops and tail recursion, open a new contextual scope, i.e., a subgraph. For tail recursion, this subgraph is closed such that the only way for data to enter and leave the graph is the `trfix` node. For loops, data enters the subgraph via `for` and `reuse` nodes and leaves it via a stateful function call node.

The dataflow construction is best explained on the dataflow graph for the labyrinth benchmark, shown in Figure 7. *ConDRust* generated this graph from the @_s specification on the right of Figure 4. Data that enters the loop subgraph via the `for` node *drives* the computation. State variables entering the loop body are `retries` and `grid`. The corresponding arcs are gated by `reuse` nodes that receive the data entering the loop. The `reuse` node attaches a reuse count n where n is the number of loop iterations, i.e., elements in the vector of pairs. Function call nodes with such a reuse count as input reuse the data values for n calls.

Task-level and pipeline parallelism

Task-parallelism automatically arises whenever nodes are (data) independent of each other, such as for example the `clone` node and the `Vec::new` node. The `for` node introduces pipeline parallelism between all data dependent nodes in the loop scope subgraph. As such the computation of the path for the third pair can already start while at the same time, the grid is updated with the found path for the second pair and the result of the first computed path is pushed onto `retries`, the result vector.

Task-level and pipeline parallelism fall out naturally from a dataflow representation of a program, but they are insufficient to compete with a threads/STM-based program. This is because exploiting data parallelism is key for scalability. Almost all programs in shared memory benchmark suites such as STAMP and PARSEC contain some data-parallel part [41, 8]. Apart from data parallelism, scalability in threads/STM implementations also depends on the retry overhead introduced by the STM in case of collisions.



(a) Static. **(b)** Dynamic.

Figure 9 Data parallelism inside a dataflow graph.

5.2 Dynamic data parallelism in a static dataflow

Data parallelism arises from an implicit (in-)dependence between the same stateless function call across loop iterations. As such, every stateless function call inside a loop is an opportunity for data parallelism. The `find_path` call in the labyrinth algorithm is an example of this. But introducing data parallelism into a static dataflow graph as shown in Figure 9a may lead to suboptimal performance. This is especially the case when the n nodes $f_{\text{SL}}^1 \dots f_{\text{SL}}^n$ feature different computation times for different input values. For example, `find_path` executes the same code but some pairs are more difficult to connect than others. As such the deterministic merge in the `collect` node stalls, waiting for straggling work [26]. This stalling does not occur in the threads/STM execution because STM does not enforce a deterministic order.

The performance problems of static work assignments are well known. This motivated Cilk's dynamic dataflow model (fork/join) and its work-stealing runtime scheduler [11]. To mitigate these problems without sacrificing determinism, we integrate dynamic dataflow into our static dataflow graph. In a dynamic dataflow graph, nodes are created at runtime. A node executes a task, i.e., a stateless function call, that gets *spawned(/forked)* on demand and executes once. Spawning a task creates a handle to its *future value*, i.e., the result of the stateless function call. This handle provides a `get` method to *join* the forked task with the spawning task by blocking until the call completed and the result is available. In Rust, the API for futures is equivalent to thread `spawn` and `join` as presented in the thread/STM version in Section 3. Tasks are processed by a pool of threads, as in Cilk. Whenever a thread is idling, it may steal tasks from other threads to reduce idle time. In the case of the labyrinth benchmark, a thread that already finished its path computation may steal queued path computations from a thread with a long-running path computation.

Determinism. The transformation in Figure 9b integrates dynamic dataflow to data-parallelize nodes with stateless function calls and uses the static dataflow to preserve the data value order, i.e, determinism and the semantics of the algorithm. Instead of replicating the stateless function call f_{SL} node, we lift it into a `spawn<`f_{SL}`>` node. For every received input, when normally a stateless function call would be executed, the `spawn<`f_{SL}`>` node submits this computation as a task to a work-stealing runtime system and emits the corresponding *future*. The downstream `get` node retrieves the value from the future. No reordering takes place because both `spawn<`f_{SL}`>` and `get` are stateless function call nodes in the static datalfow graph connected via a FIFO channel.

5.3 Amorphous data parallelism

The (data) parallelism in threads/STM-based programs is implicitly affected by two aspects: the available compute cores of the system and the operating system scheduler. Both impact the number of collisions. Our amorphous data parallelism transformation makes these implicit

effects explicit in the dataflow graph and exposes a knob to fine-tune runtime performance at compile-time. We first explain how the implicit effects influence the performance of threads/STM vs. *ConDRust* programs. Then we describe our transformation.

The implicit cap in threads/STM

In the *ConDRust*-based version of the labyrinth algorithm, the number of collisions per round is capped by n, the number of input `pairs`. In the worst case, $n - 1$ paths need to be recomputed. This is independent of whether the algorithm executes sequentially or is compiled into a dataflow graph for concurrent and parallel execution. Although the STM-based version spawns the same n threads/computations, it is unlikely for this implementation to hit this upper bound. For example, when we input 256 pairs then 256 threads race for updating the grid. In a system with 24 cores, the operating system scheduler will delay the execution of most of the threads. Putting concurrency aside for easier analysis, the first "round" would consist of 24 threads, of which at most 23 would collide. We refer to this bound as *collision limit*. The limit defines how many computations will see an outdated `grid`, i.e., shared data structure, and could thus lead to collisions. In the case of the *ConDRust* version, the algorithm defines the limit to be 256, i.e., all pairs, (for the first round) which translates into the worst case collision count of 255.

Setting a cap into irregular *ConDRust* algorithms

To compete with STM implementations, the *ConDRust* programming model makes the cap on the collision limit explicit in the algorithm. We do so by automatically transforming the algorithmic skeleton used across irregular applications to update state. As exemplified by the labyrinth benchmark, irregular applications (tail-)recurse over a worklist wl to evolve a complex data structure, i.e., a state s. In the benchmark, the worklist updates the grid.

The transformation is shown in Figure 10. To make sure it preserves the semantics, the worklist wl must be of type `Set`, i.e., the developer has to explicitly specify that there is no particular order for the elements of the worklist. The transformation distinguishes two different structures. In the *in-loop* version, the state s is updated inside the loop. The unoptimized *ConDRust* version of the labyrinth benchmark from Figure 4 is an example of this. In the *out-of-loop* version, the state update occurs at some point after the loop. This structure occurs in the `unarc` version of the *ConDRust* labyrinth implementation presented in Figure 5. In both cases, the `take_n`-node extracts the first N data items from the worklist wl and concatenates the *rest* with the recomputations *after* s was updated. Now, the compiler can optimize the parameter N, something that is not possible for threads/STM programs. N is an interesting target for further research in compiler optimization.

Determinism. Even though this transformation relies on the developer specifying that the worklist is a set, it preserves a deterministic execution. This holds as long as the set implements a deterministic iteration order when the same elements are inserted. For example, in Rust, several libraries exist that provide this property to hash set and hash map implementations.[6] Intuitively speaking, when the worklist is a set then the algorithm is independent of a particular iteration order. Our transformation essentially picks one of these orders at compile time. But when the generated program is executed then it will always be the same deterministic order that the worklist is being processed.

[6] `https://crates.io/crates/deterministic-hash`
`https://docs.rs/hash_hasher/latest/hash_hasher/`

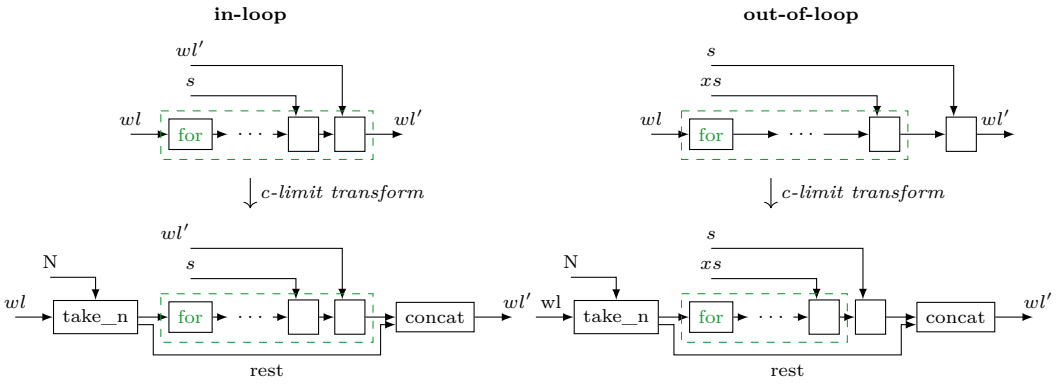


Figure 10 Transformation for amorphous data parallelism.

$$
\begin{array}{rrcl}
\text{Terms} & t & ::= & x \mid v \mid n \mid c;\ t \mid \mathrm{run}(t,t) \\
\text{Nodes} & n & ::= & \mathrm{n_{SL}}(\mathrm{f_{SL}},t,t) \mid \mathrm{n_{SF}}(\mathrm{f_{SF}},t,t,t,t) \mid \mathrm{for}(t,t,t) \mid \mathrm{reuse}(t,t,t) \mid \mathrm{trfix}(t,t,t,t) \\
\text{Channels} & c & ::= & \textbf{let}\ x = \texttt{chan}(t) \mid \textbf{let}\ (x,x) = \texttt{chan}() \\
\text{Values} & v & ::= & l \mid v_{\textcircled{R}}
\end{array}
$$

Figure 11 Syntactical constructs for terms and values of $\textcircled{R}_p$.

6 $\textcircled{R}_p$– A subset of Rust for parallel composition

With the transformations in place, we now formally specify the backend of the *ConDRust* compiler. We present the syntactic constructs for $\textcircled{R}_p$– a subset of Rust for **p**arallel composition, that the *ConDRust* compiler targets in Figure 11. $\textcircled{R}_p$ terms basically consist of two parts:

Graph construction An arc is a `chan`nel (c) in Rust's message-passing terminology and we define n, i.e., a term for each type of node in the dataflow graph.

Graph execution We abstract over an explicit implementation of a scheduler for a dataflow graph with a single `run` construct.

