Abstract

Nondeterminism is a useful and prevalent concept in the design and implementation of software systems. An important property of nondeterminism is its latent parallelism: A nondeterministic action can evaluate to multiple behaviors. If at least one of these behaviors does not conflict with concurrent tasks, then there is an admissible execution of the action in parallel with these tasks. Unfortunately, existing implementations of the atomic paradigm—optimistic as well as pessimistic—are unable to fully exhaust the parallelism potential of nondeterministic actions, lacking the means to guide concurrent tasks toward nondeterministic choices that minimize interference.

This paper investigates the problem of utilizing parallelism due to nondeterminism. We observe that nondeterminism occurs in many real-world codes. We motivate the need for devising coordination mechanisms that can utilize available nondeterminism. We have developed a system featuring such mechanisms, which leverages nondeterminism in a wide class of query operations, allowing a task to look into the future of concurrent tasks that mutate the shared state during query evaluation and reduce conflict accordingly. We evaluate our system on a suite of 12 algorithmic benchmarks of wide applicability, as well as an industrial application. The results are encouraging.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Algorithms, Measurement, Performance

Keywords nondeterminism, parallelism, concurrency control, synchronization, serializability

*The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no [321774-VSSC]

1. Introduction

Nondeterminism arises in the sequential specification of many algorithms and software applications. Examples include graph algorithms, such as obtaining some minimum spanning tree (MST) from an input graph or finding some shortest path between two nodes; search techniques, whose objective is to arrive at some admissible goal state (as in the n-queens and combinatorial-assignment problems [18]); learning strategies, such as converging on some classifier that labels all data instances correctly [54]; and more generally, any algorithm for finding a minimal or maximal subset of a collection of values which satisfies some property, or finding a representative value satisfying the property.

The connection to parallelism is natural. Correct parallel execution is governed by the requirement for atomicity, whereby concurrent tasks in the parallel trace should appear as if they executed uninterleaved, and thus atomically. The trace is then serializable, in that it is equivalent to a serial trace involving the same tasks [30]. To ensure atomicity, conflict between tasks (e.g. due to read/write or write/write accesses to the same memory) is standardly resolved by serializing their execution, thereby reducing parallelism.

Nondeterminism presents an opportunity to reduce conflict, thereby increasing parallelism: When there are multiple valid executions of a given task, arising from nondeterminism, one might coerce that task to choose the executions that have the least amount of conflict with concurrent tasks (e.g. those whose memory accesses are disjoint from the memory accessed by concurrent tasks).

The following simple example, in Java syntax, illustrates this observation:

```java
/* global */ Set elems = ...;

Task 1: @atomic {
    Iterator it=elems.iterator();
    Object o1=it().next();
    // assumes: o1 ∈ elems
    ...
}

Task 2: @atomic {
    Object o2 = ...;
    elems.remove(o2);
    // assumes: o2 ∉ elems
    ...
}
```

Task 1 above has latent nondeterminism: The Java specification for iterating over a Set object places no guarantees on iteration order. This permits task 1 to choose an object that
differs from (the object pointed-to by) $o_2$, thereby avoiding conflict with task 2.

Interestingly, in spite of the strong connection between nondeterminism and parallelism, we are not aware of proposals to exploit nondeterminism as a source of available parallelism. The only research we are aware of regarding the connection between nondeterminism and parallelism is restricted to the specification level [10, 11], where the idea is to reduce the correctness of a parallel program to checking whether every result it produces is also possible under a nondeterministic sequential version of the program.

Current implementations of the atomic construct are unable to exhaust the parallelism that inheres in nondeterministic codes. Existing techniques do not have the means to guide nondeterministic tasks toward executions that would maximize parallelism. Optimistic synchronization approaches—most notably, transactional memory (TM) [30]—handle potential conflicts through the history of the execution—what has occurred thus far—where utilizing nondeterminism requires foresight. In the above example, a standard optimistic protocol would let task 1 make a free choice which object to read from $e_{\text{elems}}$, aborting due to a read/write conflict if that object is $o_2$. The conflict is semantic, and thus abstract-level synchronization [29, 37, 44], which was recently proposed as a means of reducing conflict, would still force an abort.

Meanwhile, techniques for pessimistic synchronization (i.e. without the ability to undo operations on the shared state), including recent proposals featuring user interaction and client-driven lock inference [12, 39, 42], make conservative forecasts about the future behavior of concurrent tasks for lack of appropriate communication mechanisms, which disables the available parallelism due to nondeterminism. Because task 1 may read any object stored in $e_{\text{elems}}$ (and in particular $o_2$), such synchronization would enforce ownership of the entire $e_{\text{elems}}$ set by task 1. This would effectively serialize the execution of the two tasks.

