Safe Nondeterminism in a Deterministic-by-Default Parallel Language

Robert L. Bocchino Jr.[1] Stephen Heumann[2] Nima Honarmand[2] Sarita V. Adve[2]
Vikram S. Adve[2] Adam Welc[3] Tatiana Shpeisman[4]

[1]Carnegie Mellon University rbocchin@cs.cmu.edu
[2]University of Illinois at Urbana-Champaign dpj@cs.uiuc.edu
[3]Adobe Systems awelc@adobe.com
[4]Intel Labs tatiana.shpeisman@intel.com

Abstract

A number of deterministic parallel programming models with strong safety guarantees are emerging, but similar support for nondeterministic algorithms, such as branch and bound search, remains an open question. We present a language together with a type and effect system that supports nondeterministic computations with a deterministic-by-default guarantee: nondeterminism must be explicitly requested via special parallel constructs (marked nd), and any deterministic construct that does not execute any nd construct has deterministic input-output behavior. Moreover, deterministic parallel constructs are always equivalent to a sequential composition of their constituent tasks, even if they enclose, or are enclosed by, nd constructs. Finally, in the execution of nd constructs, interference may occur only between pairs of accesses guarded by atomic statements, so there are no data races, either between atomic statements and unguarded accesses (strong isolation) or between pairs of unguarded accesses (stronger than strong isolation alone). We enforce the guarantees at compile time with modular checking using novel extensions to a previously described effect system. Our effect system extensions also enable the compiler to remove unnecessary transactional synchronization. We provide a static semantics, dynamic semantics, and a complete proof of soundness for the language, both with and without the barrier removal feature. An experimental evaluation shows that our language can achieve good scalability for realistic parallel algorithms, and that the barrier removal techniques provide significant performance gains.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.3.1 [*Software*]: Formal Definitions and Theory; D.3.2 [*Software*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.2 [*Software*]: Language Classifications—Object-oriented languages; D.3.3 [*Software*]: Language Constructs and Features—Concurrent Programming Structures

General Terms Languages, Verification, Performance

1. Introduction

Widely used parallel programming models today (Java, C#, Posix, Win32) are based on a low-level and error-prone concept of threads. These models provide few or no guards against parallel programming errors such as data races, deadlocks, or atomicity violations. Some higher-level programming models are available, or are emerging, that prevent these kinds of errors. However, these models achieve their safety guarantees by greatly restricting side effects, either through functional programming (e.g., STM Haskell [31]) or through dataflow or data-parallel styles of programming (e.g., Concurrent Collections [23], Ct [30]).

There is much recent interest in supporting *deterministic* algorithms within general imperative languages, via static type systems [17], runtime-supported language mechanisms [10, 11, 24, 45, 49, 50], or largely transparent runtime techniques [14, 15, 27, 43]. For algorithms that have deterministic input-output behavior, such models can provide major benefits compared with traditional thread-based programming [18, 38].

There are also important algorithms, however, that do *not* have deterministic input-output behavior, and are not supported by these techniques. Some examples include clustering algorithms, optimization algorithms like branch-and-bound solvers, and graph algorithms like Delaunay mesh refinement. A common feature of such algorithms is that they permit any of multiple possible outputs to be produced for a given input. Such outputs must usually be derived from a controlled set of choices, typically from different, schedule-dependent, orders of evaluating parallel tasks, e.g., evaluating different groups of neighboring points in clustering algorithms. Importantly, the nondeterminism should not simply derive from unpredictable behavior due to data races and atomicity violations. Such behavior is not only potentially erroneous but can also can make proram executions difficult to reason about, e.g., by producing non-sequentially-consistent results. Furthermore, real-world applications are composed of a (potentially large) number of different algorithms, likely to be a mixture of deterministic and nondeterministic ones. Therefore, it is essential to be able to compose deterministic and nondeterministic algorithms in a way that is easy to reason about.

These observations pose two challenges for a safe and realistic parallel programming model: (1) How do we express nondeterminism itself in a disciplined manner that simplifies reasoning about program behavior? (2) How do we allow nondeterministic and deterministic computations to be composed without weakening the deterministic guarantees for the latter?

In this work, we present a Java-based parallel language that supports both deterministic and nondeterministic parallel code in a disciplined manner. Our language has the following major features:

1. *Deterministic parallel operations.* We provide operations that describe deterministic parallel composition of tasks. A deterministic parallel operation enforces *noninterference* between its component tasks (i.e., there are no conflicting reads or writes in any pair of tasks), ensuring that the whole operation behaves like a sequential (and therefore deterministic) composition of

POPL'11, January 26–28, 2011, Austin, Texas, USA.

its component tasks. The noninterference property is enforced *at compile time*. This part is derived from previous work [17].

2. *Nondeterministic parallel operations.* We provide operations that describe potentially nondeterministic parallel composition of tasks. These operations allow interference between tasks, but any such interference is subject to the following guarantees, again enforced *at compile time*. This part is entirely new.

   (a) *Race freedom and sequential consistency.* No execution of a valid program in our language can ever produce a data race. This property is very important, even for nondeterministic codes, because in the Java memory model, race freedom implies sequential consistency, which makes parallel programs much easier to reason about.

   (b) *Strong isolation.* Our language provides an *atomic statement* `atomic` S that executes the enclosed statement S in isolation, i.e., as if there were no interleaving with concurrently executing tasks. The isolation is *strong*, i.e., isolation is provided with respect to *all* concurrent operations, not just other ones occurring in atomic statements. Novel effect system features enable our language to be built on top of an "off the shelf" runtime, such as software transactional memory, that provides only weak isolation. Previous work [6, 31, 42] has also used effects to enforce strong isolation but, as discussed in Section 7, our language is less restrictive, and our guarantees are stronger.

   (c) *Composition of deterministic and nondeterministic operations.* A deterministic parallel operation always behaves as an isolated and sequential composition of its component tasks, *even if the operation encloses, or is enclosed in, a nondeterministic parallel operation*. This property allows local, compositional reasoning about deterministic operations, which we view as essential for a language that supports both deterministic and nondeterministic operations.

   (d) *Determinism by default.* Nondeterminism occurs *only where requested by an explicit nondeterministic operation*, and cannot occur "by accident." Specifically, if a deterministic construct does not encounter any nondeterministic construct for a given input heap state in some execution, then it has deterministic input-output behavior (i.e., it produces the same output heap state and other results in all executions, for that input heap state).

For the deterministic parallel operations (1), we build on Deterministic Parallel Java (DPJ) [17]. In DPJ, the programmer partitions the heap into *regions*, and writes *effect summaries* on methods that describe the method's read and write operations on the regions. The compiler uses the regions and effect summaries to enforce noninterference of parallel tasks at compile time. However, by design, DPJ *completely disallows nondeterministic parallel algorithms*. To provide the nondeterministic operations (2) and their associated guarantees, we must extend DPJ.

For the isolation guarantee 2(b), we build on software transactional memory (STM) [37]. STM is not the only choice here, but it is a good one, as it runs on all platforms (as opposed to hardware transactions, which require special hardware) and provides relatively strong guarantees (isolation and deadlock freedom) with very low programming overhead. However, STM alone is insufficient for our purposes. First, STM implementations usually provide only *weak isolation*, and we want strong isolation. Second, even strong isolation is not enough: it allows data races between accesses outside of transactions, and we want to disallow such data races. Finally, STM introduces significant runtime overheads, including scalar overhead and false conflicts (due to over-synchronization), which can cause poor scalability.

To solve these technical challenges, we extend the DPJ effect system in several ways. First, we add a new kind of effect called an *atomic effect* for tracking when memory accesses occur inside an atomic statement. The atomic effects allow the compiler to guarantee both race freedom (property 2(a)) and strong isolation (2(b)), by prohibiting conflicting memory operations *unless each operation is in an atomic statement*. Second, we introduce new effect checking rules to enforce composition of operations (2(c)) and determinism by default (2(d)). For composition of operations, the extended effect system disallows interference between a deterministic operation and any other concurrent operation *unless the whole deterministic operation is enclosed in an atomic statement*. For determinism by default, the interference is disallowed for deterministic parallel operations, but allowed for nondeterministic parallel operations. Third, to reduce STM overhead, we introduce *atomic regions*, so that the programmer can identify which regions may be accessed in an interfering manner. For operations to other regions, the compiler can remove or simplify the STM synchronization, because such operations never cause conflicts.

Overall, this work makes the following contributions:

1. We present a language that provides the compile-time guarantees 1 through 2(d) stated above. To our knowledge, *no previous language or system has provided all these properties for shared-memory parallel programs through any mechanisms, static or dynamic.* Our language includes novel extensions to the DPJ effect system as discussed above for enforcing race freedom, strong isolation, and determinism by default; and for reducing the runtime overhead of the underlying STM implementation.
2. We formalize our ideas using three formal languages: the first has only deterministic parallel operations, the second adds nondeterministic parallel operations, and the third adds atomic regions. We have developed a full syntax, static semantics, and dynamic semantics for all three languages. Further, we have formally stated the soundness properties given informally above, and proved that the properties follow from the semantic definitions. Here we summarize the key features of the formal language and the essential soundness results; the full details, including proofs, may be found in the lead author's Ph.D. thesis [19].
3. We describe our experience using our language to implement three nondeterministic algorithms: *Delaunay Mesh Refinement* from the Lonestar benchmarks [1], the *traveling salesman problem* (TSP), and *OO7* [25], a synthetic database benchmark. Our experience shows that porting these algorithms form "pure" Java into our language was relatively straightforward and required neither redesign of existing data structures nor restructuring of the algorithms themselves. The language naturally expresses all these algorithms, although the speedups achieved vary depending on the inherent parallelism in the algorithms and performance limitations of the underlying STM. Additionally, judicious use of atomic regions eliminated a large fraction of the STM-related overhead in two out of three benchmarks.

2. Background

In this work, we build on a language called Deterministic Parallel Java (DPJ) [17]. DPJ uses an effect system to enforce deterministic semantics for explicitly parallel programs via compile-time type checking. This section briefly explains the key constructs of DPJ; the details may be found in [17, 19]. In the rest of this paper, we refer to the preexisting language as *basic DPJ*.

DPJ provides a fork-join parallel model: the programmer creates parallel tasks using either a `foreach` statement (for a parallel loop) or `cobegin` block (for a group of mutually parallel state-

ments). DPJ's effect system guarantees that in a well-typed parallel program, any two parallel tasks have *noninterfering effects*. An effect is a set of operations on memory. Two effects interfere if they both access a common memory location and at least one of them writes to that location. The noninterference guarantee for parallel tasks implies deterministic input-output semantics for the computation.