For the construction, we abstract over a concrete channel implementation. All we rely upon is the FIFO ordering property. Composition of nodes via arcs works solely via variable bindings. For example, the term in Figure 12 constructs a graph with a single (stateless) identity function ($\mathtt{id_{SL}}$) call node. For execution, we pass the receiving endpoint *result* and the list of nodes to `run` which executes the graph and reduces to the final result.

$$
\begin{array}{l}
\textbf{let}\ src : \texttt{Recv<i32>} = \texttt{chan}(5); \\
\textbf{let}\ (result : \texttt{Recv<i32>}, \\
\quad out : \texttt{Send<i32>}) = \texttt{chan}(); \\
\texttt{run}(result, (\mathrm{n_{SL}}(\mathtt{id_{SL}}, src, out) \sim: [\,]))
\end{array}
$$

Figure 12 A $\textcircled{R}_p$ program with a single identity node and the corresponding dataflow graph.

```
let (out, result) = std::sync::mpsc::channel();
let mut nodes = Vec::new();
nodes.push(Box::new(move || -> _ {
  let x = id(5);
  out.send(x)?;
  Ok(())
}));
run(nodes);
result.recv()?
```

Figure 13 The generated code for the graph of Figure 12.

We assume a type `Node` for nodes and align our specification for channels closely with `std::mpsc::channel` from Rust's standard library where `Receiver<`T`>` and `Sender<`T`>` represent the receiving and the sending endpoint of a channel, respectively. In our encoding, the types `Recv<`T`>` and `Send<`T`>` are reference types (in $T_{\circledR}$) for locations l in the store μ, i.e., channels are values in the store. The types T, the context Γ (without usage tracking), the store μ, the store typing Σ and the environment Δ follow the specification in $\circledR_s$. The appendix has the complete definition of the syntax, the operational semantics of $\circledR_p$ and a sketch of the proof for determinism. Informally, dataflow graphs in $\circledR_p$ are essentially Kahn Process Networks (KPN) [31]. KPNs execute deterministically because incoming arcs have blocking semantics[7] and the executed code of the node is scott-continuous. Our evaluation relation adheres to both of these properties.

7 Implementation

The current prototype of *ConDRust* comes with batteries included. No need for the developer to provide any specific implementations for channels, nodes or even a scheduler. *ConDRust* is currently implemented in 20K lines of Haskell code and takes advantage of existing language parsers with defined abstract syntax tree data structures for Rust [8]. Our implementation slightly diverges from the formal description of $\circledR_p$ in terms of the `reuse` nodes. The *ConDRust* compiler implementation contains additional transformations in the backend to fuse `reuse` nodes with their downstream neighbours into a single node. That way we do not have to define reuse and non-reuse versions of all the nodes. Otherwise, our backend generates code that closely aligns with the formalization in Section 6. Figure 13 presents the generated code for the sinlge-id-node graph of Figure 12. The *ConDRust* compiler does not create source channels but inlines values directly into the corresponding nodes. The generated Rust code uses Rust's channels from the standard library and creates a closure for each of the nodes in the dataflow graph. The code generator moves the channels into the closure of the node such as for example `out`, the sending endpoint of the channel for the final result. The `run` function just spawns a thread for each of the nodes and rejoins them, just as in the threads/STM code of Figure 2. For the dynamic dataflow part, the compiler generates code that uses the tokio runtime which provides a work-stealing scheduler.[9] *Our compiler generates safe Rust code and as such the Rust compiler verifies the absence of data races.*

Limitations. Our current implementation does not yet fully implement the type system that we formally specified in Section 4. In particular, we did not yet rigorously implement the guard for amorphous data parallelism transformation that checks whether the worklist is indeed a (hash) set. This is not due to a fundamental restriction. We believe that implementing this is straightforward and thus focused on more challenging aspects, such as implementing the transformations and code generation.

8 Evaluation

We evaluate *ConDRust* on benchmarks from 3 different benchmark suites. Our selected benchmarks cover a broad spectrum ranging from stateless to irregular algorithms. In our evaluation, we seek to answer the following questions:

[7] Blocking semantics prevent the construction of a non-deterministic merge node, the explicit notion of non-determinism in dataflow [2].
[8] `https://github.com/harpocrates/language-rust`
[9] `https://github.com/tokio-rs/tokio`

Name	Benchmark Suite	State	Algorithm Type	Data Parallelism	I/O
BlackScholes	PARSEC [8]	–	Regular	Dynamic	–
K-Means	STAMP [41]	Point-cluster assignment	Regular	Dynamic	–
Labyrinth	STAMP [41]	3D grid of coordinates	Irregular	Amorphous	–
Canneal	PARSEC [8]	Interconnected Mesh	Irregular	Amorphous	–
Intruder	STAMP [41]	Hash map	Regular	State local	–
Genome	STAMP [41]	Hash map	Regular	State local	–
Key-value store	YCSB [19]	Nested hash map	Regular	Dynamic/ State local	✓

Figure 14 Benchmark description.

1. Is the *ConDRust* programming model expressive enough for a broad range of applications?
2. What is the effort to translate the sequential benchmark code to *ConDRust* code? (Appendix Section A.2 has a first comparison of the effort to convert sequential programs into threads/STM programs and *ConDRust* programs.)
3. Can the *deterministic* code, that the *ConDRust* compiler generates, deliver performance that is on par with the *non-deterministic* threads/STM-based code?
4. What is the effect of the amorphous data parallelism transformation from Section 5.3?

Benchmarks. Figure 14 summarizes the benchmarks used in our evaluation. We selected 7 benchmarks from 3 different benchmark suites: STAMP [41], PARSEC [8], and YCSB [19]. STAMP is a benchmark suite intended to investigate the performance of software and hardware transactional memory implementations. Benchmarks in STAMP basically fall into two categories: (1) Algorithms where most of the execution time is spent inside transactions and (2) algorithms with very small transactions. Transaction size directly correlates to the amount of computation that depends on the global state structure. We selected several benchmarks from both categories. Labyrinth, Intruder and Genome fall into the first category, K-Means is located in the second. K-Means clustering spends only 7% of the execution time inside transactions. No other STAMP benchmark spends fewer cycles inside transactions. Labyrinth is one of the 3 benchmarks in STAMP that spend nearly 100% of their execution time inside lengthy transactions. It is also one of 2 STAMP benchmarks with an irregular algorithmic structure. The other benchmarks in STAMP have similar characteristics to the ones that we selected [41]. SSCA2 has characteristics similar to K-Means while Yada (Delaunay Mesh Refinement) and Bayes are similar to Labyrinth. Vacation is similar to Genome. Vacation simulates database transactions with STM which is not possible in real-world database systems where transactions involve network I/O between the client and the database server. For a more realistic setting with I/O, we chose YCSB, the state-of-the-art benchmark for key-value stores. We selected 2 more benchmarks, Canneal and BlackScholes, from PARSEC that both fall into the second category with small transactions. Canneal performs simulated annealing and is also irregular. Transactions in Canneal are short but the overall time spent inside transactions is about 70%. BlackScholes is the baseline for linear scalability. Overall our benchmark set consists of ca. 12k lines of Rust code.

Setup. Our experiments ran on an Intel Core i9-10900K CPU with 3.70 GHz, 32 GB RAM and 20 hardware threads, i.e., 10 cores and 10 hyperthreads. The operating system was Ubuntu version 20.04. We used the latest `rust-stm` version 0.4.0 and extended it to support deterministic transactions [49]. We executed each experiment 30 times and report the mean.

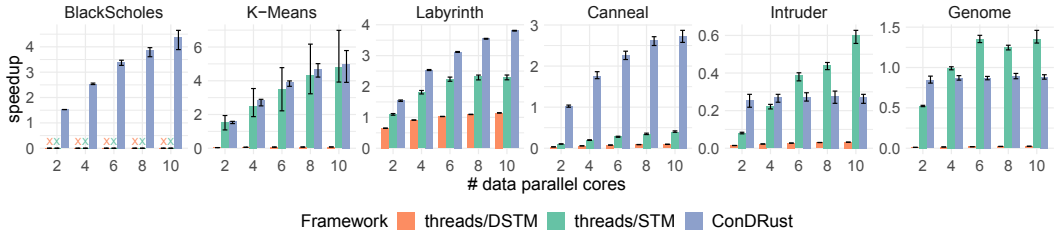


Figure 15 Speedup comparison of threads/DSTM, threads/STM and *ConDRust* across different benchmarks. Baseline is the sequential version.

Whenever possible, we used the data sets of the original benchmarks. Otherwise, we ported the data generation too. Appendix Section A has the input configurations.

Metrics. For our experiments, we ported the C/C++ reference implementations from STAMP and PARSEC to safe Rust. For each benchmark, we created 4 versions:

sequential a sequential baseline implementation,

threads/DSTM a concurrent version based on threads and DSTM, and

threads/STM a concurrent version based on threads and STM,

ConDRust a *ConDRust* version.

Benchmarks in STAMP, PARSEC and other benchmark suites (such as Lonestar for Galois programs [32]) for concurrent programming do not address the problem of finding the best granularity of work to place onto a thread. The benchmark code explicitly splits work into chunks and the size of these chunks is determined by the number of parallel threads. We follow this principle because *ConDRust* does not address the thread granularity problem either. For the STAMP and PARSEC benchmarks, we report the speedup over the sequential baseline. For the YCSB benchmark, we measure throughput.

Since we are particularly interested in the exploitation of data parallelism, we vary the *number of data parallel cores*. This is the natural metric for these applications and their respective threads/STM implementations. For the *ConDRust* versions, there are more threads because every dataflow node is assigned its own thread and we vary the number available threads for the dynamic part of the dataflow graph. Although *ConDRust* executes deterministically, we expect *ConDRust* programs to have performance that is on par with the threads/STM version if the collision-limit is tuned properly. For this reason, we explore different values of the limit and compare with the threads/STM performance. Auto-tuning approaches or heuristics can be used in the future to automatically tune the collision-limit.

8.1 Benchmark study

Figure 15 shows the overview of our benchmark study. For BlackScholes, K-means clustering, Labyrinth, and Canneal, the deterministic *ConDRust* programs outperform even the non-deterministic threads/STM counterparts. Benchmarks for genome sequencing and intrusion detection exploit data parallelism that *ConDRust* cannot yet exploit. In all of these benchmarks, DSTM synchronization delivers poor performance. In the following, we analyze the benchmark results from left to right and present further details. We add features one at a time. We start with dynamic data parallelism, then investigate irregular algorithms and then turn to *ConDRust*'s limitations. Afterwards, we case study a key-value store.