Existing solutions do not utilize the observation that the tasks can coordinate access to different objects, thereby avoiding conflict. Acting upon this observation requires a communication mechanism that lets tasks broadcast their intentions (i.e. may-modify information), such that other tasks can specialize their nondeterministic choices. For example, if task 1 is aware of task 2’s intention to remove $o_2$ from $e_{\text{elems}}$, then it can read another object.

**Scope** We investigate the problem of exploiting parallelism resulting from nondeterminism in the presence of mutation operations, where synchronization is required. We have found that the key is a concurrency paradigm in which conflict is based on the future, rather than the past. In our model, tasks share information about the behaviors they may exhibit in the future, and then specialize themselves (i.e. choose from available nondeterministic options) to reduce conflict. To our knowledge, this is the first instance of a general model for extracting parallelism from available nondeterminism.

In practice, this is achieved by letting each task view only the portion of the shared state that is invariant over the effects of concurrent tasks, and make its choices based on this partial view. In the example above, for instance, the available view for task 1 does not contain (the object pointed-to by) $o_2$. Later, in Sections 3–4, we discuss the assumptions governing this synchronization approach, as well as the conditions under which progress is guaranteed.

We have realized our approach in TANGO, a general synchronization protocol implemented as a software tool for loop parallelization in both Java and C#. TANGO enforces a pessimistic synchronization scheme guaranteeing deadlock-free, serializable parallel execution. TANGO exploits nondeterminism in pure monotonic query operations, such as testing an element for membership in a container or checking for connectivity between graph nodes. This is achieved by specializing the query, such that it evaluates on an invariant portion of the shared state.

For efficient runtime query specialization, TANGO performs compile-time analysis of task intentions (e.g. the intention of task 2 to remove $o_2$). As is the case with other approaches [5], TANGO is best suited for the broad class of applications where task intentions are “predictable” and constrained to a small portion of the shared state [22]. This lets concurrent operations view and act on most of the shared state already at an early point, enabling a high level of parallelism.

**TANGO** comes with a library of popular abstract data type (ADT) implementations, including Set, Map and Graph, and requires little annotation overhead. As we detail in Section 6, we were able to apply TANGO to challenging loops within a commercial application with relative ease, and obtained speedups of up to 3.2x. We also report on experiments over a suite of 12 loop-based algorithms that are considered challenging to parallelize due to ordering constraints on loop iterations as well as global-scope queries. The results are encouraging, showing TANGO to be comparable in performance to parallelization schemes that were tailored specifically for some of our benchmarks.

**Contributions** This paper makes the following principal contributions:

**Nondeterminism and Specialization** We take a first step in developing a concurrent programming paradigm that turns latent nondeterminism into a source of parallelism. In our approach, conflict is based on potential future behaviors, and threads make nondeterministic choices that avoid conflict (Section 4). In particular, threads may specialize both their view of the shared state (e.g. reads) and their modifications to the shared state (e.g. writes).

**Synchronization Protocol** We have developed a general rollback-free synchronization protocol that enforces par-
var sortedEdges = edges.OrderBy(x => -x.Weight);

foreach (var edge in sortedEdges) {
    edges.Remove(edge);
    if (!FindPath(edge.U, edge.V)) {
        edges.Insert(edge);
    }
}

Figure 1. The REVERSE DELETE algorithm in LINQ syntax, annotated for parallelization under TANGO

parallelism due to pure monotonic queries with available nondeterminism (Section 5). Our protocol generalizes several recent parallelization schemes that were tailored for specific applications, and achieves comparable performance on these applications. Underlying our protocol is the notion of specialization: a novel synchronization primitive that lets a task determine its behavior to reduce conflict.

Greedy Algorithms We describe experiments on a suite of 12 benchmarks, 8 of which are greedy algorithms (Section 6.1). These experiments, beyond measuring the performance of our protocol, enable understanding of the available parallelism in greedy algorithms according to the way in which a greedy step depends on the previous steps.

Implementation and Evaluation We have implemented our protocol as a tool for loop parallelization, available both in Java and in .NET. We present experiments over the tool (Section 6) that consists of (i) comparing it with several other synchronization techniques on the 12 algorithmic benchmarks (Section 6.1), and (ii) applying it to a major component in a commercial product (Section 6.2). The results of both experiments are encouraging.

2. Running Example

We illustrate our approach with the REVERSE DELETE algorithm, whose description—using LINQ syntax—appears in Figure 1. (For now, ignore the Monotonic attribute.) REVERSE DELETE computes a minimum spanning tree (MST) from a connected, edge-weighted graph. It does so in a greedy style. At each step, the algorithm considers the next heaviest edge, attempting to find a path between its incident nodes assuming the edge is discarded. If the search fails, then the speculative edge removal is reverted seeing that the edge is essential to preserve connectivity.