The DPJ effect system works as follows. The programmer assigns every object field and array cell to a *region* and annotates every method with a *method effect summary* stating (a superset of) the reads or write operations performed by the method, in terms of regions. The compiler checks two things: (1) that the effect summaries are a superset of the actual effects in the method body; and (2) that no two parallel statements are interfering. The effect summaries on method definitions enable modular checking of effects.

```
1  class Node<region P> {
2    region L, R;
3    double mass in P;
4    Node<P:L> left in P:L;
5    Node<P:R> right in P:R;
6    void setMass(double mass) writes P {
7      this.mass = mass;              // writes P
8    }
9    void setMassOfChildren(double mass) writes P:* {
10       cobegin {
11         if (left != null) left.setMass(mass);   // writes P:L
12         if (right != null) right.setMass(mass);  // writes P:R
13       }
14   }
15 }
```

Figure 1. Some features of basic DPJ for deterministic parallelism.

Figure 1 illustrates the use of regions and effects in basic DPJ. In line 1, we declare class `Node` to have one region parameter `P`. Line 3 declares field `mass` in region `P`; the actual region of the field is determined when the class is instantiated into a type, as shown in lines 4 and 5. Line 2 declares names `L` and `R` that have static scope (i.e., they are shared by all instances of class `Node`). Lines 4 and 5 declare fields `left` and `right` and place them in regions `P:L` and `P:R`, respectively. The form `P:L` is called a *region path list*, or RPL, and it expresses the hierarchical structure of regions: intuitively, `P:L` and `P:R` are both nested under `P`. The use of `L` and `R` puts the two fields in different regions, while the use of `P` allows different `Node` objects instantiated with different bindings to `P` to have their fields in different regions. Because `L` and `R` are distinct names, `P:L` and `P:R` are guaranteed to refer to different regions, for any common binding to `P`.

Lines 6 and 9 illustrate the use of method effect summaries. Method `setMass` (line 6) has declared effects `writes P`, while `setMassOfChildren` (line 9) has declared effects `writes P:*`, where the `*` is a wildcard representing any sequence of names. If an effect declaration is omitted, it defaults to most general effect (writes the whole heap).

The compiler performs checks (1) and (2) stated above by accumulating the effects of a method, `foreach` statement, or `cobegin` statement. The analysis is simple and local because, at each call site, the declared effects of the invoked method provide the effects of the invocation, after substituting actual for formal region parameters. For example, the effect of `left.setMass(mass)` in line 11 is `writes P:L`, obtained by substituting `P:L` (from the type of left) for the class parameter `P` in the declared method effect `writes P` (the read of field `left` is subsumed because in DPJ, write effects imply read effects). Similarly, the compiler infers the effect of a field access or assignment to $e.f$ by substituting the region named in the type of e for the parameter in the declared region of the field f.

As an example of check (1) (correct method effects), the effect of `setMass` is legal because the method body writes to field `mass` in region `P` and has no other heap effects. As an example of check (2) (parallel noninterference), the compiler infers that the effect of lines 11 and 12 are `writes P:L` and `writes P:R`, respectively. Because `P:L` and `P:R` must be disjoint regions, for any common binding to `P`, the effects are noninterfering.

Although this example is somewhat simplistic, these and other features of DPJ can express a range of realistic parallel idioms [17], including parallel updates on arrays of objects, parallel traversals and updates of a tree, in-place divide-and-conquer on arrays, and commutative operations within parallel tasks.

3. Language Support for Nondeterminism

We now informally describe the language mechanisms for nondeterministic parallel control, parallel safety guarantees, and optimization support. We illustrate the new language features with a running example of the *traveling salesman problem* (TSP). Section 4 describes the language more formally.

3.1 The TSP Computation

The traveling salesman problem, or TSP, is the well-known problem of finding a shortest cycle in a weighted graph that visits all the nodes once (i.e., a Hamiltonian cycle). TSP can be solved by *branch and bound search*, a common algorithm for solving optimization problems and a classical example of a nondeterministic computation. Figures 2–4 show simplified Java-like pseudocode for TSP. The global data (lines 1–13) include a weighted graph that is the input to the program; a priority queue for storing the paths being explored; and a "best" (i.e., shortest) tour, which is refined as the computation progresses, eventually storing the answer. Two regions are used to hold the data: `ReadOnly` for fields that will not be modified during the computation, and `Mutable` for those that will be. The priority queue's type `PriorityQueue<Path<ReadOnly>, Mutable>` indicates that it contains objects of type `Path<ReadOnly>`, and that the internal data used to represent the queue itself is in region `Mutable`. The main computation loop (lines 15–23) iterates in parallel over several worker tasks. Each task generates a prefix to search (using the pseudocode in Figure 3) and adds it to a priority queue. When all prefixes have been generated, the tasks remove prefixes from the priority queue and search them (using the pseudocode in Figure 4), until there are no more prefixes to search.

3.2 Nondeterministic Parallel Control

To express nondeterministic parallel computations, we introduce a parallel loop denoted `foreach_nd`, where `nd` stands for "nondeterministic." Line 15 in Figure 2 shows an example. This construct is identical to `foreach` in basic DPJ [17], except it says explicitly that conflicting accesses, and therefore potential nondeterminism, are allowed between the loop iterations of `foreach_nd`. We also introduce a `cobegin_nd` construct corresponding to `cobegin` in basic DPJ. Collectively, we refer to these four (`foreach`, `foreach_nd`, `cobegin`, and `cobegin_nd`) as *parallel constructs*.

The resulting parallel control structure is just fork-join parallelism and can be represented as a static task graph, where each node or *task* is a single iteration of a parallel loop (`foreach` or `foreach_nd`) or a single statement in a `cobegin` or `cobegin_nd`. All four parallel constructs have an implicit "join" synchronization at the end of the construct for the tasks of the construct. The (directed) edges in the task graph represent either program order, or forking at the start of a parallel construct, or the "join" synchronization at the end of a parallel construct. Two tasks are *concurrent* if they are not ordered in the task graph. Two memory accesses are concurrent if they occur in concurrent tasks.

```
1  /* Regions for partitioning data */
2  region ReadOnly, atomic Mutable;
3
4  /* Graph we are working on; immutable */
5  Graph<ReadOnly> graph in ReadOnly = the TSP graph;
6
7  /* Priority queue for tour prefix paths */
8  final PriorityQueue<Path<ReadOnly>, Mutable> priorityQueue =
9      new PriorityQueue<Path<ReadOnly>, Mutable>();
10 priorityQueue.add(new Path<ReadOnly>(startNode));
11
12 /* The answer */
13 Path<ReadOnly> bestTour in Mutable = infinite path;
14
15 foreach_nd(int i in 0, NWORKERS) {
16     Path<ReadOnly> prefix = null;
17     do {
18         atomic {
19             prefix = generateNextPrefix();
20         }
21         if (prefix != null) searchAllToursWithPrefix(prefix);
22     } while (prefix != null);
23 }
```

Figure 2. Global data and main computation for the Traveling Salesman Problem.

```
1  Path generateNextPrefix() reads ReadOnly writes Mutable {
2      while (!priorityQueue.isEmpty() &&
3             priorityQueue.best().length() < bestTour.length()) {
4          Path<ReadOnly> prefix = priorityQueue.removeBest();
5          if (prefix.nodeCount() > PREFIX_CUTOFF) {
6              return prefix;
7          } else {
8              for (each edge edge that can be added to prefix
9                    while staying under bestTour.length()) {
10                 Path<ReadOnly> newPrefix =
11                     new Path<ReadOnly>(prefix, edge);
12                 priorityQueue.add(newPrefix);
13             }
14         }
15     }
16     return null;
17 }
```

Figure 3. Generating the next tour prefix.

```
1  void searchAllToursWithPrefix(Path<ReadOnly> prefix)
2      reads ReadOnly writes atomic Mutable {
3      for (each Hamilton cycle tour in graph with prefix prefix) {
4          atomic {
5              if (tour.length() < bestTour.length()) {
6                  bestTour = tour;
7              }
8          }
9      }
10 }
```

Figure 4. Searching all tours with a given prefix.

The specific parallel constructs used to fork and join tasks are not fundamental to our work. The language mechanisms used to enforce safety properties (described next) can be applied directly to other fork-join parallel programming languages, e.g., Cilk [16], a large subset of OpenMP [4], and potentially other parallel languages in which the compiler can identify all groups of concurrent tasks. Distinguishing the constructs that permit interference (i.e., may be nondeterministic) from those that do not (and so are deterministic) is a useful property, but again, not necessary for any of our other guarantees.

3.3 Safety Properties for Nondeterministic Code

As stated in the introduction, the goal of our language design and type system is to achieve four safety guarantees for nondeterministic and deterministic code: (i) data race freedom; (ii) strong isolation for nondeterministic parallel constructs; (iii) sequential equivalence for deterministic parallel constructs; and (iv) a property we call *determinism by default*, defined below. These four properties give programmers a simple, elegant execution model for reasoning about partly nondeterministic programs. Below, we discuss the language mechanisms for expressing synchronization, the effect system features for enforcing the properties, and the resulting execution model seen by programmers.

Synchronization: To ensure correctly synchronized accesses in the presence of interference (defined in Section 2), we add an *atomic statement* to the language. This construct is similar to previous work [8, 28, 32, 33], except that in conjunction with our effect system, discussed below, our atomic statements provide stronger guarantees. A statement `atomic` S indicates that S is to be run as if *all other concurrent execution* were suspended while S is executing. This is called strong isolation [40, 47].

With reference to the TSP example, in Figure 2 each call to `generateNextPrefix` is enclosed by an atomic statement that protects the accesses to the shared priority queue. Note that while the calls to `generateNextPrefix` are effectively serialized, each worker can start its call to `searchAllToursWithPrefix` as soon as its call to `generateNextPrefix` is done, in a pipelined manner. This pattern can achieve good speedups because most of the work in this code is done in `searchAllToursWithPrefix`. In Figure 4, an atomic statement protects the concurrent updates to `bestTour`.