Dynamic data parallelism

The first benchmark is BlackScholes, a bulk data parallel workload. We present two versions. The first version is most intuitive. It creates vectors for the results on-demand and joins these vectors into a single result vector. That is, memory allocation is interspersed with the computation of stock options that does not require a shared data structure. This final result vector is shared state that requires synchronization. The second version pre-allocates all memory. This is the version that is most commonly used in the literature [16].

In the experiment of Figure 15, we used the intuitive benchmark version and naively opted for DSTM and STM to do the synchronization. The single-threaded *ConDRust* version completed the calculation of 40M stock options in 650 ms. The DSTM/STM versions timed out after 10 minutes. This shows that protecting a data structure blindly with STM is not really a practical solution. Efficient transactions have to be fine-grained and as such require a re-implementation of the data structure. That is why STAMP includes several dedicated STM-based data structure implementations.

To be fair, we also compared the *ConDRust* version against 3 more versions. The first version (T+lock) uses a lock instead of a transaction to protect the data structure. Figure 16 shows that this version does finish but features poor performance. Protecting large data structures with locks is no option either, and thus fine-grained locking is required. To explore this, we use an available implementation of a concurrent queue. But even this T+ConcQueue version does not scale well either. Finally, in Rust, threads are implemented as futures, i.e., when joined, they return a result of the computation on the thread. Figure 16 shows that only futures deliver performance comparable with the dataflows that *ConDRust* generates.

We also evaluated the second version of the benchmark that pre-allocates the memory. The *ConDRust* and futures versions both pre-allocate the individual result vectors for the parallel computations. To avoid the memory-allocations and copies to produce a single flat result vector, these version return a vector with the nested individual result vectors, i.e., a vector of vectors. We also created a futures (unsafe) version that mimics exactly the C version of PARSEC. That is, it pre-allocates the result vector and passes ranges to the individual computations where the results can be written to. In safe Rust, we did not find a way to tell the borrow checker that these ranges do not overlap and data races cannot occur. Hence, we had to introduce unsafe code. The results in Figure 17 show that the *ConDRust* version benefits from pre-allocating memory but is not yet on par with the futures versions. This is due to the fact that the computations are rather small such that additional runtime overhead of the concurrent *ConDRust* code becomes visible. We are certain that optimizations are possible that reduce this runtime overhead further.

K-means clustering is the first recursive algorithm with state. The K-means plot in Figure 15 shows that scalability of deterministic *ConDRust* programs is on par with the non-deterministic threads/STM counterparts.

Amorphous data parallelism

Labyrinth and Canneal are irregular applications. The plots in Figure 15 show that *ConDRust* generates programs that even outperform the threads/STM versions. This is because costly synchronization overhead is not present in the generated dataflow programs. In both benchmarks, we used the `unarc` optimization from Section 3 to avoid state cloning. *ConDRust* performed both transformations from Section 5, for dynamic data parallelism and amorphous data parallelism. To gain insights into the algorithm structure and the effects of these transformations, we perform further analyses.

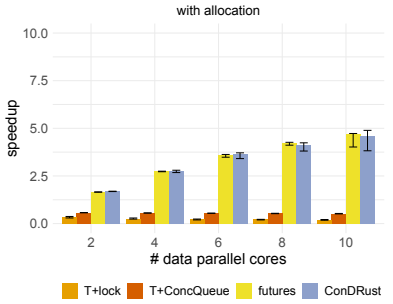


Figure 16 BlackScholes with memory allocation.

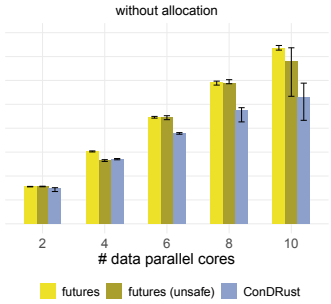


Figure 17 BlackScholes without memory allocation.

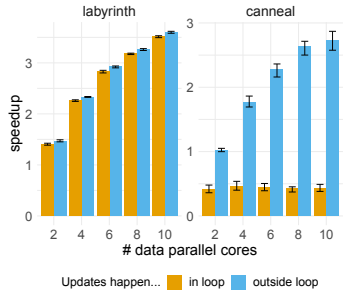


Figure 18 In-loop vs. out-of-loop state updates.

State update placement. We study the effect of placing the update to the state inside or outside the loop that iterates the worklist (Section 5.3), with results shown in Figure 18. The in-loop version has the update to the state inside the loop. Respectively, the generated dataflow combines data parallel computations with pipeline parallel state update. The outside-loop version performs the state update only after the loop when all computations have been computed. Pipeline parallelism does not arise but the algorithm can leverage the `unarc` optimization from Section 3 for a zero-clone version. The results show that pipeline parallelism does not really lead to speedups on either of the two benchmarks. In the case of labyrinth, both versions have the same performance and for canneal, the in-loop version does not scale at all. Pipeline parallelism only pays off when pipeline stages are balanced. This is not the case for both benchmarks where the first stage computes while the second stage only updates. The results also show that the performance of the Labyrinth benchmark is not sensitive to the update placement. This is mainly because of the low overhead incurred in cloning the state of the labyrinth. Canneal has a much larger state which explains the bad scalability of the in-loop update.

Collision-limit. To study the effect of the amorphous data parallelism transformation in *ConDRust*, we compiled both benchmarks once with and once without this transformation. Figure 19 shows that the amorphous data parallelism has no effect on the performance of Canneal but has a big impact on the performance of Labyrinth. The plots in Figure 20 vary the collision-limit (c-limit) as a multiple of the thread count such that work distributes evenly across the data parallel workers (i.e., threads). The right-most bars in the plots show the performance without amorphous data parallelism. In Canneal (re-)computation is cheap and the state large. Setting a low collision limit just prevents data parallelism to take full effect. In Labyrinth, finding a path is expensive and so are re-computations. As such, the collision-limit for optimal performance is only a small multiple of the thread count. With our optimization, the compiler can tune performance along these complexity coordinates: state size and (re-)computation complexity.

Threads/STM. For the Canneal benchmark, the *ConDRust* version in Figure 15 even greatly outperforms the threads/STM version. In fact, the threads/STM version for Canneal does not scale at all. This is due to the characteristics of the algorithm. The workload issues tens to hundreds of thousands of rather *short* transactions which makes the STM overheads a dominating factor. This is not the case in the Labyrinth benchmark where long-running transactions outweigh their overhead.

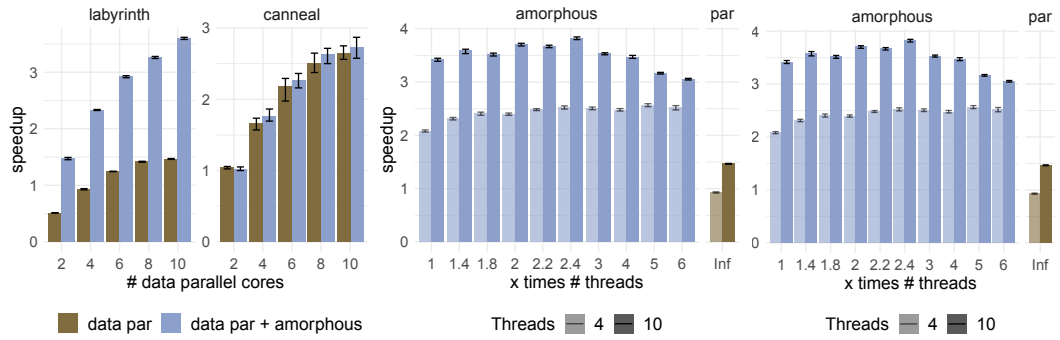


Figure 19 Amorphous.

Figure 20 Collision-limit effects.

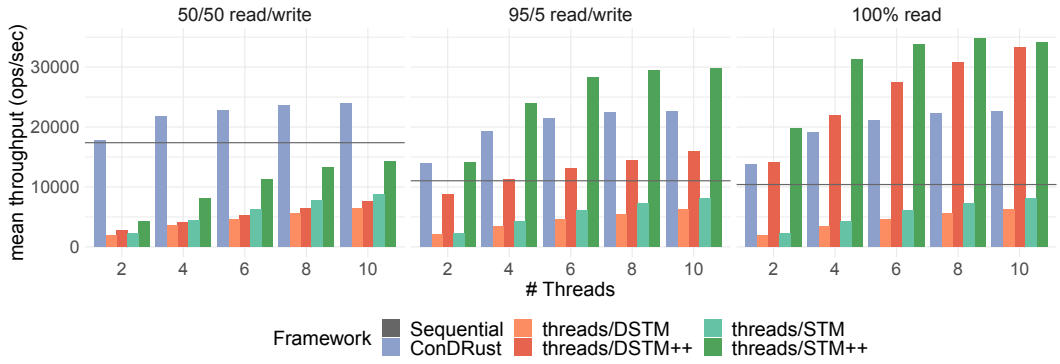


Figure 21 Throughput comparison for the key-value store implementations.

8.2 Beyond data parallelism

For Intruder and Genome, *ConDRust* fails to extract the data parallelism. Both benchmarks operate on a partitionable state structure. Intruder uses a hash map to reassemble network packets. Similarly, Genome uses a hash map to assemble and deduplicate genome sequences. The computation is stateful but operates only on a local part of the structure, for example, a bucket in the hash map. Deriving parallelism from extended knowledge about the state type and its structure is an interesting future research direction.

Conclusions. Overall, from the results in Figure 15 and our detailed analysis in Figures 16–20, we conclude that *ConDRust* generates dataflow programs that scale for certain application classes. The generated code executes deterministically while the sequential input programs remain verifiable. We managed to implement several benchmarks which establishes confidence that the programming model is expressive for a broad range of applications. Stateful functions with local effects on the state, as well as platforms with heterogeneous hardware [45] are an interesting future research direction.

8.3 Case study: Key-value store

For the YCSB benchmark, we populated the key-value store with 10,000 entries and configured a load of 30,000 operations, executed by 8 threads in parallel. We ran 3 configurations: (1) a write-heavy configuration with 50% reads and 50% writes, (2) a cloud-typical configuration

with mostly reads (95%) and (3) a read-only configuration. Figure 21 presents the throughput results for the key-value store implementations. In addition to the threads/(D)STM versions, we add another version called threads/(D)STM++. The threads/(D)STM++ version uses additional atomic-read instructions for read operations. We provide this implementation for fairness reasons because the threads/(D)STM version does not scale at all. This is due to the fact that the benchmark essentially just queries the data structure or updates it, with no substantial computation. Hence, the load creates a lot of accesses to the data structure, similar to Canneal. Even worse, STM clones the read value to provide an isolated private view on the read data. In the case of the key-value store, that data is essentially the entire underlying hash map. The atomic-read operations prevent this effect.