This program is difficult to parallelize for at least two reasons. First, it imposes ordering constraints: The input edges are traversed according to a total order (from the heaviest edge to the lightest one). Second, side effects on the edge set cause loop-carried dependencies that must be preserved. If the $i$-th iteration deletes its designated edge, then iteration $i + 1$ must account for this deletion when searching for a path between its corresponding nodes.

Figure 2. Illustrative (weighted, undirected) input graph for the REVERSE DELETE program listed in Figure 1 (a); a view of this graph excluding edges $(n_1, n_2)$ and $(n_1, n_4)$ (which is also its MST) (b); and another view where only the two lightest edges are kept (c)

However, closer inspection of the semantics of this program suggests that there is a lot of available parallelism on certain classes of input graphs. The FindPath call is nondeterministic: Any path returned by this call can serve as a witness for the deletion of the corresponding edge, provided that the edges comprising this path are not deleted by earlier iterations. Thus, the existence of multiple (nontrivial) paths between a pair of connected nodes is a source of parallelism, allowing several loop iterations to run in parallel if each has available a witness path that is invariant over edge deletions by other iterations.

To illustrate this, we refer to graph (a) in Figure 2. In this graph, the deletion of edge $(n_1, n_2)$ can be executed in parallel to deleting $(n_1, n_4)$ if the supporting path for the deletion of $(n_1, n_2)$ is $[n_1, n_3, n_4, n_5, n_2]$. Path $[n_1, n_4, n_5, n_2]$ cannot, however, support the deletion of $(n_1, n_2)$, because deleting edge $(n_1, n_4)$ invalidates this path.

3. Our Approach

In this section, we describe and illustrate the main features of our approach, and discuss its scope and limitations.

3.1 Exploiting Parallelism due to Nondeterminism

We take a first step in developing synchronization methods that are aware of latent nondeterminism and exploit it to increase parallelism. We have identified a prevalent loop coding pattern, appearing e.g. in greedy algorithms, whereby each iteration tests the program state for some property, and then—based on the result—decides which mutation operation to apply (if at all). In REVERSE DELETE, for example, the iteration checks for connectivity between a pair of previously connected nodes. If the answer is negative, then the nodes’ incident edge is restored into the graph. In certain cases, such as the FindPath query, the tested property has available nondeterminism.

We have designed a general protocol, TANGO, that realizes parallelism due to nondeterminism through specialization. Intuitively, specialization is the idea of restricting
the choices of a nondeterministic command, so that it has less interference with concurrent tasks (e.g. by accessing less shared resources), thereby enabling more parallelism.

Specifically, TANGO implements the notion of specialization over nondeterministic commands that are also monotonic. Indeed, many nondeterministic queries are also monotonic, in that successful evaluation of the query on a substate implies that the query holds over the entire state. This is the case with REVERSE DELETE, for instance, where if the FindPath test succeeds over graph $G$, and graph $G'$ contains $G$, then FindPath is guaranteed to succeed also over $G'$.

Roughly speaking, monotonicity permits serializable evaluation of a command on a substate. Focusing the command on a substate, rather than the entire state, restricts its (nondeterministic) behavior, thereby achieving specialization. TANGO’s choice of which substate to evaluate the task on is guided by the effects of concurrent tasks. That substate is the portion of the shared state that is disjoint from the effects of other tasks. In REVERSE DELETE, for example, restricting the path test to a subgraph that does not include an edge $e$ does not interfere with the deletion of $e$.

Projecting this discussion onto the example of graph (a) in Figure 2, we obtain correct parallel execution of the first two iterations, with respective edges $(n_1, n_4)$ and $(n_1, n_2)$, if the second iteration performs connectivity testing over a view of the graph that is disjoint from the possible effects of the first iteration; namely, without edge $(n_1, n_4)$. The resulting view is shown as graph (b) in Figure 2 (which coincides with the MST of the graph in (a)). Thanks to monotonicity, the success of the test, with witness path $[n_1, n_3, n_4, n_5, n_2]$, is serializable.

### 3.2 Preliminaries: Guards, Methods and Relations

The coding idiom motivating our concrete protocol, where each loop iteration performs a test and then acts on its result, encourages language-level abstractions that separate tests from actions. Dijkstra has formulated this distinction in his language of Guarded Commands in terms of guards (being boolean queries) and methods (being state transformers) [17], which we adopt also for our setting. As an illustration, in Figure 1 we denote the guard with dotted underline, and the methods with solid underline. We assume that guards are pure (i.e. free of side effects).