An atomic statement may appear inside any of the four parallel constructs, as well as inside other atomic statements. Two nested atomic statements in the same task are "flattened": that is, the inner `atomic` becomes a no-op, and atomicity is enforced entirely at the outer `atomic`. If a parallel task created in an atomic statement contains a nested atomic statement, then the nesting behaves in the standard way: the inner `atomic` enforces isolation with regard to other tasks created by the immediately enclosing parallel construct, while the outer `atomic` enforces isolation as to tasks created by any outer enclosing parallel construct.

Effect System: We now discuss how our effect system enforces the four safety properties stated at the outset of this section.

Data Race Freedom and Strong Isolation: We use the following strategy to ensure both data race freedom and strong isolation. First, a transactional runtime guarantees at least weak isolation of atomic statements (i.e., isolation between different atomic statements, but not between atomic statements and unguarded code). Second, the effect system ensures that for any pair of conflicting memory accesses, each access occurs inside an atomic statement. For example, in Figure 4, any two concurrent accesses to `bestTour` are both enclosed in instances of the atomic block at line 4 (the concurrency is created by the `foreach_nd` at line 15 of Figure 2). This requirement ensures strong isolation, because no conflicts between unguarded memory accesses and atomic statements are allowed. It also ensures race freedom, because no conflicts between pairs of unguarded accesses are allowed.

Notice two things about our strategy. First, our language may be built on top of a standard software transactional memory (STM) implementation, which typically guarantees only weak isolation for performance reasons. Second, our strategy prohibits all data races. Even TM systems with strong isolation generally allow data races between pairs of accesses occurring outside any transaction.

To accomplish the effect checking, we extend the DPJ effect system to distinguish effects that are *atomic* (meaning the effect occurred inside an atomic statement) from effects that are *non-atomic* (meaning the effect occurred outside any atomic statement). The compiler ensures that interference occurs *only between atomic effects*.

To enable sound modular reasoning about method invocations, we make atomic effects explicit in method effect summaries. For

example, the effect `writes atomic Mutable` in the summary for `searchAllToursWithPrefix` (Figure 4) says that any possible writes to region `Mutable` occur inside atomic blocks in the body of the method or its callees. In checking method effect summaries, our system is sound but conservative: it is correct to summarize a write to region R occurring inside an `atomic` block as either `writes atomic` R or simply `writes` R; the latter is more conservative than necessary but is correct. However, it is not correct to summarize an access occurring outside any atomic section as an atomic effect, because such an effect would report a transactional guard when in fact there is none.

For example, the effect system can verify that all interfering accesses within the `foreach_nd` in Figure 2 are atomic effects. First, the variable `prefix` is local to each task and so generates no conflicts across tasks. Second, according to the effect summary for `generateNextPrefix` (Figure 3), the method invocation in line 19 produces conflicting effects on region `Mutable`. These effects are enclosed within the atomic statement starting at line 18 and so are recorded as atomic effects that may interfere. Third, the call to `searchAllToursWithPrefix` is not within an atomic statement; but according to its effect summary it generates only read effects (which do not interfere with themselves) and atomic write effects (which are allowed to interfere with themselves, and are to a different region from the read effects).

Sequential Equivalence for Deterministic Constructs: An important property we wish to preserve from basic DPJ is *sequential equivalence* for deterministic constructs: that is, `foreach` and `cobegin` are equivalent to the sequential execution of their constituent tasks in program order. To enforce this property, we obviously need to disallow interference between `cobegin` or `foreach` branches, even if the interfering effects are atomic. For example, this program is not allowed:

```
cobegin {
  atomic x = 0;
  atomic x = 1;
}
```

For this we just have a simple typing rule that interference between atomic effects is allowed only inside `foreach_nd` or `cobegin_nd`.

However, that is not enough, because interference can also occur between a deterministic task and a concurrent nondeterministic task. For example, consider the following program:

```
z = 0;
cobegin_nd {
  cobegin { atomic x = z; atomic y = z; } // S1, S2
  atomic z = 1;   // S3
}
```

This program could produce the result `x` $= 1,$ `y` $= 0$ by executing `S2; S3; S1`. This result violates sequential equivalence of `cobegin`, because it does not correspond to any sequentially consistent execution of the program where the `cobegin` block is executed in program order. Instead, we wish to ensure that a `foreach` or `cobegin` executes in isolation, even if it appears inside `foreach_nd` or `cobegin_nd`.

Our solution to this problem is to convert atomic effects occurring inside a deterministic construct to non-atomic effects when propagating them to the outer context. In the example above, when checking interference, the compiler sees *nonatomic* reads to `z` in the first `cobegin_nd` branch: those reads occurred in atomic statements, but became nonatomic when passing outwards across the `cobegin`. On the other hand, the second branch has atomic writes to `z`. Therefore the `cobegin_nd` branches have illegal read-write interference (i.e., not both guarded by atomic) on `z`. To write this program legally in our language, the programmer could put the whole `cobegin` in an atomic statement.

Determinism by Default: Finally, by virtue of the isolation of deterministic constructs, and the noninterference between their internal tasks, both discussed above, we have the following property: if a deterministic construct does not dynamically execute any nondeterministic construct, then the execution of the deterministic construct is, in fact, deterministic. That is, a given input heap state to the deterministic construct always produces a fixed result value and fixed output heap state. We refer to this property as *determinism by default*: nondeterministic input-output behavior may be introduced *only* by the execution of an explicit nondeterministic construct.

Implications for Programmers: The properties discussed above, and treated more formally in the next section, provide two key benefits for programmers. First, concurrency errors such as data races or unintentional nondeterminism will be detected via compile-time type checking; this benefit existed in base DPJ for deterministic programs and has now been extended to nondeterministic ones. Second, once a program has been type checked, the above properties greatly simplify how programmers can reason about the possible (nondeterministic) execution behaviors.

With regard to the second point, many programmers and testing tools analyze program behavior by reasoning about the possible interleavings (or schedules) of parallel operations. The above properties simplify this reasoning in several important ways (we focus on `cobegin` here without loss of generality; `foreach` is analogous):

1. We only need to consider interleavings of isolated `atomic`s, `cobegin`s, and unguarded accesses, because of strong isolation and sequential equivalence of `cobegin`. The `nd` constructs do not constrain interleavings.
2. We can reason about the tasks of a `cobegin` sequentially: the first task can be fully evaluated without any intervening accesses from elsewhere, immediately followed by a complete evaluation of the second task.
3. `cobegin_nd` provides the only source of nondeterminism. Even within such a construct, the effect system guarantees that any block of code that is outside an atomic statement and does not execute any atomic statement (call this an *atomic-free section*) cannot interfere with any concurrent task. Therefore, programmers need not consider interactions between any atomic-free sections when reasoning about program behavior.

Put together, these observations mean that *the only source of multiple interleavings is from different orderings of atomic sections*, thereby significantly reducing the number of interleavings that programmers must consider. Furthermore, programmers can control the granularity of the atomic sections to control the number of possible interleavings.

The following example illustrates these observations (assume the `S` terms are all atomic-free statements).

```
cobegin_nd {
  { S11; S12; atomic S13 }
  { S21; atomic S22; S23 }
}
```

Even if all the statements are primitive operations (reads or writes), if sequential consistency is not guaranteed, then up to $6! = 720$ different interleavings are possible. If sequential consistency holds, then there are still up to 20 different interleavings. In our language, however, *we may consider only two sequentially consistent interleavings*: one with `atomic S13` appearing before `atomic S22`, and vice versa. For example, any execution generated by our language is equivalent to executing the entire first `cobegin_nd` branch before the entire second branch, or vice versa.

3.4 Performance: Removing Unneeded Barriers

We use a Software Transactional Memory (STM) runtime system to implement the `atomic` construct because STM provides weak atomicity, better composability than locks, and potentially better scalability because of optimistic rather than pessimistic synchronization. One key drawback of STMs is the overhead due to *transactional read and write barriers* for every load or store to shared data (e.g., see [52]). These barriers are snippets of code, often automatically inserted by a transactions-aware compiler, that invoke the STM runtime to implement some transactional concurrency control protocol. The barriers can either read and write shared memory directly (so-called *in-place update* STM) and *undo* all transactional operations when a transaction aborts, or they can buffer updates into a private data structure (so-called *write buffering* STM) and apply all the buffered changes into shared memory when a transaction successfully commits. In both cases, barriers can incur significant overhead and minimizing them is essential for performance.

We observe that we can use the region and effect system to remove unnecessary STM barriers, where there is no interference. However, the effect system as described so far does not carry enough information to perform this analysis locally. For example, suppose a method m reads a variable x inside an atomic section. Then the read needs a barrier if and only if m is invoked in some context where there is interference on x. There is no information in the method body that enables the compiler to make that judgment; interprocedural analysis would be required.

However, with a slight extension to the effect system, we can enable *local* reasoning about this kind of noninterference. Specifically, we have the effect system distinguish two kinds of regions: those that may interfere (and so need barriers everywhere) and those that cannot (and so do not need read barriers anywhere). We call the first kind *atomic regions*. The programmer can declare a region to be `atomic`, such as region `Mutable` on line 2 of Figure 2.

The key benefit is that, for a non-atomic region, the compiler can remove read barriers entirely and, assuming STM using in-place updates, it can turn write barriers into logging-only barriers (synchronization is not needed, because there is no interference, but transactions must still log the old value on writes, in case the transaction aborts). This completely eliminates the barrier overheads for read-only shared data. It also substantially reduces the barrier overheads for task-local data and noninterfering shared data.

To enable sound reasoning about atomic regions and barrier elimination, we require some constraints on the use of these regions. Any region declaration (field region, local region, or region parameter) may be declared to be atomic. We impose the following requirements:

- When instantiating a type, an atomic (respectively, non-atomic) region parameter may only be passed an atomic (respectively, non-atomic) region name as the argument. This is straightforward to enforce using the region declarations.
- A region that is involved in interfering effects must be declared atomic. This is enforced by the compiler as described below.

The barrier elimination also requires a refinement in the semantics of atomic effects described in the previous section. An effect in an atomic statement is marked atomic *only if it operates on an atomic region*. For example, the read of region `ReadOnly` in Figure 4 (due to the operation `tour.length()`) does not generate an atomic effect, even though it is inside the `atomic` block at line 4. The write to region `Mutable` does generate an atomic effect. If region `Mutable` had *not* been declared `atomic`, the write to `bestTour` would generate a non-atomic effect. The compiler would then flag the effect declaration at line 2 of Figure 4 as an error because an atomic effect does not cover a non-atomic effect, as noted earlier.