Even compared against the threads/(D)STM++ optimized version, the *ConDRust* generated code scales better in the write-heavy configuration and is almost on par with threads/STM++ in the cloud-typical configuration. Naturally, threads/(D)STM++ performs better when there are only reads. The *ConDRust* key-value store implementation cannot extract the parallelism from partitioning the hash map of the key-value store, similar to the Intruder and Genome benchmarks. The takeaway of this experiment is that the developer can write simple sequential code and the *ConDRust* compiler provides speedups that are on par with fine-tuned threads/STM implementations, while preserving determinism.

9 Related work

Various approaches exist to make concurrent programming deterministic but they either are not expressive enough or target functional rather than imperative programs. Language extensions such as the effect (type) system proposed by the Deterministic Parallel Java (DPJ) project primarily focus on proving the deterministic guarantees and providing these to the developer in the most non invasive way [13]. The DPJ authors conclude: *"[. . .] studying a wide range of realistic parallel algorithms has shown us that some significantly more powerful capabilities are needed for such algorithms. "* NESL is a functional language with the well-known higher-order functions `map` and `reduce` to parallelize stateless applications [9]. MapReduce is the programming model that has seen popularity for the very same reasons but has fallen from grace due to its limited expressivity, i.e., no states, no variables, no loops etc. None of the approaches derives scalable concurrency straight from imperative sequential programs that can be formally verified. The closest in spirit is MOLD, a tool that translates sequential imperative Java programs into MapReduce programs [46]. But MOLD does neither define a precise subset that it can translate nor reasons about verifiability or determinism of the compiled program.

Deterministic parallelism is a well-studied area but so far no approach could provide on-par performance with non-deterministic executions. To provide deterministic parallelism in MapReduce, the developer has to make sure that the function passed as an argument to `reduce` is associative and commutative. Commutativity also plays a key role in revisions, an extension to NESL's programming model to support shared state [10]. In this case, the developer has to provide a commutative function that is used at runtime to acquire a lock on a data value. Programming-wise, this shares similarities to programming lattice-based data structures [33]. Semantic-wise, the execution of revisions is the same as for software transactions which underpin most of the runtime approaches for deterministic parallelism in one form or the other. Notable examples include deterministic Galois [42], DeSTM [49] and LiTM [57]. Even CoreDet, the only fully compiler-based approach, uses the notion of a transaction and a barrier to synchronize threads and enforce a deterministic commit

order [7]. DMP, CoreDet's predecessor, fully relied on software transactions which did not scale because the transactions became too large [22]. CoreDet relied on hardware transactions, but implementations of these turned out to support only very small transactions and were even disabled again from Intel processors because their inherent complexity allowed various side-channel attacks [24, 18, 34]. *ConDRust* requires no additional commutativity properties nor specialized hardware. Nevertheless, we do acknowledge that performance could certainly benefit from properties such as associativity and commutativity.

We are not the first to recognize the benefits of dynamic task scheduling and the collision limit for irregular applications. However, we are the first to build atop a dataflow runtime and are not aware of a compiler with explicit transformations for these key performance concepts. Higher-level abstractions for data parallelism often build upon dynamic dataflow constructs such as Cilk's fork/join primitives [11]. Examples include Galois collections and the parallel loops in the style of NESL in the revisions programming framework. All other approaches, use threads to let the operating system schedule operations. Similar to our collision limit, the authors of revisions perform rounds of computations in batches to bound the number of computations per round [10]. LiTM implements revisions as simple transactions and the internal algorithm that executes these transactions is almost identical to the result of the in-loop state update transformation to limit the collisions per round in Figure 10 [57]. But LiTM again inherits all the overhead that is connected with an STM implementation such as maintaining read/write sets and lock tables. *ConDRust* does not incur such overheads because there are no data races in the generated dataflow programs and as such no synchronization is required.

10 Conclusion and Future Work

We presented *ConDRust*, a new programming model and compiler to translate verifiable sequential imperative Rust programs into scalable concurrent ones. The developer can use existing tools such as Kani to formally verify the sequential program. For scalable concurrency, the *ConDRust* compiler translates the sequential composition into a concurrent one based on threads and message-passing channels. Our compiler design fosters semantic-preserving transformations that preserve interesting properties such as determinism. In our evaluation, the *ConDRust* compiler generated code that even outperformed non-deterministic concurrent programs. Our compiler is aware of stateful calls and serializes them without costly synchronization. This benefit is big enough to outweigh the cost of enforcing a particular deterministic order even for stateful irregular applications that are notoriously hard to parallelize. Our results motivate the following interesting directions for future work:

Semantic preservation In this paper, we argued only informally that our compiler transformations preserve the semantics of the input program. Nevertheless, the described transformations can serve as the foundation for a formally-verified version of our compiler.

References $\circledR_s$, the subset for sequential imperative composition, presented in this paper, does not include references. The developer has to use runtime-checked reference implementations (`Arc`s) and according optimizations such as the `unarc` optimization from Section 3. Adding references to $\circledR_s$ is certainly an interesting future research direction.

Partitioned state A limitation of our programmming model so far is the missing notion of functions that operate on disjoint parts of a state structure. Performance for such algorithms is not on par with their concurrent counterparts. What is a sufficient encoding of partitioned state in $\circledR_s$?

References

1 Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space-efficient parallel functional programming. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434299.

2 Arvind, Kim P. Gostelow, and Wil Plouffe. Indeterminacy, monitors, and dataflow. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 159–169, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/800214.806559.

3 Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360573.

4 Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017. doi:10.1145/3015146.

5 Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, 1991.

6 Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, pages 283–303. Springer, 2010.

7 Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 53–64, 2010.

8 Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, volume 2011, page 37, 2009.

9 Guy E. Blelloch. *NESL: a nested data parallel language.* Carnegie Mellon Univ., 1992.

10 Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 181–192, 2012.

11 Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/209936.209958.

12 Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, page 4, USA, 2009. USENIX Association.

13 Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, 2009.

14 Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 8*, pages 395–409. Springer, 2005.

15 Cristiano Calcagno, Hongseok Yang, and Peter W. O'hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science: 21st Conference Bangalore, India, December 13–15, 2001 Proceedings 21*, pages 108–119. Springer, 2001.

16 Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4), December 2015. doi:10.1145/2829952.

17 Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–36, 2003.

18 Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, et al. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40, 2010.

19 Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

20 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

21 Enrico Armenio Deiana, Brian Suchy, Michael Wilkins, Brian Homerding, Tommy McMichen, Katarzyna Dunajewski, Peter Dinda, Nikos Hardavellas, and Simone Campanoni. Program state element characterization. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, pages 199–211, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3579990.3580011.

22 Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 85–96, 2009.

23 David J. DeWitt, Robert Gerber, Goetz Graefe, Michael Heytens, Krishna Kumar, and Murali Muralikrishna. Gamma-a high performance dataflow database machine. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1986.

24 Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14, 2014.

25 Sebastian Ertel, Justus Adam, Norman A Rink, Andrés Goens, and Jeronimo Castrillon. Stclang: State thread composition as a foundation for monadic dataflow parallelism. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pages 146–161, 2019.

26 Kim P. Gostelow and Wil Plouffe. Indeterminacy, monitors, and dataflow. In *Proceedings of the sixth ACM symposium on Operating systems principles*, pages 159–169, 1977.

27 Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. *ACM SIGMOD Record*, 19(2):102–111, 1990.

28 Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

29 Wesley M. Johnston, JR. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

30 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

31 Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

32 Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76. IEEE, 2009.

33 Lindsey Kuper and Ryan R. Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84, 2013.

34 Michael Larabel. Intel to disable tsx by default on more cpus with new microcode. `https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode`, 2021. [Online; accessed 02-March-2022].

35 John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM. `doi:10.1145/178243.178246`.

36 Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions on electronic computers*, pages 346–365, 1961.

37 E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. `doi:10.1109/MC.2006.180`.

38 Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

39 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. `doi:10.1145/1538788.1538814`.

40 Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, September 2020. `doi:10.14778/3407790.3407808`.

41 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

42 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: On-demand, portable and parameterless. *ACM SIGPLAN Notices*, 49(4):499–512, 2014.

43 Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

44 Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *ESOP*, volume 6602, pages 439–458. Springer, 2011.

45 Christian Pilato, Stanislav Bohm, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Vojtech Cima, Radim Cmar, Dionysios Diamantopoulos, Fabrizio Ferrandi, Jan Martinovic, et al. Everest: A design environment for extreme-scale big data analytics on heterogeneous platforms. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1320–1325. IEEE, 2021.

46 Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 909–927, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2660193.2660228`.

47 Mike Rainey, Ryan R. Newton, Kyle Hale, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut A. Acar. Task parallel assembly language for uncompromising parallelism. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1064–1079, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3453483.3460969`.

48 Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

49 Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. Destm: Harnessing determinism in stms for application development. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 213–224, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2628071.2628094`.

50 Federico Reghenzani, Giuseppe Massari, and William Fornaciari. Timing predictability in high-performance computing with probabilistic real-time. *IEEE Access*, 8:208566–208582, 2020.

51 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

52 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

53 Aaron Turon. *Understanding and expressing scalable concurrency*. PhD thesis, Northeastern University, 2013.

54 Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, pages 321–330, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510457.3513031.

55 Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. Citeseer, 1990.

56 Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

57 Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: A lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.

58 Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

A Evaluation

This section provides the details for our evaluation. We list configuration parameters and afterwards show code metrics to compare the threads/STM programs with the *ConDRust* programs.

A.1 Configurations

Benchmark	Arguments
BlackScholes	in_40M.txt
K-Means	-n 40 -t 0.00001 random-n65536-d32-c16.txt
Labyrinth	random-x512-y512-z7-n512.txt
Canneal	–swaps 15000 -t 2000 -m 128 400000.nets
Intruder	-a 10 -l 16 -n 4096 -s 1
Genome	-g 16384 -s 64 -n 16777216
YCSB	kv-store size = 10,000 records, operation count: 30,000

Figure 22 Benchmark parameters and inputs.

Figure 22 lists the configurations that we used in our experiments. Whenever possible, we used the data sets from the original benchmarks. When this was not possible, we ported the data generation too.

A.2 Programmability Comparison

Table 1 compares *ConDRust* and threads/STM in terms of the programming effort required to derive the respective implementation from a sequential one. Of course effort in itself is hard to measure, as different abstractions and frameworks require different thought processes

Table 1 Comparison of the programming effort required to adapt a sequential program to *ConDRust* (ConD_{rs}) and threads/STM.

Benchmark	State Modifications		Function Recompositions		Synchronization Primitives		Concurrency Code	
	ConD_{rs}	STM	ConD_{rs}	STM	ConD_{rs}	STM	ConD_{rs}	STM
K-Means	**0**	2	**7**	**7**	**0**	11	**0**	2
Labyrinth	**1**	**1**	10	**6**	**0**	4	**0**	2
Canneal	12	**3**	12	**8**	**0**	31	**0**	2
Intruder	**0**	1	**4**	**4**	**0**	12	**0**	2
Genome	**3**	8	11	**5**	**0**	15	**0**	4

when used. Therefore, the table compares how a number of key properties of the applications in question changed. State modifications denote changes to fields of the program state and are a direct result of adapting an application to another framework. In order to derive a *concurrent composition* and accomodate state modifications, functions must be changed. These changes are denoted as function recompositions. Furthermore, the derivation of a concurrent application requires in case of the threads/STM approach the introduction of concurrency and synchronization code. Of course, some modifications prompt further changes affecting categories. A state change may require adjusting multiple function signatures and bodies, while added concurrency requires synchronization. Hence, fewer modifications are always better, as they require less effort. Note that this comparison is potentially biased, as the original STAMP suite did not include sequential versions. We derived these manually from the parallel code, which may result in the sequential versions being easier to port to the threads/STM framework.

The first difference is that *ConDRust* programs are free of concurrency abstractions and synchronization primitives. This does not only enable verification but prevents the introduction of concurrency hazards such as data races or deadlocks. We observe that increased use of synchronization primitives results in more transaction conflicts and degraded performance. As synchronization in threads/STM works on the type level, the framework requires generally more state modifications, but fewer individual function recompositions. This means that while a smaller percentage of the code base is changed, the changes are more substantial. A single recomposition here may include incorporating synchronization or concurrency, such that functions or parts thereof can be run in a transaction. Since transactions may fail, failure models have to be considered while altering the code. Also, since transactions can not be nested, special care must be taken to avoid that. Finally, transaction size plays an important role in the overall performance and must therefore be carefully chosen.

ConDRust on the other hand for the most part only needs few state changes. These are mostly done to remove types that are not thread-safe and could hence not be used in a concurrent environment. In the case of Canneal, a large struct had to be used to replace a non type safe state sharing approach. The main work required to derive a *ConDRust* implementation indeed lies in the decomposition and recomposition of functions. In contrast to threads/STM this often only entails breaking up bigger functions into several smaller ones (which are counted individually) and removing references from function definitions. As a result, the code bases became more fine-grained and compartimented, with each function only handling a single task.

$$
\begin{array}{rrcl}
\text{Terms} & t & ::= & x \mid v \mid \texttt{|}x:T\texttt{|->}\ T\ \{\ t\ \} \mid t(t) \mid \textbf{let}\ x:T=t;t \mid \textbf{let mut}\ x:T=t;t \mid \\
 & & & \mathrm{f}_{\mathrm{SL}}(t_1) \mid t_s.\mathrm{f}_{\mathrm{SF}}(t_1) \mid \textbf{for}\ x\ \textbf{in}\ t\ \{\ t\ \} \mid \textbf{trfix}\ t\ t \\
\text{Evaluation context} & E & ::= & \square \mid E(t) \mid v(E) \mid \textbf{let}\ x:T=E;t \mid \textbf{let mut}\ x:T=E;t \mid \\
 & & & \mathrm{f}_{\mathrm{SL}}(E) \mid E.\mathrm{f}_{\mathrm{SF}}(t) \mid v_s.\mathrm{f}_{\mathrm{SF}}(E) \mid \\
 & & & \mid \textbf{for}\ x\ \textbf{in}\ E\ \{\ t\ \} \mid \textbf{for}\ x\ \textbf{in}\ v\ \{\ E\ \} \mid \textbf{trfix}\ E\ t \mid \textbf{trfix}\ v\ E \\
\text{Values} & v & ::= & l \mid v_{\textcircled{R}} \mid \texttt{|}x:T\texttt{|->}\ T\ \{\ t\ \} \\
\text{Types} & T & ::= & \texttt{Ref<}T\texttt{>} \mid T_{\textcircled{R}} \mid \textbf{mut}\ T_{\textcircled{R}} \\
\text{Typing context} & \Gamma & ::= & \varnothing \mid x:(n,T) \\
\text{Store} & \mu & ::= & \varnothing \mid \mu, l=v \\
\text{Store typing} & \Sigma & ::= & \varnothing \mid \Sigma, l:T \\
\text{Environment} & \Delta & ::= & \varnothing \mid \mathrm{f}_{\mathrm{SL}}:T\texttt{->}T \mid \mathrm{f}_{\mathrm{SF}}:\textbf{mut}\ T\texttt{->}T\texttt{->}T
\end{array}
\qquad \frac{t \mid \mu \longrightarrow t' \mid \mu'}{E[t] \mid \mu \longrightarrow E[t'] \mid \mu'}\ \text{E-CTXT}
$$

Figure 23 Syntactical constructs and evaluation context of $\textcircled{R}_s$.

Overall, we observe that *ConDRust* requires less severe changes to the code base. The changes that are required are merely the breaking up of functions to expose parallelism and the removal of state sharing.

B $\textcircled{R}_s$– A subset of Rust for sequential composition

In this section, we formally specify $\textcircled{R}_s$, the subset of the Rust language that encompasses the *ConDRust* programming model. We start with the synatx and the operational semantics with focus on the integration of stateless and stateful function calls. Afterwards, we specify the type system that guards the usage of state. A clear specification of state is important for the compiler to reason about the various forms of parallelism in the derived dataflow representation of the program.

B.1 Syntax

ConDRust supports the subset of Rust's syntax that is necessary to compose calls to stateless and stateful functions (also called methods). We define this subset in Figure 23 as $\textcircled{R}_s$– a subset of Rust for **s**equential composition. The semantics of $\textcircled{R}_s$ are the same as for Rust. For this paper, we restrict the terms of the language to variables x, abstractions (closures in Rust) $\texttt{|}x:T\texttt{|->}\ T\ \{\ t\ \}$, algorithm application $t(t)$, immutable and mutable bindings, **for**-loops and tail-recursion (**trfix**). We restrict the presentation of $\textcircled{R}_s$ in the following (common) ways:

1. Abstractions and calls may only have a single parameter. The extension to support multiple parameter is straightforward.
2. We desugar top-level algorithm definitions into `let`-bound closures such that a top-level defined function can be used in multiple locations of succeeding function definitions.

The evaluation context E specifies that terms evaluate from left to right in a call-by-value fashion. $\textcircled{R}_s$ and Rust are imperative languages such that require a store μ to model the state of the program. Store locations l are part of the syntactical constructs. The small-step operational semantics $t \mid \mu \longrightarrow t' \mid \mu'$ relates a term t and a store μ to a term t' and a store μ'. The store μ maps labels to values where $\mu, l \mapsto v$ denotes the usual conjuction of store mappings μ and the mapping from label l to value v. Values are store locations, (tail recursion) abstractions and the values defined in the Rust language itself. In the specification of $\textcircled{R}_s$'s operational semantics, we assume general types and values for booleans, tuples and lists with constructors [ ] for the empty list and $v \sim\!: vs$ (cons) where v is the head with the tail vs. In Rust, the corresponding data structure to a list is a vector (`Vec`).

B.2 Operational Semantics

The usual way for values to enter the evaluation is via the key ingredient in *ConDRust*'s programming model: *stateless functon calls* $\mathrm{f}_{\mathrm{SL}}(t_1)$ and *stateful function calls* $t_s.\mathrm{f}_{\mathrm{SF}}(t_1)$. [10] The definition of stateless and stateful functions themselves are not part of R_s. We define Δ as an environment in the typing relation that holds the typing information for the stateless and stateful functions used in the term to be evaluated. As such, the operational semantics for calls to stateless and stateful functions rely upon the evaluation relation of Rust ($\Downarrow_{\text{R}}$):

$$\boxed{t \mid \mu \longrightarrow t' \mid \mu'}$$

$$\frac{\mathtt{f}_{\mathrm{SL}}(v) \mid \varnothing \Downarrow_{\text{R}} v_r \mid \varnothing}{\mathtt{f}_{\mathrm{SL}}(v) \mid \mu \longrightarrow v_r \mid \mu} \text{ E-FSL} \qquad \frac{l_s.\mathtt{f}_{\mathrm{SF}}(v_1) \mid l_s \mapsto v_s \Downarrow_{\text{R}} v_r \mid l_s \mapsto v_s'}{l_s.\mathtt{f}_{\mathrm{SF}}(v_1) \mid \mu, l_s \mapsto v_s \longrightarrow v_r \mid \mu, l_s \mapsto v_s'} \text{ E-FSF}$$

Inside these functions, developers re-gain the full feature set of Rust. Stateless calls do not have side-effects. Side-effects for stateful calls are restricted to a particular state location l_s in the store μ. These are the only rules that leverage Rust's evaluation relation ($\Downarrow_{\text{R}}$). In fact, calls are the only places where computation takes place while the rest of the language is for composition. Our operational semantics are based on the standard beta-reduction such that $[x \mapsto t_1]t_2$ with $x \in FV(t_2)$ replaces all occurences of the free variable x in t_2 with t_1. The usual rule then covers application of simple abstractions:

$$\texttt{|}x : T\texttt{|->}\ T\ \{\ t_2\ \}(v_1)\ \mid\ \mu\ \longrightarrow\ [x \mapsto v_1]t_2\ \mid\ \mu\quad \text{(E-AbsApp)}$$

Stateless bindings solely rely on beta-reduction. Mutable bindings register values in the store.