Programs	$\mathcal{P}$	::=	$\mathcal{R}^*\ \mathcal{C}^*\ e$
Classes	$\mathcal{C}$	::=	`class` C`<`ρ`>` { $F^*\ M^*$ }
Region Names	$\mathcal{R}$	::=	`region` r
Fields	F	::=	$T\ f$ `in` R
Methods	M	::=	$T\ m(T\ x)\ E$ { e }
Regions	R	::=	$r \mid \rho$
Types	T	::=	C`<`R`>`
Effects	E	::=	$\emptyset$ | `reads` R | `writes` R | $E \cup E$
Expressions	e	::=	`this.`f | `this.`f`=`e | $e.m(e)$ | v | `new` T | `seq(`e`,`e`)` | `cobegin(`e`,`e`)`
Variables	v	::=	`this` | x

Figure 5. Core language syntax. C, ρ, f, m, and x are identifiers.

We can now explain how the last rule above is enforced. If a region is not marked atomic but has an effect that causes interference in some parallel construct, the compiler will detect an error either at the parallel construct or at the method effect summary. For example, if region `Mutable` were not marked `atomic`, the write to `bestTour` would generate a normal effect `writes Mutable`. This would cause the effect summary at line 2 of Figure 4 to be flagged as an error, as noted above. If the effect summary were changed to not mark the write effect atomic, then the call to `searchAllToursWithPrefix` at line 21 of Figure 2 would generate a nonatomic effect, and the compiler would report the interference there.

4. Formal Semantics and Soundness

To make precise the ideas discussed in the previous section, we have studied three variants of the same formal language, each one building on the last:

1. The first variant, which we call the *deterministic language*, is a simple expression language with regions, effects, and deterministic parallel composition. It is a version of Core DPJ [17] simplified to focus on the key elements for this work.
2. The second variant, which we call the *deterministic-by-default language*, adds nondeterministic parallel composition, atomic expressions, and atomic effects to the deterministic language.
3. The third variant, which we call the *atomic regions language*, adds atomic regions for removing or simplifying transactional barriers.

Without loss of generality, we only include `cobegin` and `cobegin_nd` in these simple languages; the treatment for `foreach` and `foreach_nd` is similar.

4.1 Overview of Language Variants

We first explain the syntactic structure of all three languages, and we summarize the soundness guarantees that each one provides. In the following subsections, we explain the formal semantics of each language variant, state the soundness guarantees more formally, and sketch how the guarantees follow from the semantic definitions. The full details, including all the semantics rules and proofs of all the claims, may be found in the lead author's Ph.D. thesis [19].

Deterministic Language: Figure 5 gives the syntax of the deterministic language. A program $\mathcal{P}$ consists of zero or more region declarations, zero or more class definitions, and an expression to evaluate. A class $\mathcal{C}$ consists of a class name C, a region parameter ρ, zero or more field declarations, and zero or more method declarations. A field F specifies a type, a field name, and a region. A method M consists of a return type, a method name, a formal parameter type, a formal parameter, an effect, and an expression to

Effects	E	::=	... | `atomic reads` R | `atomic writes` R
Expressions	e	::=	... | `cobegin_nd(`e`,`e`)` | `atomic` e

Figure 6. Syntax of the deterministic-by-default language (extends Figure 5).

Regions	$\mathcal{R}$	::=	... | `atomic region` r
Classes	$\mathcal{C}$	::=	... | `class` C`<atomic` ρ`> {` F^* M^* `}`

Figure 7. Syntax of the atomic regions language (extends Figure 6).

evaluate. A region R is either a region name r or a region parameter ρ. A type T is a class instantiated with a region parameter, C<R>. An effect E is a possibly empty union of read effects and write effects on regions.

For expressions e, we model field access, field assignment, method invocation, variables, new objects, sequential composition (`seq`), and deterministic parallel composition (`cobegin`). A variable v is `this` or a method formal parameter x. The operational semantics of the first five expressions in Figure 5 is exactly as in Java. The last two expressions evaluate both component expressions (either sequentially or in parallel) and return the value of the second component as the value of the entire expression.

The deterministic language provides the following semantic guarantees, stated more formally as Theorems 1–2 in Section 4.2. They follow from the fact that the executions of the two branches of any `cobegin` expression are required to be noninterfering:

1. *Equivalence of `cobegin` and `seq`:* In terms of the final result (final value produced and final heap state), there is no difference between executing `cobegin(`e`,`e'`)` and `seq(`e`,`e'`)`. As a consequence, the entire program is guaranteed to behave like a sequential program (the one that results by replacing `cobegin` everywhere with `seq`).
2. *Determinism:* If an expression e evaluates to completion, then the value it produces is deterministic. Moreover, if e is evaluated in a sequential context (i.e., not inside a `cobegin`), then the final heap state is deterministic. In particular, the final heap produced by a terminating execution of the whole program is deterministic.

Deterministic-by-Default Language: Figure 6 shows the additional syntax for the deterministic-by-default language. We extend the syntax of effects to record atomic effects. We also add (1) `cobegin_nd`, which is the same as `cobegin`, except that it allows interference guarded by atomic expressions; and (2) expressions `atomic` e, which signal that expression e should be executed in *isolation*: that is, as if it were executed all at once, with no interleavings from the rest of the execution.

The deterministic-by-default language provides the following semantic guarantees, stated more formally as Theorems 3–6 in Section 4.3:

1. *Race freedom and sequential consistency:* Program execution contains no data race. This result follows because the effect system requires that all parallel interference occur between pairs of accesses guarded by atomic expressions. Further, in the Java memory model, race freedom implies sequential consistency, i.e., one can reason about execution as a *program-ordered* interleaving of memory operations.
2. *Strong isolation:* For the same reason that the program is race free, expressions `atomic` e execute e in isolation, *even if the underlying implementation guarantees only weak isolation*. Moreover, the effect system disallows any interference between the `cobegin` and concurrent operations that would violate isolation of the `cobegin`. Therefore, every `cobegin` expression executes in isolation. Together, race freedom and strong isolation imply that execution is a sequentially consistent interleaving *of isolated expressions*.
3. *Equivalence of `cobegin` and `seq`*: Because `cobegin(`e`,`e'`)` executes in isolation, it is equivalent to an isolated execution of `seq`, i.e., `atomic seq(`e`,`e'`)`. As discussed in Section 3.3, for the deterministic-by-default language, we make `cobegin` behave like `atomic seq`, and not just `seq`, to guarantee that `cobegin` executes deterministically, *even inside a `cobegin_nd`.*
4. *Determinism by default:* Both atomic and cobegin expressions execute deterministically in the same sense as discussed for the deterministic language, *even inside a `cobegin_nd`, unless* they contain a dynamic instance of `cobegin_nd`.

Atomic Regions Language: The third variant of the formal language allows some regions to be marked `atomic`, and *only operations on those regions generate atomic effects*. Operations on non-atomic regions *never generate atomic effects, even in an atomic expression*. Figure 7 shows the new syntax.

The execution semantics of this language variant is identical to that of the deterministic-by-default language, except that the compiler can distinguish, and potentially optimize, operations within an atomic expression that never interfere with concurrent tasks. In Section 5, we discuss a prototype compiler that uses these rules to optimize our STM by omitting or simplifying barriers (inside an atomic expression) for such noninterfering operations.

4.2 Deterministic Language

Static Semantics: The typing is done with respect to an environment Γ, which consists of elements (v, T) stating that variable v has type T. The key rule for *subeffects* (i.e., when one effect conservatively summarizes another, written $\Gamma \vdash E \subseteq E'$), is that a write effect on region r covers a read effect on the same region r:

$$\textsc{SE-Reads-Writes}\quad \frac{}{\Gamma \vdash \texttt{reads}\ R \subseteq \texttt{writes}\ R}$$

The rules for typing programs, classes, fields, etc., are straightforward. The rule for typing methods enforces *effect subsumption*: that is, that a method's actual effect must be a subeffect of its declared effect:

$$\textsc{Method}\quad \frac{\Gamma \vdash T_r \quad \Gamma \vdash T_x \quad \Gamma \vdash E \quad \Gamma \cup (x, T_x) \vdash e : T_r, E' \quad \Gamma \vdash E' \subseteq E}{\Gamma \vdash T_r\ m(T_x\ x)\ E\ \{\ e\ \}}$$

The key rules for *noninterfering effects* (i.e., effects that may safely go in parallel, with deterministic composition, written $\Gamma \vdash E \# E'$) are that reads never interfere with reads, and writes never interfere with reads or writes to different regions:

$$\textsc{NI-Reads}\quad \frac{}{\Gamma \vdash \texttt{reads}\ r \# \texttt{reads}\ r'} \qquad \textsc{NI-Writes}\quad \frac{r \neq r'}{\Gamma \vdash \texttt{writes}\ r \# \texttt{writes}\ r'}$$

$$\textsc{NI-Reads-Writes}\quad \frac{r \neq r'}{\Gamma \vdash \texttt{reads}\ r \# \texttt{writes}\ r'}$$

As in Core DPJ [17, 19], every expression has a type and an effect. The rules for typing expressions e with type T and effect E ($\Gamma \vdash e : T, E$) are straightforward. The most important rule says that in parallel composition, the effects of the expressions being evaluated in parallel must be noninterfering:

$$\textsc{Cobegin}\quad \frac{\Gamma \vdash e : T, E \quad \Gamma \vdash e' : T', E' \quad \Gamma \vdash E \# E'}{\Gamma \vdash \texttt{cobegin(}e\texttt{,}e'\texttt{)} : T', E \cup E'}$$

Dynamic Semantics: We give a small-step operational semantics describing the recursive reduction of expressions.

Execution State: The execution state is (e, H), consisting of an expression to evaluate and a heap. A heap H is a partial function from object references to pairs (O, T), where O is an object, and T is the type of O. An object O is a mapping from field names f to object references o. null is a special reference that is in $\mathsf{Dom}(H)$ but does not map to an object. Attempting to invoke a method of null causes the execution to fail. We extend the static syntax of expressions to represent computations:

$$e ::= \ldots \mid o \mid (e, \Sigma, E) \mid e_i$$

The additional expressions have the following meanings: object references o are the values produced by reducing expressions; a *local execution state* (e, Σ, E) records an expression e to evaluate, an environment Σ containing the bindings for the free variables in e, and an effect E of reducing e; and the indices i enable us to say unambiguously which expression is reduced in a given execution step, as explained below.