$$\begin{aligned} \textbf{let}\ x : T = v_1;\ t_2\ \mid\ \mu\ &\longrightarrow\ [x \mapsto v_1]t_2\ \mid\ \mu && \text{(E-Let)} \\ \textbf{let mut}\ x : T = v_1;\ t_2\ \mid\ \mu\ &\longrightarrow\ [x \mapsto l_1]t_2\ \mid\ \mu, l_1 = v_1 \quad \text{where } l_1 \notin dom(\mu) && \text{(E-LetMut)} \end{aligned}$$

As such, the only values in the store refer to mutable state references. We further restrict control flow to loops and tail recursion leaving out other forms such as conditionals that play only a minor role in the parallel execution of a program. Loops iterate over a list of values.

$$\begin{aligned} \textbf{for}\ x\ \textbf{in}\ [\,]\ \{\ t_2\ \} \mid \mu\ &\longrightarrow\ () \mid \mu && \text{(E-LoopDone)} \\ \textbf{for}\ x\ \textbf{in}\ v \sim\!: vs\ \{\ t_2\ \} \mid \mu\ &\longrightarrow\ \textbf{let}\ x_1 = [x \mapsto v_1]t_2;\ \textbf{for}\ x\ \textbf{in}\ vs\ \{\ t_2\ \} \mid \mu \quad \text{where } x_1 \notin FV(t_2) && \text{(E-LoopStep)} \end{aligned}$$

In Section 5 and in our implementation, loops may in fact iterate over all data types that implement Rust's `Iterator` trait which for instance includes `HashSet`. This allows the developer to specify that the loop does not depend on a particular order and enables our second transformation that extracts amorphous data parallelism. To model this in R_s, we assume a stateless function that uses the iterator to collect the items into a list before looping over them. In fact, `collect` is a standard function of Rust's `Iterator`. We allow loops with an unknown iteration count via tail recursion. Tail recursion is a derived form:

[10] We allow *ConDRust* algorithms to be called from anywhere in a Rust program. Such a call may have arguments. The assumptions would be stated in Γ and require another context to acces them during evaluation. We omit this detail at this point in favor of a concise presentation.

$$\boxed{\Delta, \Gamma \mid \Sigma \vdash t : T}$$

$$\frac{\Sigma(l) = T}{\Gamma \mid \Sigma \vdash l : \texttt{Ref<}T\texttt{>}}\ \text{T-Loc} \qquad \frac{}{\Delta, x : (1, T) \mid \Sigma \vdash x : T}\ \text{T-Var}$$

$$\frac{\begin{array}{c}\Delta, \Gamma_1 \mid \Sigma \vdash t_1 : T_1 \\ \Delta, \Gamma_2 \mid \Sigma \vdash t_2 : T_1 \texttt{ -> } T_2\end{array}}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma \vdash t_2(t_1) : T_2}\ \text{T-AppAbs} \qquad \frac{x \notin \Gamma \qquad \Gamma, x : (n, T_1) \mid \Sigma \vdash t : T_2}{\Delta, \Gamma \mid \Sigma \vdash \mid x : T_1 \mid \texttt{ -> } T_2 \ \{\ t\ \} : T_1 \texttt{ -> } T_2}\ \text{T-Abs}$$

$$\frac{\Delta, \Gamma_1 \mid \Sigma_1 \vdash t_1 : T_1 \qquad \Delta, \Gamma_2, x : (1, T_1) \mid \Sigma_2 \vdash t_2 : T_2}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma_1, \Sigma_2 \vdash \textbf{let}\ x : T_1 = t_1;\ t_2 : T_2}\ \text{T-Let}$$

$$\frac{\Delta, \Gamma_1 \mid \Sigma_1 \vdash t_1 : T_1 \qquad \Delta, \Gamma_2, x : (n, \texttt{Ref<}\textbf{mut}\ T_1\texttt{>}) \mid \Sigma_1 \vdash t_2 : T_2}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma_1, \Sigma_2 \vdash \textbf{let mut}\ x : T_1 = t_1;\ t_2 : T_2}\ \text{T-LetMut}$$

$$\frac{\begin{array}{c}\Delta, \Gamma \mid \Sigma \vdash t_1 : T_1 \\ \Delta(\texttt{f}_{SL}) = T_1 \texttt{ -> } T_2\end{array}}{\Delta, \Gamma \mid \Sigma \vdash \texttt{f}_{SL}(t_1) : T_2}\ \text{T-FSL} \qquad \frac{\begin{array}{c}\Delta, \Gamma_1 \mid \Sigma_1 \vdash t_1 : \texttt{Ref<}\textbf{mut}\ T_1\texttt{>} \qquad \Delta, \Gamma_2 \mid \Sigma_2 \vdash t_2 : T_2 \\ \Delta(\texttt{f}_{SF}) = \textbf{mut}\ T_1 \texttt{ -> } T_2 \texttt{ -> } \bigcirc\, T_3\end{array}}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma_1, \Sigma_2 \vdash t_1.\texttt{f}_{SF}(t_2) : T_3}\ \text{T-FSF}$$

$$\frac{\Delta, \Gamma_2 \mid \Sigma \vdash t_2 : \texttt{Vec<}T_1\texttt{>} \qquad \Delta, \Gamma_3, s : (1, \texttt{Ref<}\textbf{mut}\ T_s\texttt{>}), x_1 : (n, T_1) \mid \Sigma \vdash t_3 : ()}{\Delta, \Gamma_2 \oplus \Gamma_3, s : (1, \texttt{Ref<}\textbf{mut}\ T_s\texttt{>}) \mid \Sigma \vdash \textbf{for}\ x_1\ \textbf{in}\ t_2\ \{\ t_3\ \} : ()}\ \text{T-Loop}$$

where $\forall s \in FV(t) \wedge s \neq x_1$

$$\frac{\Delta, \varnothing \mid \Sigma \vdash t_1 : (\texttt{bool}, T_2, T_1) \qquad \Delta, \varnothing \mid \Sigma \vdash t_2 : T_1 \texttt{ -> } (\texttt{bool}, T_2, T_1)}{\Delta, \varnothing \mid \Sigma \vdash \textbf{trfix}\ t_1\ t_2 : T_2}\ \text{T-Fix}$$

Figure 24 *ConDRust*'s type system tracks and restricts variable usage.

$$\begin{array}{l}\textbf{let}\ f = \mid x : T_1 \mid \texttt{ -> } T_2\ \{ \\ \quad \textbf{let}\ (x_1, x_2, x_3) = t_b; \\ \quad \textbf{if}\ x_1\ \{\ x_2\ \} \\ \quad \textbf{else}\ \{\ f(x_3)\ \} \\ \};\end{array} \quad \stackrel{\text{def}}{=} \quad \begin{array}{l}\textbf{let}\ f = \mid x_0 : T_1 \mid \texttt{ -> } T_2\ \{ \\ \quad \textbf{let}\ f' = \mid x : T_1 \mid \texttt{ -> } (\texttt{bool}, T_2, T_1)\ \{\ t_b\ \}; \\ \quad \textbf{trfix}\ f'(x_0)\ f' \\ \};\end{array}$$

Desugaring captures t_b, the computation of the recursion, as f'. Our tail-recursive combinator **trfix** takes two arguments. Argument 1 is an application f' to x_0, the initial parameter of a recusive call. This application reduces to a triple (x_1, x_2, x_3) with the boolean descriminator x_1, the final term x_2 and x_3, the argument to the tail-recursive call. Argument 2 is f' for recursion. We encode the conditional that guards this tail recursive call into **trfix**'s semantics:

$$\begin{array}{rcll}\textbf{trfix}\ (\texttt{true}, v_2, v_3)\ v_4 \mid \mu & \longrightarrow & v_2 \mid \mu & \text{(E-FixDone)} \\ \textbf{trfix}\ (\texttt{false}, v_2, v_3)\ v_4 \mid \mu & \longrightarrow & \textbf{trfix}\ v_4(v_3)\ v_4 \mid \mu & \text{(E-FixRecur)}\end{array}$$

In case the descriminee is `true`, we return the final result v_2. Otherwise, we apply the recursive argument v_3 to the abstraction that is always the second argument of **trfix**. Again, we restrict the presentation to a single recursive argument and argue that the extension to multiple arguments is straightforward. In the context of this paper, we are particularly interested in the case where the arguments to the recursion are a state to be updated and a worklist that triggers these updates.

B.3 Type system

The *ConDRust* programming model carefully distinguishes between stateless and stateful computations. This enables the compiler to perform the translation of an algorithm into a dataflow representation and extract data parallelism while preserving the algorithm semantics.

The type system enforces the following programming discipline:

1. A variable may either be used as state or as input to a function call.
2. A variable that is input to a function call may only be used once.
3. A state variable may be used more than once except for a loop term where it may only be used once.

These somewhat restrictive rules are key enablers of our approach.

Data often needs to be shared across the concurrent parts of the program. A good example of this is the grid in the STM implementation that required a deep clone to make the concurrent execution scale. Often, introducing parallelism into a program represents a trade-off between speedup and memory efficiency. To make sophisticated decisions that strike a good balance for this trade-off, data structure knowledge is required by the compiler. We leave this to future work and enable the developer to explicitly make that decision by `clone`ing or sharing cloned `Arc`s. These techniques are already common practice for sharing data in Rust.

In *ConDRust*'s type system, presented in Figure 24, we use concepts from linear types to track and restrict variable usage [55]. The typing context Γ (defined in Figure 23) captures not only the types of variables but also their usage count (see rule T-VAR). The types T_{R} are the types of the Rust programming language. With the rules T-LET and T-LETMUT, we distinguish between variables that are input to functions and state. In T-LET, *input variables* have a non-referential type and require a usage count of 1, i.e., they can only be used exactly once.[11] In T-LETMUT, *state variables* reference locations in the store μ and are marked with Rust's annotation for `mut`able types. Rules T-FSL and T-FSF specify the use of input and state variables in the type of the stateless and stateful function. Values in input position t_1 are of type T_1 while the state position t_s is required to be of type mutable reference `Ref<`T_s`>`. We define the state encapsulation property of a stateful function on the output type as $\bigcirc T_3$. A type T has this property if it does not contain borrowed references.

The type information of the values behind the references is captured in the store typing Σ. As such, a typing judgement $\Delta, \Gamma \mid \Sigma \vdash t : T$ reads as follows:

▶ **Definition 4** (Well-typed). *Given an environment Δ and a (local variable) context Γ with type assumptions on store locations Σ; a term t is well-typed if there exists a type T such that $\Delta, \Gamma \mid \Sigma \vdash t : T$.*

The rule T-LOOP restricts state variables to a single usage in the loop term t_3. Only due to this restriction, the *ConDRust* compiler can derive pipeline parallelism. Rule T-FIX prevents tail recursive functions from accessing contextual variables at all by requiring $\Gamma = \varnothing$. Accessing captured variables in a recursive closure is also uncommon in Rust because the closure needs to explicitly communicate the lifetime of such variables. That is particularly challenging, for a closure that performs an unknown number of iterations.[12] In the spirit of

[11] We treat unused variables as an undesirable property of a program that would benefit from a similar error message. Rust actually has similar warnings/errors for unused variables.

[12] Recursive closures need to be captured in structs to explicitly communicate lifetime information for the captured variables to the borrow checker. For more details see: `https://stevedonovan.github.io/rustifications/2018/08/18/rust-closures-are-hard.html`

$$
\begin{array}{rcll}
\Gamma & \oplus & \varnothing & = \varnothing \\
\varnothing & \oplus & \Gamma & = \varnothing \\
\Gamma_1, x : (n : T) & \oplus & \Gamma_2, x : (m : T) & = \Gamma_1 \oplus \Gamma_2, x : (n + m, T)
\end{array}
$$

Figure 25 Conjunction of typing contexts.

$$
\begin{array}{rrcl}
\text{Terms} & t & ::= & x \mid v \mid n \mid c;\ t \mid \text{run}(t, t) \\
\text{Nodes} & n & ::= & \text{n}_{\text{SL}}(\text{f}_{\text{SL}}, t, t) \mid \text{n}_{\text{SF}}(\text{f}_{\text{SF}}, t, t, t, t) \mid \text{for}(t, t, t) \mid \text{reuse}(t, t, t) \mid \text{trfix}(t, t, t, t) \\
\text{Channels} & c & ::= & \textbf{let}\ x = \texttt{chan}(t) \mid \textbf{let}\ (x, x) = \texttt{chan}() \\
\text{Values} & v & ::= & l \mid v_{\circledR} \\
\text{Types} & T & ::= & \texttt{Ref<}T\texttt{>} \mid T_{\circledR} \mid \textbf{mut}\ T_{\circledR} \\
\text{Typing context} & \Gamma & ::= & \varnothing \mid x : T \\
\text{Store} & \mu & ::= & \varnothing \mid \mu, l = v \\
\text{Store typing} & \Sigma & ::= & \varnothing \mid \Sigma, l : T \\
\text{Environment} & \Delta & ::= & \varnothing \mid \text{f}_{\text{SL}} : T\texttt{->}T \mid \text{f}_{\text{SF}} : \textbf{mut}\ T\texttt{->}T\texttt{->}T
\end{array}
$$

Figure 26 Syntactical constructs of $\circledR_p$.

linear types, we merge store typings via logical conjunction and define the conjunction for contexts in Figure 25. Based on this Rust subset, the *ConDRust* compiler can translate a sequential algorithm into a dataflow graph that makes all inherent parallelism explicit.

C $\circledR_p$– A subset of Rust for parallel composition

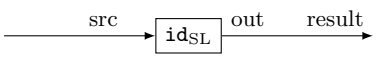
With the transformations in place, we now formally specify the backend of the *ConDRust* compiler to show that the generated code executes deterministically. We present the syntactic constructs for $\circledR_p$– a subset of Rust for **p**arallel composition, that the *ConDRust* compiler targets in Figure 26. Terms in this subset basically consist of two parts:

Graph construction An arc is a `channel` (c) in Rust's message-passing terminology and we define n, i.e., a term for each type of node in the dataflow graph.

Graph execution We abstract over an explicit implementation of a scheduler for a dataflow graph with a single `run` construct.

We abstract over a concrete channel implementation. All we rely upon is the FIFO ordering property which we specify via the usual list constructors: [ ] empty list, $v :\sim v'$ (cons) where v is the head with the tail v' and the dual $v' :\sim v$ (snoc) where v is the last element in the list and v' the list of the preceding elements. Additionally, we assume the presence of tuples in the Rust values $v_{\circledR}$ and types $T_{\circledR}$. Composition of nodes via arcs works solely via variable bindings. For example, the following term constructs a graph with a single (stateless) identity function ($\texttt{id}_{\text{SL}}$) call node:

$$
\begin{array}{l}
\textbf{let}\ src : \texttt{Recv<i32>} = \texttt{chan}(5); \\
\textbf{let}\ (result : \texttt{Recv<i32>}, out : \texttt{Send<i32>}) = \texttt{chan}(); \\
\texttt{run}(result, (\text{n}_{\text{SL}}(\texttt{id}_{\text{SL}}, src, out) :\sim [\,]))
\end{array}
$$



For execution, we pass the receiving endpoint *result* and the list of nodes to `run` which executes the graph and reduces to the final result. We assume a type `Node` for nodes and align our specification for channels closely with `std::mpsc::channel` from Rust's standard library where `Receiver<`T`>` and `Sender<`T`>` represent the receiving and the sending endpoint

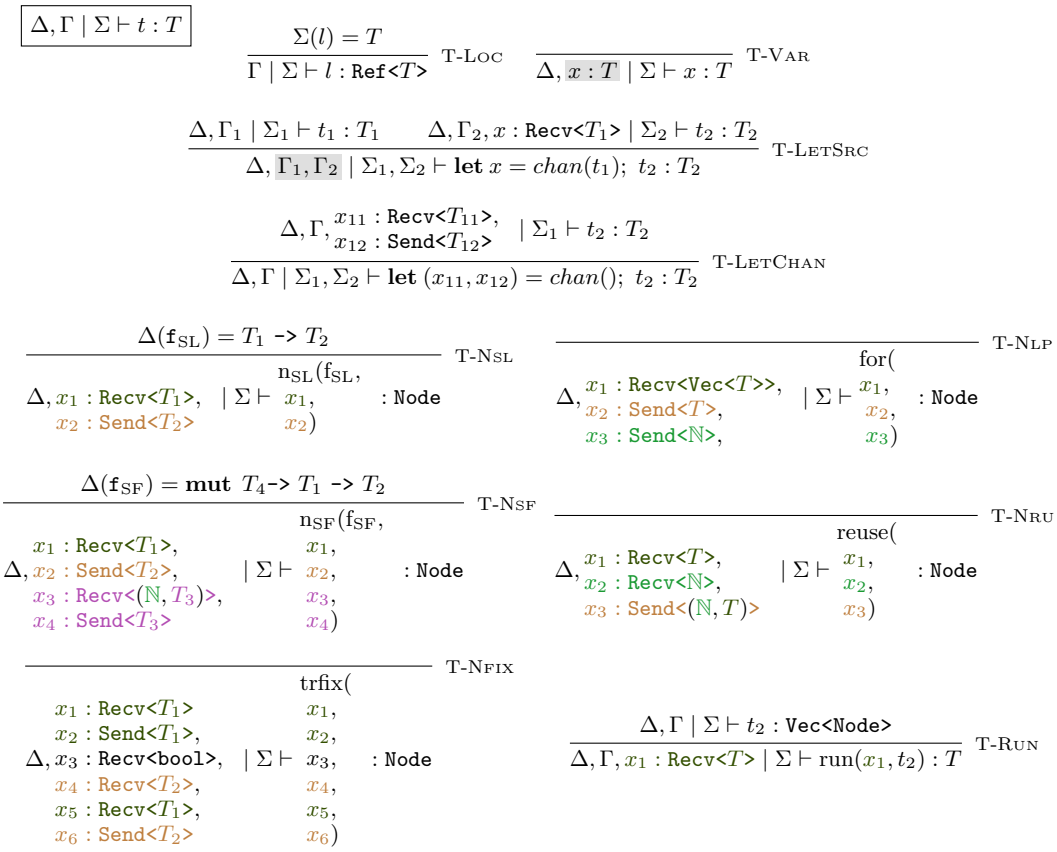


Figure 27 The linear type system of $\circledR_p$ for the construction and execution of the dataflow graph.

of a channel, respectively. In our encoding, the types `Recv<T>` and `Send<T>` are reference types (in $T_{\circledR}$) for locations l in the store μ, i.e., channels are values in the store. In fact, channels and their respective elements are the only values in the store. This simplication is possible because in *ConDRust* every state arrives along an arc, i.e., channel, where we preserve it across loop iterations. The context Γ (without usage tracking), the store μ, the store typing Σ and the environment Δ follow the specification in $\circledR_s$.

In the following, we first define the specifics of graph construction via the typing rules and afterwards present the operational semantics for graph execution to finally present our proof (sketch) for determinism.

C.1 Linear dataflow construction

Figure 27 defines the type system of $\circledR_p$. A dataflow graph consists of channels and nodes. In a dataflow graph each arc has exactly one sending node and one receiving node. To encode this invariant, we again resort to a linear type system approach and highlight linear aspects accordingly. The rules T-Loc and T-Var type locations and variables. Channel construction is typed in rules T-LetSrc and T-LetChan. Source channels (`chan(t)`) bind only a sending endpoint to pipe parameters from the surrounding Rust program into the dataflow graph. All other channels (`chan()`) bind a receiving and a sending endpoint. In both cases, the usual conjunction of typing contexts (Γ_1, Γ_2) asures that each endpoint is

used exactly once. The rest of the rules (T-NSL, T-NSF, T-NLP, T-NRU, T-NFIX) concern the construction of nodes and the execution of the graph (T-RUN). We increase readability of the inference rules in the typing and evaluation relation in two ways. First, to make the flow of the different types of data more pbvious, we highlight receiving and sending endpoints, value reuse and state. Second, to better align the type assumptions, we deliberately present the terms for nodes and `run` with variables instead of subterms, i.e., $n_{\text{SL}}(x_1, x_2)$ instead of $n_{\text{SL}}(t_1, t_2)$. This is not a restriction because we defined that channel endpoints have to be bound such that (node and `run`) terms requiring endpoint types can only be variables.

To keep the formal specification concise, only the T-NSF rules enables the reuse of data – in this case state. In the full formal specification and in our implementation, there are at least two versions for all nodes: one where the received input is of type T and another where it has an attached resuse count $(\mathbb{N}, T)$. Without loss of generality, we shows this only for the state input of the stateful function call node n_{SF}.