A program execution is a sequence of steps

$$((e_{\mathcal{P}}, \emptyset, \emptyset)_i, \mathsf{null}) \rightarrow^* (e_i, H),$$

for some i, e, and H, where $e_{\mathcal{P}}$ is the main program expression, i is an arbitrary index denoting the top-level expression in the reduction, e_i is the evolution of expression $(e_{\mathcal{P}}, \emptyset, \emptyset)_i$, and H is the evolved heap (represented as a domain containing `null` plus all object references o added during the execution). A terminating execution has $e_i = (o, \emptyset, E)_i$, where o is the "answer" computed by the program, and E is the union of all effects on H done in the execution.

Expression Semantics: Field access and assignment work in the standard way, except that we track dynamic effects, to state and prove the soundness results. As an example of the effect tracking, we give the rule for field access:

$$\textsc{Dyn-Field-Access}\quad \frac{(\texttt{this}, o) \in \Sigma \quad H(o) = (O, C\texttt{<}r\texttt{>}) \quad F(C, f) = T\ f\ \texttt{in}\ R}{((\texttt{this}.f, \Sigma, \emptyset), H) \rightarrow ((O(f), \Sigma, \texttt{reads}\ \sigma_{C\texttt{<}r\texttt{>}}(R)), H)}$$

The function $\sigma_{C\texttt{<}r\texttt{>}}$ substitutes the region argument r for the parameter ρ of class C. The rule for field assignment is similar, except that the right-hand-side subexpression is evaluated first, the heap is updated, and the effect is a write instead of a read.

For evaluation of subexpressions, we use the following standard rule:

$$\textsc{Dyn-Subexp}\quad \frac{(e, H) \rightarrow (e', H')}{(e'', H) \rightarrow (e''[e_i \leftarrow e'_i], H')}$$

It says that if we can reduce expression e to e' starting with heap H, and e appears with index i as a subexpression of e'', then we can reduce e'' by rewriting the subexpression e_i in place and updating the heap.

The rules for `cobegin` illustrate subexpression evaluation:

$$\textsc{Dyn-Cobegin-Eval}\quad \frac{\mathsf{fresh}(i) \quad \mathsf{fresh}(j)}{((\texttt{cobegin(}e\texttt{,}e'\texttt{)}, \Sigma, \emptyset), H) \rightarrow (\texttt{cobegin((}e, \Sigma, \emptyset)_i\texttt{,(}e', \Sigma, \emptyset)_j\texttt{)}, H)}$$

$$\textsc{Dyn-Cobegin-Accumulate}\quad \frac{}{(\texttt{cobegin((}o, \Sigma, E)_i\texttt{,(}o', \Sigma, E'\texttt{)}_j\texttt{)}, H) \rightarrow ((o', \Sigma, E \cup E'), H)}$$

DYN-COBEGIN-EVAL creates two new subexpressions with fresh indices i and j for evaluation in environment Σ. The evaluation steps of e_i and e_j can be arbitrarily interleaved, via rule DYN-SUBEXP. This interleaving models the parallelism. When both are done, DYN-COBEGIN-ACCUMULATE accumulates the results into the top-level expression. Field assignment, method invocation, and sequential composition are similar, except that the subexpressions are evaluated in a fixed sequential order (so there is no parallelism, except inside a `cobegin`). The rules are entirely standard, and are stated in full in the thesis.

Soundness Results: As summarized in Section 4.1, there are two main soundness results for the core deterministic language. To state the results, we call out the reduction of a particular expression inside the evolution of the whole program. We write

$$((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} (e'_i, H')$$

to denote a *reduction of expression* i occurring in program execution. That means $\mathcal{P}$ is well-typed with main expression $e_{\mathcal{P}}$, and there is a program execution

$$((e_{\mathcal{P}}, \emptyset, \emptyset)_j, \mathsf{null}) \rightarrow^* (e''_j, H) \rightarrow^* (e'''_j, H'),$$

such that e''_j contains $(e, \Sigma, \emptyset)_i$, which is the first appearance of expression i in the execution; and e'''_j contains e'_i. (e''_j, H) is called the *initial state* of the reduction, and (e'_j, H') is called the *final state*.

Our first result states that there is no semantic difference in this language between `seq` and `cobegin`: at any point in the execution that is an initial state for a `cobegin` reduction, we can replace `cobegin` with `seq` and get exactly the same results. The proof follows directly from the noninterference property guaranteed by the static and dynamic semantics.

Theorem 1 (Equivalence of `cobegin` and `seq`)**.**

$$((\texttt{cobegin}(e, e'), \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$$

if and only if

$$((\texttt{seq}(e, e'), \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$$

with the same initial state, except that expression i is as shown.

The second result states that expression evaluation is input-output deterministic, up to the choice of object reference names. The proof follows from the fact that once `cobegin` is replaced by `seq` everywhere according to Theorem 1, the only nondeterminism left in the rules is the choice of reference names and expression indices.

Theorem 2 (Input-Output Determinism)**.** *If $((e, \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_j, H')$ and $((e, \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o', \Sigma, E')_j, H'')$ with the same initial state, then $o \cong o'$, where $\cong$ denotes equivalence up to renaming object references, and $E = E'$. Moreover, if $(e, \Sigma, \emptyset)_j$ is not a subexpression of any `cobegin` expression, then $H \cong H'$.*

4.3 Deterministic-by-Default Language

Static Semantics: Figure 8 gives the static semantics of atomic effects. Rule SE-ATOMIC-1 formally expresses the idea that non-atomic effects cover atomic effects: that is, if E occurred in an atomic expression, then we can summarize the effect as either `atomic` E or E. Note that the converse is not true, because we cannot soundly report an atomic effect where there is no atomic expression. Rule SE-ATOMIC-2 provides the subeffect relation for two atomic effects. Rule NI-ATOMIC says that an atomic effect is noninterfering if the underlying effect is.

The judgment $\Gamma \vdash \mathsf{nondet}(E, E')$ states that it is safe to run expressions with effects E and E' nondeterministically in parallel, inside a `cobegin_nd`. Figure 8 gives the key rules for making the judgment.

Figure 9 gives the rules for typing nondeterministic parallel composition and atomic expressions. Rule COBEGIN_ND is similar to COBEGIN, except that interfering effects are allowed if they

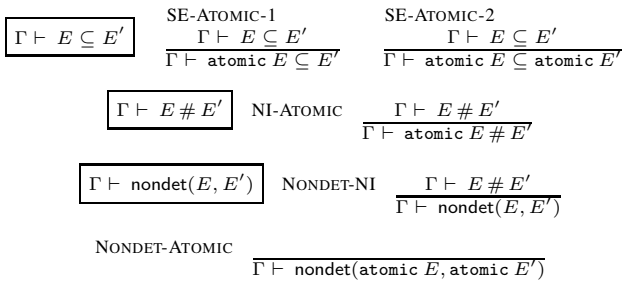

Figure 8. Static semantics of atomic effects (selected rules).

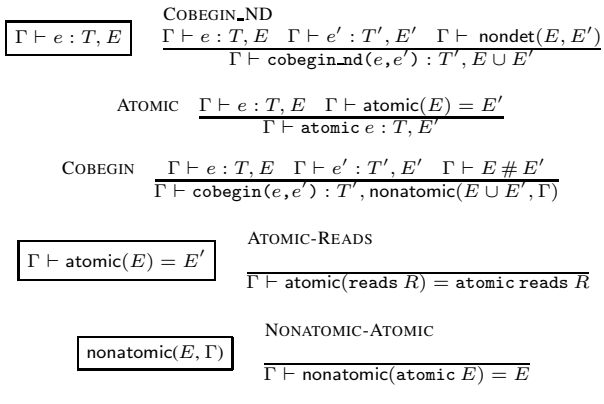

Figure 9. Static semantics of `cobegin_nd` and atomic expressions (selected rules). The judgment $\Gamma \vdash \mathsf{atomic}(E) = E'$ says that E' is the effect obtained after adding `atomic` from all reads and writes in E; and $\Gamma \vdash \mathsf{nonatomic}(E)$ is the reverse.

are both guarded by atomic expressions. Rule ATOMIC collects the effect E of the expression e, then marks all the constituent read and write effects atomic, to reflect the fact that E is occurring in an atomic expression. Finally, rule COBEGIN has changed. In addition to checking noninterference, as in the basic language, the new rule converts all atomic effects occurring inside the `cobegin` to ordinary effects using the judgment $\Gamma \vdash \mathsf{nonatomic}(E) = E'$. This ensures that *no atomic effects are ever propagated outward from inside a* `cobegin`. The last rule is key to ensuring that `cobegin` executes in isolation, as discussed in Section 3.3.

Dynamic Semantics: We describe the dynamic semantics of the nondeterministic language in two parts, the first operational and the second non-operational. The first, operational, part is just the same semantics as for the basic language (Section 4.2), with a few minor adjustments to accommodate the new features. The second, non-operational part, describes a *weak isolation constraint* on execution histories generated by the operational part. The overall dynamic semantics comprises all execution histories described by the operational semantics that also satisfy weak isolation. In practice, weak isolation would be enforced by a runtime implementation (such as software transactional memory [37]).

Operational Semantics of Expressions: The operational semantics is identical to the one described in Section 4.2, with three changes. First, we add a rule to execute `cobegin_nd`; it is identical to the rule for `cobegin` shown in Section 4.2 (i.e., in the operational semantics, there is no difference between executing `cobegin_nd` and `cobegin` — the difference is all in the static semantics).

Second, we add a rule for executing an expression `atomic` e. We execute e, then mark all its effects atomic, for purposes of effect tracking:

$$\frac{\mathsf{fresh}(i)}{((\texttt{atomic}\ e, \Sigma, \emptyset), H) \rightarrow (\texttt{atomic}\ (e, \Sigma, \emptyset)_i, H)}$$ DYN-ATOMIC-EVAL

$$\frac{\emptyset \vdash \mathsf{atomic}(E) = E'}{(\texttt{atomic}\ (o, \Sigma, E)_i, H) \rightarrow ((o, \Sigma, E'), H)}$$ DYN-ATOMIC-MARK-EFFECTS

Finally, we modify the rule DYN-COBEGIN-ACCUMULATE (shown in Section 4.2) to mark the effects of a `cobegin` expression non-atomic:

$$\frac{\emptyset \vdash \mathsf{nonatomic}(E \cup E') = E''}{(\texttt{cobegin}((o, \Sigma, E)_i, (o', \Sigma, E')_j), H) \rightarrow ((o', \Sigma, E''), H)}$$ DYN-COBEGIN-ACCUMULATE

Weak Isolation Constraint: To state the weak isolation constraint, we need the concept of a *reduction history* $\mathbf{H}$, which is a sequence of program execution steps witnessing $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} (e'_i, H)$. If $((e, \Sigma, \emptyset)_i), H)$ is the initial program state, then we call $\mathbf{H}$ a *program execution history* and write $\mathbf{H}_{\mathcal{P}}$. Two histories occur *in parallel under* `cobegin` (or `cobegin_nd`) if they each occur in reducing different branches of the same `cobegin` (or `cobegin_nd`) expression, in the same program execution.