C.2 Operational semantics

In the small-step operational semantics, we use the store μ to define the relation of nodes that can be executed. Evaluation is again from left to right as in $\circledR_s$ following the E-CTXT rule from Figure 23. The construction of channels allocates dedicated locations in the store and the types `Send` and `Recv` are effectively references to a store location l.

$$\boxed{t \mid \mu \longrightarrow t' \mid \mu'}$$

$$\textbf{let } x_1 : \texttt{Recv<\_>} = \text{chan } v; t \mid \mu \longrightarrow [x_1 \mapsto l]t \mid \mu, l \mapsto (v \sim: [\,]) \quad \text{(E-LETSRC)} \quad \text{where } l \notin dom(\mu)$$

$$\textbf{let } (x_1 : \texttt{Recv<\_>}, x_2 : \texttt{Send<\_>}) = \text{chan}; t \mid \mu \longrightarrow [x_1 \mapsto l, x_2 \mapsto l]t \mid \mu, l \mapsto [\,] \quad \text{(E-LETCHAN)} \quad \text{where } l \notin dom(\mu)$$

For a source channel, the corresponding value is stored directly into the list and only the receiving end is emitted. Hence, the channel emits exactly one data value and remains empty for the rest of the computation. The receiving end x_1 and the sending end x_2 for all other channels, point to the same location l in the store. A channel is initially an empty list. For both types of channels, the evaluation makes a step using beta reduction.

The interesting part is the execution of the dataflow graph. Computation is complete when there is a value in the channel of the final endpoint.

$$\frac{}{\text{run}(v, l) \mid \mu, l \mapsto (v \sim: [\,]) \longrightarrow v \mid \mu} \text{ E-DONE} \qquad \frac{\exists v_n \in v_g.\ v_n \mid \mu \longrightarrow v_n \mid \mu'}{\text{run}(v_g, l) \mid \mu, l \mapsto [\,] \longrightarrow \text{run}(v_g, l) \mid \mu'} \text{ E-RUN}$$

Otherwise, there must a node v_n in the list of nodes v_g that can take a step that updates the store μ. This property holds because our linear typed construction does not allow for dangling channels and the following rules for the nodes in the graph always send data in every step. A stateless call node can make a step if its incoming channel referenced by label l_1 has data available:

$$\frac{\mathtt{f}_{SL}(v_1) \mid \varnothing \Downarrow_{\circledR} v_1' \mid \varnothing}{\mathtt{n}_{SL}(\mathtt{f}_{SL}, l_1, l_2 \mid \mu, \begin{smallmatrix} l_1 \mapsto (v_1 \sim: v_{12}), \\ l_2 \mapsto v_2 \end{smallmatrix} \longrightarrow \mathtt{n}_{SL}(\mathtt{f}_{SL}, l_1, l_2 \mid \mu, \begin{smallmatrix} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 :\sim v_1') \end{smallmatrix}} \text{ E-NSL}$$

The node retrieves the head v_i of the channel's list, performs the call and appends the resulting data value v_1' to the list of the outgoing channel (l_2). The execution of the stateless function itself is the same as defined in the operational semantics for stateless calls (E-FSL) in R_s defind in Section 4. Loop nodes follow the same execution pattern:

$$\frac{v_1.\texttt{size}() \mid \varnothing \Downarrow_{\text{R}} n \mid \varnothing}{\texttt{for}(l_1,l_2,l_3) \mid \mu, \begin{array}{l} l_1 \mapsto (v_1 \sim\!: v_{12}), \\ l_2 \mapsto v_2, \\ l_3 \mapsto v_3 \end{array} \longrightarrow \texttt{for}(l_1,l_2,l_3) \mid \mu, \begin{array}{l} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 :\sim\!: v_1), \\ l_3 \mapsto (v_3 :\sim n) \end{array}} \text{E-NLoop}$$

A loop node streams the incoming list v_1 by concatenating ($:\sim:$) it with the outgoing channel's list v_2. We define list concatenation as usual:

$$xs :\sim\!: (y_1 \sim\!: (y_2 \sim\!: \ldots (y_n \sim\!:)) \ldots)) = (\ldots((xs :\sim y_1) :\sim y_2) \ldots :\sim y_n)$$

The loop node additionally emits the number of streamed elements n to a `reuse` node that pairs it with the gated value:

$$\texttt{reuse}(l_1,l_2,l_3) \mid \mu, \begin{array}{l} l_1 \mapsto (v_1 \sim\!: v_{12}), \\ l_2 \mapsto (v_2 \sim\!: v_{22}), \\ l_3 \mapsto v_3 \end{array} \longrightarrow \texttt{reuse}(l_1,l_2,l_3) \mid \mu, \begin{array}{l} l_1 \mapsto v_{12}, \\ l_2 \mapsto v_{22}, \\ l_3 \mapsto (v_3 :\sim (n, v_1)) \end{array} \quad \text{(E-NReuse)}$$

In our presentation, only state is reused. To do so, the stateful function node receives the state value v_2 with the attached reuse count n on its state channel l_3.

$$\frac{l_s.\texttt{f}_{SF}(v_1) \mid l_s \mapsto v_3 \Downarrow_{\text{R}} v_{22} \mid l_s \mapsto v_3'}{\texttt{n}_{SF}(\texttt{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{ll} l_1 \mapsto (v_1 \sim\!: v_{12}), \\ l_2 \mapsto v_2, \\ l_3 \mapsto ((n, v_3) \sim\!: v_{31}), \\ l_4 \mapsto v_4 \end{array} \longrightarrow \texttt{n}_{SF}(\texttt{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{ll} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 :\sim v_{22}), \\ l_3 \mapsto ((n-1, v_3') \sim\!: v_{31}), \\ l_4 \mapsto v_4 \end{array}} \text{E-NSfReuse}$$

$$\frac{l_s.\texttt{f}_{SF}(v_1) \mid l_s \mapsto v_3 \Downarrow_{\text{R}} v_{22} \mid l_s \mapsto v_3'}{\texttt{n}_{SF}(\texttt{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{ll} l_1 \mapsto (v_1 \sim\!: v_{12}), \\ l_2 \mapsto v_2, \\ l_3 \mapsto ((1, v_3) \sim\!: v_{31}), \\ l_4 \mapsto v_4 \end{array} \longrightarrow \texttt{n}_{SF}(\texttt{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{ll} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 :\sim v_{22}), \\ l_3 \mapsto v_{31}, \\ l_4 \mapsto v_4 :\sim v_3' \end{array}} \text{E-NSfEmit}$$

Rule E-NSfReuse preserves the computed state v_3' with a decremented reuse count in the *incoming* state channel l_3. When the reuse count is 1 then Rule E-NSfEmit emits v_3' to the outgoing state channel. Both rules rely on $\Downarrow_{\text{R}}$ which requires the state value v_3 to be behind a reference l_s. But l_s exists solely to satisfy this requirements. It is not contained anymore in our store μ. That is the result of the translation from an imperative into a functional program in the *ConDRust* compiler. The `trfix` node (which we omit at this point for brevity) follows the same principle to wait for a single recursive call to complete: It enqueues a $\perp$ data value into its incoming arc x_1 that is dequeued only when a recursion is finished and the resulting value was sent along the respective channel x_6 (see T-NFix).

C.3 Determinism

With the operational semantics and typing relation defined, we can prove that evaluation in R_p and as such execution in *ConDRust* is deterministic. The evaluation relation keeps the selection of the next node abstract. It just states the particular data availability requirements

for the nodes to be evaluated. Stateful call nodes have more than one outgoing channel effectively creating subgraphs, i.e., task-level parallelism, and potentially adding more than a single successor into the evaluation relation. Additionally, a for-node emits a whole stream of data values allowing the downstream nodes to be executed repeatedly, i.e., in a pipeline parallel fashion. As such, there may be more than one node ready to be evaluated. The evaluation relation does not specify a concrete evaluation order and applies to any scheduler that follows the defined evaluation rules. As such, we show that evaluation in $t \mid \mu \longrightarrow t' \mid \mu'$ is deterministic.

▶ **Lemma 5** (Single-step Determinism). *If $t : T$ is a well-typed term in $\circledS_p$ then*

$$t \mid \mu, l \mapsto vs \longrightarrow t' \mid \mu', l \mapsto (v \sim: vs) \quad \wedge$$
$$t \mid \mu, l \mapsto vs \longrightarrow t'' \mid \mu'', l \mapsto (v \sim: vs) \Rightarrow t' = t'' \wedge \mu' = \mu''.$$

Proof Sketch. The proof is by induction on a derivation of t and the store μ. Assume a term t' whose activation into the evaluation relation requires data value v available at store location l. For a term t whose evaluation places v into store location l, we distinguish the following two cases:

1. t is the construction of a source channel or
2. t is the evaluation of an upstream node.

The first case is immediate. In the second case, we also know that by the induction hypothesis the activations of this (upstream) node is deterministic. Now assume that t evaluates to t'' by storing a value at location l. By the linear construction of the channels in the dataflow graph, store location l only has a single receiver that is owned by exactly one node. Hence, $t' = t''$ and consequently $\mu' = \mu''$. ◀

Dataflow graphs in $\circledS_p$ are essentially Kahn Process Networks (KPN) [31]. KPNs execute deterministically because incoming arcs have blocking semantics[13] and the executed code of the node is scott-continuous. Our evaluation relation essentially adheres to both of these properties.

However, lemma 5 is insufficient to prove determinism for the whole computation, i.e., for the multi-step evaluation $t \mid \mu \longrightarrow^* v \mid \mu'$:

▶ **Theorem 6** (Determinism). *If $t : T$ is a well-typed term with $t_s \xrightarrow{\text{ConDRust}} t$ then $t \mid \mu \xrightarrow{c}{}^* v \wedge t \mid \mu \xrightarrow{c}{}^* v' \Rightarrow v = v'$.*

The proof of this thereom needs a proof of termination which in turn requires two things: well-founded recursion and cycle freedom. Cycle freedom is based on a formal specification of the transformations in *ConDRust* to prove that the dataflow graph does not have cycles other than the ones guarded by `trfix` nodes. This formalization is outside the scope of this paper and left for future work.

[13] Blocking semantics prevent the construction of a non-deterministic merge node, the explicit notion of non-determinism in dataflow [2].