Definition 1 (Conflict relation on atomic expressions)**.** *Fix a program execution history $\mathbf{H}_{\mathcal{P}}$, and let I be the set of expression indices appearing in $\mathbf{H}_{\mathcal{P}}$ that label atomic expressions (i.e., expressions introduced by rule* DYN-ATOMIC-EVAL*). The* ***conflict relation on atomic expressions in*** *$\mathbf{H}_{\mathcal{P}}$ is the transitive closure of the following relation on $I \times I$: (i, j) is in the relation if $i \neq j$, and there are conflicting memory accesses a_i and a_j such that (a) a_i occurs in the reduction of an atomic expression e_i; (b) a_j occurs in the reduction of an atomic expression e_j; (c) the reductions of e_i and e_j occur in parallel under* `cobegin_nd`*, and (d) a_i precedes a_j in $\mathbf{H}_{\mathcal{P}}$.*

Notice that we put the relation *only on operations under* `cobegin_nd`*, not under* `cobegin`; and we *do not include any conflicts occurring outside of atomic expressions*. That is because the type system will ensure there are no conflicts between `cobegin` tasks or outside of atomic expressions; this is the soundness result that we state below.

Now we can define the weak isolation constraint on executions in the language. For the remainder of this section, we assume an implementation that guarantees weakly isolated program execution histories $\mathbf{H}_{\mathcal{P}}$.

Definition 2 (Weakly isolated histories)**.** *Let $\mathbf{H}$ be a history. If the conflict relation on atomic expressions in $\mathbf{H}$ is a partial order, then we say that $\mathbf{H}$ is* ***weakly isolated****.*

Soundness Results: As summarized in Section 4.1, there are three main soundness results for the nondeterministic language: race freedom, strong isolation, and determinism by default.

Race Freedom: The first result says that the language is race free, assuming that pairs of memory accesses in different atomic expressions are well-synchronized (which is true of any transactional implementation). The proof follows from the fact that the type system disallows parallel interference, except for pairs of accesses both occurring in atomic expressions.

Theorem 3 (Race freedom)**.** *If $\vdash \mathcal{P}$, then a history $\mathbf{H}_{\mathcal{P}}$ that has synchronization orderings consistent with the conflict relation stated in Definition 1, contains no data race.*

Strong Isolation: To state the strong isolation result formally, we use the well-known concept of *serializable histories* [44]. We say

that a history $\mathbf{H}$ witnessing $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} (e_i', H')$ is *serial with respect to expression* i if every step in the history transforms expression i or a subexpression of expression i. We say that $\mathbf{H}$ is *serializable with respect to expression* i if it is possible to generate a history $\mathbf{H}'$ with the same initial and final states as $\mathbf{H}$ that contains a serial history witnessing $((e, \Sigma, \emptyset)_i, H'') \rightsquigarrow_{\mathcal{P}} (e_i', H''')$, for some heaps H'' and H'''. In other words, an expression reduction is serializable if we "could have done it serially," with the same results.

The following theorem says that a history is serializable if it does not occur in a `cobegin_nd`; or it does not reduce any atomic expression; or it reduces a `cobegin` or atomic expression. The proof follows from the type system's guarantees of noninterference for `cobegin` tasks, and noninterference for `cobegin_nd` tasks except where guarded by atomic expressions; together with the weak isolation assumption for atomic expressions.

Theorem 4 (Strong isolation)**.** *Suppose* $\vdash \mathcal{P}$*, let* $\mathbf{H}_{\mathcal{P}}$ *be a weakly isolated history executing* $\mathcal{P}$*, and let* $\mathbf{H}$ *be a history witnessing* $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} (e_i', H')$ *contained in* $\mathbf{H}_{\mathcal{P}}$*. Then* $\mathbf{H}$ *is serializable with respect to expression* i *if (1)* $(e, \Sigma, \emptyset)_i$ *is not a subexpression of any* `cobegin_nd` *expression; or (2) no atomic expression appears in* $\mathbf{H}$*; or (3)* e *is a* `cobegin` *or atomic expression.*

Determinism by Default: Finally, we show that the nondeterministic language is deterministic by default. First we show the equivalent of Theorem 1 for the nondeterministic language: there is no difference between `cobegin` e and `atomic seq` e. Note that we need the `atomic` here because in general a `seq` occurring under a `cobegin_nd` can interfere with the other branch of the `cobegin_nd`. The result follows directly from Theorem 1 and Theorem 4. ("Initial state" is defined before Theorem 1.)

Theorem 5 (Semantic equivalence of `cobegin` and `atomic seq`)**.**

$$((\texttt{cobegin}(e, e'), \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$$

if and only if

$$((\texttt{atomic seq}(e, e'), \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$$

with the same initial state, except that expression i *is as shown.*

Second, we show the equivalent of Theorem 2 for the nondeterministic language: for the expressions called out by Theorem 4 as having serializable reductions, the execution of such expressions is also input-output deterministic, *unless* there is explicit nondeterminism via `cobegin_nd`. The proof follows from Theorem 2, together with the definition of a serializable history.

Theorem 6 (Determinism by default)**.** *Suppose* $\vdash \mathcal{P}$*, and let* $\mathbf{H}$ *be a history witnessing* $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$ *that is serializable with respect to expression* i*, where no* `cobegin_nd` *expression ever appears in expression* i*. If* $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o', \Sigma, E')_i, H'')$ *with the same initial state as in* $\mathbf{H}$*, then* $o \cong o$*, where* $\cong$ *denotes equivalence up to renaming object references, and* $E = E'$*. Moreover, if* $(e, \Sigma, \emptyset)_i$ *is not a subexpression of any* `cobegin` *or* `cobegin_nd` *expression, then* $H' \cong H''$*.*

4.4 Atomic Regions Language

Static Semantics: First, the rules for constructing types require that atomic regions bind only to region parameters declared `atomic`, and non-atomic regions bind only to region parameters not declared `atomic`. This requirement ensures that memory regions are treated consistently across method invocation. Second, we refine the judgment $\Gamma \vdash \mathsf{atomic}(E) = E'$ so that an atomic expression "makes an effect atomic" only if the effect is on an atomic region. This rule ensures that, in applying rule COBEGIN-ND to check a `cobegin_nd` expression, the compiler will never allow effects on non-atomic regions to interfere. The rules are stated in full in the thesis.

Dynamic Semantics and Soundness: The dynamic semantics of this language is exactly as given in the previous section, with two changes. First, marking effects in rule DYN-ATOMIC-MARK-EFFECTS happens according to the refined definition of $\emptyset \vdash \mathsf{atomic}(E) = E'$: that is, effects are marked atomic only if they operate on atomic regions. Second, we redefine the conflict relation on atomic expressions (Definition 1) so that only conflicts involving accesses to statically identifiable atomic regions (i.e., region names r or region parameters ρ marked `atomic`) are synchronized by the implementation. The soundness result says that Theorems 3–6 hold for this language variant; the proofs, given in the thesis, are entirely straightforward.

5. Prototype Implementation

To implement our new mechanisms, we extended the compiler used for basic DPJ. It is a modified version of Sun's `javac`, which translates DPJ code to standard Java. The compiler implements DPJ's parallelism constructs by generating calls to the `ForkJoinTask` library [2], which schedules tasks onto a pool of worker threads; further details are available in [17].

In this work, we extended the compiler to implement atomic blocks using the Deuce STM library [3]. We used the well-respected Transactional Locking II (TL2) algorithm [28]. TL2 is a write-buffering (i.e., lazy versioning) algorithm with optimistic reads. Deuce supports concurrency control at the object field level, and uses a light-weight, custom reflection mechanism to access object fields inside transactions.

We selected this STM system for pragmatic reasons of ease of implementation, and because it implements a well-known high-performance STM algorithm. We have not attempted to maximize absolute performance in our implementation; it could be improved significantly by using a different STM system, such as one integrated with the JVM [46]. Our method is applicable to other types of STM systems and algorithms (including those utilizing in-place updates).

For each atomic block, the compiler generates code to execute the body of the atomic block as a transaction, retrying until the transaction commits successfully. Nested atomic blocks are flattened. Methods that are transitively callable within atomic blocks are cloned; versions containing barriers are used when they are called within atomic blocks. Within atomic blocks, the compiler inserts normal read and write barriers for accesses to fields in atomic regions. As discussed in section 3.4, the compiler omits barriers for read accesses to non-atomic regions, and it generates logging-only barriers for write accesses.

We modified the TL2 implementation in Deuce to support these optimized logging-only barriers. However, because TL2 is a write-buffering algorithm, we would have to use read-barriers to obtain correct values in the read-after-write cases. To avoid read barriers entirely, we modified the algorithm to perform in-place updates for these locations and we maintain a separate undo log to revert effects of such updates in case the transaction aborts. Reads to such locations do not need barriers because they can now obtain their values directly from the original memory location.

6. Evaluation

The ideas presented in this paper raise four key questions for experimental investigation: (1) Can the language express nondeterministic algorithms in a natural way? (2) Can the algorithms expressed in the language give good performance? (3) How effective is the optimization of STM barriers? (4) What is the annotation overhead of the language?

We used four nondeterministic algorithms to evaluate these questions: two different versions of TSP, Delaunay mesh triangulation from the *Lonestar Benchmarks* [1], and *OO7*, a synthetic database benchmark that has been used in previous studies of parallel performance [47, 51]. These codes are discussed further below.

6.1 Benchmarks and Expressing Parallelism

Traveling Salesman Problem: We studied two versions of the TSP algorithm, which we call **TSP-PQ** and **TSP-R**. TSP-PQ is the algorithm described in Section 3. As discussed there, the algorithm proceeds in two phases: the first phase breaks the problem up into subproblems and adds them to a priority queue, and the second phase concurrently removes items from the queue and processes each one using sequential recursive search. The priority queue orders the work, so that more promising subtrees are explored first.

TSP-R is a variant that eliminates the priority queue and uses recursion to express the entire algorithm. At each level of the tree, the algorithm computes a bound for each subtree and compares the bound against the global current best tour. Bounds that are definitely no better than the current best are excluded, while bounds that may be better are explored recursively. The recursion occurs in parallel until a specified depth of the tree; in our studies we used a depth varying with the log of the number of threads. TSP-R is a simpler algorithm than TSP-PQ, but it potentially suffers from more contention, as the global best tour must be read before every recursive descent into a subtree to avoid exploring too many bad paths. By contrast, because TSP-PQ uses a priority queue to order the paths, it can read the global best tour less often (once per tree level).

We adapted both versions of TSP from code that was used in previous studies of STM performance [46, 47]. Our TSP-PQ code uses the identical algorithm to the original code, and expresses the parallelism in the same way. The original code had a data race, and we added one extra atomic block to eliminate that race. Our TSP-R code is a transformation of TSP-PQ that eliminates the priority queue, checks the bound at each level of the tree, and parallelizes the recursion.

Delaunay Mesh Refinement: This code uses Chew's algorithm [26, 36] to find and eliminate "bad triangles," i.e., those that do not satisfy some quality constraint from a Delaunay triangulation of a mesh of points. The program is nondeterministic since different orders of processing of bad elements lead to different meshes, although all such meshes satisfy the quality constraints [26]. The program uses a `foreach_nd` loop, and each iteration of the loop spawns a new worker thread (at most one per core). Each worker thread has a private worklist of bad triangles. In each iteration of the worklist loop, the worker selects one bad triangle from the work list, forms a *cavity* around it, re-triangulates the cavity, and adds any new bad triangles back to the worklist. Cavity finding and re-triangulating code sections access the shared mesh data structure and are enclosed in atomic blocks.

OO7: OO7 simulates a number of clients, each performing a fixed number of queries on an in-memory database. Each query is enclosed in an atomic block. The performance metric is the throughput (queries per unit time), and we measure how this scales by varying the number of clients while keeping the number of queries performed by each one constant. The program uses a `foreach_nd` loop, with one iteration corresponding to each client. We configured it to use a number of clients equal to the number of worker threads, so there is always one thread per client. Thus, the total amount of work performed is proportional to the number of threads.

Expressing Parallelism: We successfully expressed *all* the parallelism that did not use data races, in these four nondeterministic algorithms. As discussed above, we eliminated a race in TSP-PQ that was presumably there to avoid synchronization; we could have also written TSP-R with a similar race. The four codes do not use any deterministic algorithms but such algorithms do not incur any runtime performance overheads in our language; such overheads are dominated by that of atomic sections in nondeterministic components. The performance and expressivity of the language for deterministic algorithms were studied previously [17].

6.2 Performance

To evaluate performance, we measured the self-relative speedup (i.e., the speedup compared to running the transactional code on one thread) achieved by the three codes. We focused on self-relative speedup rather than absolute speedup because (a) optimizing the code generation for atomic statements has not been a focus of this paper, and (b) the Deuce STM, although using a good *algorithm*, lacks many many essential performance features of a high performance Java STM [46]. Self-relative speedups have the effect of "factoring out" some of the performance impact of the STM implementation while capturing the scalability of the benchmarks.

We ran and measured the codes on a 24-core system using four Intel Xeon E7450 processors (each with six cores), running Windows Server 2008. Figure 10 shows the self-relative speedups with barrier optimizations enabled, using running times for Delaunay and TSP, and throughput scaling for OO7. Because the runtimes are nondeterministic, we averaged 5–10 runs for each data point, using an interquartile method to exclude a few extreme outliers. For both TSP variants, we used the one-thread version of TSP-PQ, which was the faster of the two, as the baseline. Both versions of TSP show good scaling, and OO7 shows moderately good scaling, throughout the range of numbers of threads t we examined. TSP-R shows better (superlinear) speedup for smaller t; this is because the parallel algorithm is very efficient in that range: it rules out subtrees quickly, and so visits only about $1/4$ of the tree nodes at $t = 2$ compared to $t = 1$. However, the scaling curve for TSP-R flattens out as t increases, most likely due to higher contention than TSP-PQ.

The speedup curve for Delaunay is poor: it flattens out and reaches only 3x on 22 threads. We profiled the code to understand the source of this behavior and traced it to the method `System.identityHashcode()` in the JVM. This standard Java function is extensively used in Deuce to index into lock tables. We observed that the time spent in this function grows with the number of threads. In Delaunay, which has large transactions, this overhead negatively affected the speedup curve. This problem can by solved by modifying the JVM, but we leave that (and other optimizations for `atomic`) to future work.

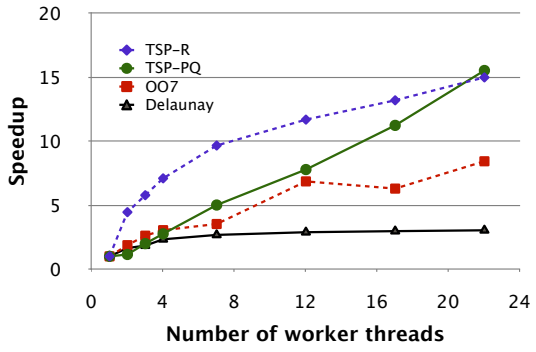

Figure 10. Self-relative speedups. For OO7, we scaled the amount of work with the number of worker threads, and measured speedup based on throughput scaling (number of queries done per unit time). The barrier optimization was enabled for all of these benchmarks.

6.3 Impact of Barrier Elimination

We compared the performance of two versions of the parallel code for each benchmark: with and without the barrier simplification optimization for non-atomic regions. Figure 11 shows the improvement in running time for the optimized code compared to the unoptimized code. Figure 12 shows the reduction in the number of dynamically-executed barriers due to our optimizations.

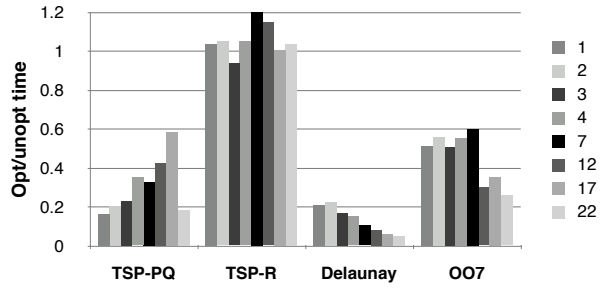

Figure 11. Ratio of optimized runtimes (with barrier elimination) to unoptimized runtimes (without barrier elimination). A value lower than 1 means the optimization increased performance.

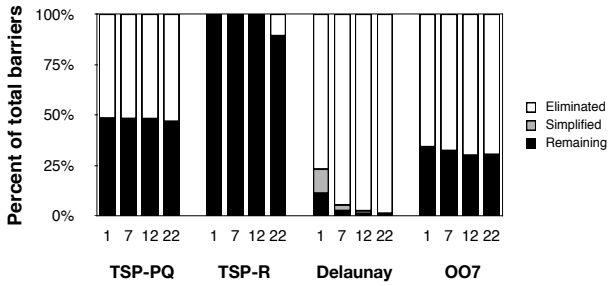

Figure 12. Reduction in barriers due to optimizations, showing the proportion of barriers from the unoptimized version that are eliminated entirely, simplified to log-only write barriers, or that remain as full barriers in the optimized version, for each of the three benchmarks with 1, 7, 12, and 22 worker threads.

The optimization has a substantial impact on performance for three of the four benchmarks (TSP-PQ, Delaunay, and OO7). The performance improvements correlate well with the barrier reductions. The optimizations give essentially no improvement for TSP-R, because the transactions are very short (reads and read-modify-write operations on the best tour). As a result (1) there are few if any barriers to remove; and (2) transactional overhead is not a significant component of the overall runtime. On the other hand, TSP-PQ, OO7, and Delaunay use longer transactions, providing more opportunities for reducing overhead.

Our optimizations can eliminate barriers both by actually removing barrier operations on certain statements and also by reducing the number of times that transactions must be retried. The latter effect occurs because removing unnecessary barriers reduces the number of false conflicts incurred by the STM system. As shown in Table 1, this effect is more pronounced with larger numbers of worker threads, so our optimizations not only reduce scalar overheads but also improve scalability. For example, in Delaunay, the optimization changed this ratio from 0.944 to 0.999 on 2 threads but from 0.244 to 0.944 on 22 threads.

	Delaunay		OO7	
threads	opt	unopt	opt	unopt
2	0.999	0.944	0.944	0.932
3	0.975	0.848	0.877	0.872
4	0.998	0.810	0.822	0.560
7	0.993	0.647	0.700	0.210
12	0.996	0.405	0.539	0.100
17	0.995	0.291	0.442	0.071
22	0.994	0.244	0.369	0.071

Table 1. Ratio of committed transactions to started transactions for Delaunay and OO7. Lower numbers indicate more aborted transactions. For both versions of TSP, all numbers are 1.000.

Program	Total SLOC	Annotated SLOC	Region Decls	RPLs	Params	Effect Summ.
TSP-PQ	433	77 (17.8%)	2(1)	101(4)	6(2)	14/20
TSP-R	200	34 (17%)	2(1)	42(4)	2(0)	6/12
OO7	1570	105 (6.7%)	4(1)	76(7)	6(0)	52/104
Delaunay	1994	302 (15.1%)	3(1)	374(3)	21(7)	165/216
Total	4197	518 (12.3%)	11(4)	593(18)	35(9)	237/352

Table 2. Annotation counts for the four benchmarks. In the middle columns, the numbers in parentheses represent the number of annotations marked `atomic`. In the last column, x/y means of y total method definitions in the program, x were annotated with effect summaries.

6.4 Annotation Overhead

Table 2 provides a quantitative measure of the annotation overhead of writing the four benchmarks in our language. Column 1 after the vertical bar shows the total number of non-blank, non-comment lines of source code, counted by `sloccount`. Column 2 gives the count of annotated lines, as an absolute number and as a percentage of the total lines. The following three columns show the number of region declarations, RPLs (including arguments to `in`, arguments to types and methods, and arguments to effect summaries), and region parameters. The number of annotations marked `atomic` is shown in parentheses after the main number. The last column shows the number of effect summaries before the slash, and the number of method definitions after the slash.

While the average number of annotated lines (12.3%) is nontrivial, we believe it is not unduly high, given the strong safety properties of the programming model. As in our prior work [17], most of the RPL annotations were arguments to types. The overhead could be reduced by inferring some of the annotations [48], but we leave that for future work.

Our approach does impose the limitation that if a programmer wishes to use a class region parameter as an atomic region in some context and a non-atomic region in some other context, then the class must be *cloned*: the programmer must create two copies of the class, one with the atomic parameter and one with the non-atomic parameter. The cloning is required because different barriers must be generated for methods of the class that operate transactionally on the parameter, depending on whether the region bound to the parameter is atomic. The cloning could be done automatically by the compiler, similarly to what C++ does for templates. While we have not implemented this approach, we believe it does not raise any significant technical issues.

In the benchmarks we studied, only Delaunay required class cloning. In Delaunay, we needed both atomic and non-atomic versions of the list and map structures used in the benchmark.

7. Related Work

Type and Effect Systems: Several researchers have described effect systems for enforcing a locking discipline in nondeterministic programs that prevents data races and deadlocks [5, 20, 34] or guarantees isolation for critical sections [29]. Matsakis et al. [41] have recently proposed a type system that guarantees race-freedom for locks and other synchronization constructs using a construct called an "interval" for expressing parallelism. While there is some overlap with our work in the guarantees provided (race freedom, deadlock freedom, and isolation), the mechanisms are very different (explicit synchronization vs. atomic statements supported by STM). Further, these systems do not provide determinism by default. Finally, there is no other effect system we know of that provides both race freedom *and* strong isolation together.

STM Correctness (Language): STM Haskell [31] provides an isolation guarantee, but for a pure functional language that uses monads to limit effects to the transactional store, unlike our imperative shared-memory language. Moore and Grossman [42] and Abadi et al. [6] use types and effects to guarantee strong isolation for an imperative language, but their languages permit races where neither access occurs in a transaction. Finally, none of these languages allows both transactional and non-transactional effects to the same memory, as our language does.

Beckman et al. [12] show how to use a form of alias control called *access permissions* [21] to verify that the placement of atomic blocks in a threaded program respects the invariants of a specification written by the programmer — for example, that a condition is checked and acted upon atomically. This approach is complementary to ours: we provide guarantees of race freedom, strong isolation, and determinism by default for all programs in our language; on top of that one could check that additional programmer-specified invariants are satisfied.

STM Correctness (Compiler and Runtime): Several STMs guarantee strong isolation by preventing interference between transactions and non-transactional accesses at runtime. Most of these systems use a combination of sophisticated static whole-program analysis, runtime optimizations, and other runtime techniques like page protection to optimize strong isolation [7, 22, 46, 47]. While these techniques can significantly reduce the cost of strong isolation, they cannot completely eliminate it. In contrast, our language-based approach provides strong isolation without imposing extra runtime overhead.

Reducing STM Overheads: Much research has been devoted to reducing the cost of compiler-generated STM barriers on transactional memory accesses. Early work [8, 32] showed how to eliminate several classes of transactional overhead including redundant barriers, barriers for accesses to provably immutable memory locations, and certain barriers for accesses to objects allocated in a transaction. Recent work by Afek et al. [9] uses the logic of program reads and writes within a transaction to reduce STM overhead: for example, a shared variable that is read several times can be be read once and cached locally. These optimizations complement ours, as they target different kinds of STM overhead from our work.

Beckman et al. [13] show how to use access permissions to remove STM synchronization overhead. While the goals are the same as ours, the mechanisms are different (alias control vs. type and effect annotations). The two mechanisms have different tradeoffs in expressivity and power: for example, Beckman et al.'s method can eliminate write barriers only if an object is accessed through a unique reference, whereas our system can eliminate barriers for access through shared references, so long as the access does not cause interfering effects. However, alias restrictions can express some patterns (such as permuting unique references in a data structure) that our system cannot. As future work, it would be interesting to explore these tradeoffs further.

Finally, several researchers have eliminated STM overhead for accesses to thread-local data using whole-program static escape analysis [47] and programmer annotations to specify code blocks that do not require instrumentation [52]. Unlike our work, this work either requires whole-program analysis, or it relies on unverified programmer annotations.

Nondeterministic Parallel Programming: Several research efforts are developing parallel models for nondeterministic codes with irregular data access patterns, such as Delaunay mesh refinement. Galois [36] provides a form of isolation, but with iterations of parallel loops (instead of atomic statements) as the isolated computations. Concurrency is increased by detecting conflicts at the level of method calls, instead of reads and writes, and using semantic commutativity properties. Lublinerman et al. [39] have proposed *object assemblies* as an alternative model for expressing irregular, graph-based computations.

These models are largely orthogonal to our work. In Galois, strong isolation holds if all shared data is accessed through well-defined APIs, but this property is not enforced, either statically or at runtime. We believe that our type and effect mechanisms could be applied to Galois to ensure this property. The object assemblies model may have stronger isolation guarantees than Galois, but it is very specialized to irregular graph computations, in contrast to the more general fork-join model we present here.

Kulkarni et al. [35] have recently proposed *task types* as a way of enforcing a property they call *pervasive atomicity*. This work shares with ours the broad goal of reducing the number of concurrent interleavings the programmer must consider. However, Kulkarni et al. adopt an actor-inspired approach, in which data is non-shared by default, and sharing musk occur through special "task objects." This is in contrast to our approach of allowing familiar shared-memory patterns of programming, but using effect annotations to enforce safety properties. Finally, none of the work discussed above provides any deterministic-by-default guarantee.

8. Conclusion

We have shown how to design a type and effect system that, together with a weakly atomic runtime system, achieves our stated goals of providing disciplined and safe nondeterminism, including race freedom, strong isolation of atomic operations and deterministic parallel operations, compositional reasoning about deterministic and nondeterministic operations, and determinism by default. We have also shown how to leverage the system to remove unnecessary barriers from the transactional implementation, thereby enhancing performance.

Acknowledgements

This work was supported by the National Science Foundation under grants CCF 07-02724 and CNS 07-20772, and by Intel, Microsoft, and the University of Illinois through UPCRC Illinois. An anonymous reviewer encouraged us to study TSP-R. Dan Grossman and Brad Chamberlain provided helpful suggestions on a draft of this paper and the supporting proofs.

References

[1] `http://iss.ices.utexas.edu/lonestar/`.

[2] `http://gee.cs.oswego.edu/dl/concurrency-interest`.

[3] `http://http://sites.google.com/site/deucestm`.

[4] OpenMP Application Program Interface, Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf, 2008.

[5] M. Abadi et al. Types for safe locking: Static race detection for Java. *TOPLAS*, 2006.

[6] M. Abadi et al. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

[7] M. Abadi et al. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP*, 2009.

[8] A.-R. Adl-Tabatabai et al. Compiler and runtime support for efficient software transactional memory. In *PLDI*, 2006.

[9] Y. Afek et al. Lowering STM overhead with static analysis. In *LCPC*, 2010.

[10] M. D. Allen et al. Serialization sets: A dynamic dependence-based parallel execution model. In *PPOPP*, 2009.

[11] A. Aviram et al. Efficient system-enforced deterministic parallelism. 2010.

[12] N. E. Beckman et al. Verifying correct usage of atomic blocks and typestate. In *OOPSLA*, 2008.

[13] N. E. Beckman et al. Reducing STM overhead with access permissions. In *IWACO*, 2009.

[14] T. Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Int'l. Conf. on Arch. Support for Programming Langs. and Operating Systs. (ASPLOS)*, 2010.

[15] E. D. Berger et al. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, 2009.

[16] R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. *PPOPP*, 1995.

[17] R. L. Bocchino et al. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.

[18] R. L. Bocchino et al. Parallel programming must be deterministic by default. In *HotPar*, 2009.

[19] R. L. Bocchino Jr. *An Effect System and Language for Deterministic-by-Default Parallel Programming*. PhD thesis, University of Illinois, Urbana-Champaign, IL, 2010.

[20] C. Boyapati et al. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[21] J. Boyland. Checking interference with fractional permissions. *SAS*, 2003.

[22] N. G. Bronson et al. Feedback-directed barrier optimization in a strongly isolated STM. In *POPL*, 2009.

[23] Z. Budimlic et al. Multicore implementations of the concurrent collections programming model. In *CPC*, 2009.

[24] S. Burckhardt et al. Concurrent programming with revisions and isolation types. In *OOPSLA*, 2010.

[25] M. J. Carey et al. A status report on the OO7 OODBMS benchmarking effort. In *OOPSLA*, 1994.

[26] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *SCG*, 1993.

[27] J. Devietti et al. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[28] D. Dice et al. Transactional locking II. In *DISC*, 2006.

[29] C. Flanagan et al. Types for atomicity: Static checking and inference for Java. *TOPLAS*, 2008.

[30] A. Ghuloum et al. Ct: A flexible parallel programming model for tera-scale architectures. Intel White Paper, 2007.

[31] T. Harris et al. Composable memory transactions. In *PPoPP*, 2005.

[32] T. Harris et al. Optimizing memory transactions. In *PLDI*, 2006.

[33] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[34] B. Jacobs et al. A programming model for concurrent object-oriented programs. *TOPLAS*, 2008.

[35] A. Kulkarni et al. Task types for pervasive atomicity. In *OOPSLA*, 2010.

[36] M. Kulkarni et al. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[37] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[38] E. A. Lee. The problem with threads. *Computer*, 2006.

[39] R. Lublinerman et al. Parallel programming with object assemblies. In *OOPSLA*, 2009.

[40] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comp. Arch. Letters*, 5(2):17, 2006.

[41] N. D. Matsakis and T. R. Gross. A time-aware type system for data-race protection and guaranteed initialization. In *OOPSLA*, 2010.

[42] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, 2008.

[43] M. Olszewski et al. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

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

[45] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS*, 1998.

[46] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA*, 2008.

[47] T. Shpeisman et al. Enforcing isolation and ordering in STM. In *PLDI*, 2007.

[48] M. Vakilian et al. Inferring Method Effect Summaries for Deterministic Parallel Java. Technical Report UIUCDCS-R-2009-3038, U. Illinois, 2009.

[49] C. von Praun et al. Implicit parallelism with ordered transactions. In *PPOPP*, 2007.

[50] A. Welc et al. Safe futures for Java. In *OOPSLA*, 2005.

[51] A. Welc et al. Revocation techniques for Java concurrency. *Concurrency and Computation: Practice and Experience*, 2006.

[52] R. M. Yoo et al. Kicking the tires of software transactional memory: Why the going gets tough. In *SPAA*, 2008.