In our technical description, as well as our implementation, we instantiate our conceptual approach into a relational setting, whereby the shared state between loop iterations is represented and manipulated as relations. This style of data organization is familiar from databases, and has been utilized by several recent works [26, 27, 51].

The key advantage of the relational form is in abstracting data from its representation, which permits a clear, high-level description of what guards and methods do. This is also the rationale behind recent language features, such as the LINQ queries in C#. In fact, our prototype implementation is based on the LINQ language constructs, which offer a convenient means of expressing guards.

The relational abstraction consists of tuples and relations. A tuple $t = \{c_1 : v_1, c_2 : v_2, \ldots \}$ maps a set of columns, $\{c_1, c_2, \ldots \}$, to values. A relation is a set of tuples over identical columns. We support the standard relational operations, which we describe below in ML-like syntax (where $\{r\}$ fetches the current value of $r$, and $r \leftarrow v$ denotes assignment):

- empty $r = r \leftarrow \emptyset$
- insert $r t = r \leftarrow \{t\}$
- remove $r t = r \leftarrow \{t\} \setminus \{t\}$
- query $r b = b \leftarrow \phi(r)$

Informally, empty $r$ creates a fresh relation $r$; insert $r t$ inserts tuple $t$ into relation $r$; remove $r t$ removes tuple $t$ from $r$; and query $r b$ tests relation $r$ for property $\phi$ and stores the (boolean) result in $b$. For simplicity, we leave the syntax for $\phi$ unspecified. This level of detail is not required for our technical description, and our prototype tool features the built-in LINQ querying constructs.

Revisiting our running example in Figure 1, we can express the shared state as the edges binary relation. Remove and Insert are modeled directly by their relational counterparts, remove and insert, and FindPath binary relation $\text{FindPath}(u, v)$ is translated into the query $u, v \in \text{tc}((u, v), \text{edges})$, where $\text{tc}$ denotes transitive closure, and the entire query checks whether tuple $(u, v)$ is in the transitive closure of the tuples $t$ such that $\text{fst}(t) = u$ over the edges relation.

### 3.3 The TANGO Protocol: Informal Sketch

The concrete TANGO protocol leverages the observations discussed in Section 3.1 as follows:

**User Specification** The distinction between guards and methods is made explicit thanks to the relational setting, as explained in Section 3.2. For our current prototype, we ask the user to decorate monotonic guards with the Monotonic attribute, as exemplified in Figure 1. This attribute is likely amenable to automatic inference, but we leave this enhancement for future versions of our tool.

---

1. We refer to monotonicity in the mathematical sense, whereby a function $f$ between a pair of ordered sets is monotonic if it preserves the given order; i.e. $\forall x, y. x < y \Rightarrow f(x) < f(y)$.
2. We stress that other approaches, based e.g. on ADTs or on object-oriented notions of encapsulation, are also plausible. Recent studies have in fact shown that abstract-level synchronization, according to ADT operations and their semantic effect, can improve parallel performance dramatically [29, 37]. We leave this exploration for future research.
3. In our discussion here, and in general, we refer to tasks and loop iterations interchangeably, where what we mean by a task is the atomic parallel execution of a single iteration.
**Compile-time Analysis** At compile time, a lightweight static analysis computes a safe approximation of the effects of each iteration. Specifically, for loops iterating over sequences, the analysis tries to link these effects to a symbolic representation of the element pointed-to by the loop variable (the edge variable in the loop in Figure 1).

**Runtime Synchronization** At runtime, during parallel loop execution, the TANGO synchronization protocol automatically builds “safe” views for monotonic guards, as illustrated with graphs (a) and (b) in Figure 2. The view is constructed according to the possible effects of higher-importance concurrent tasks (e.g. earlier iterations, as in REVERSE DELETE). If a monotonic guard evaluates positively based on a limited view of the entire state, then parallel progress has been achieved. Otherwise, there are two options: If the guard fails, but all higher-importance tasks have completed already, then the guard result is serializable. Otherwise, view construction is retried per the observation that the possible effects of parallel tasks only decrease with time (becoming empty when a task completes), permitting a more complete view on retry.

A remaining challenge, which we have yet to discuss, is support for parallel method execution (like the edges.Remove and edges.Insert calls in Figure 1). If, for instance, the second iteration deletes edge \((n_1, n_2)\) while the first iteration is still running, then parallel execution is not necessarily serializable, since there is no sequential run where the first iteration observes a state where edge \((n_1, n_2)\) is absent.

To deal with this challenge, TANGO follows an approach that is similar in spirit to flat combining [28]. Instead of directly applying a method to the shared state, TANGO records the method application to a task-private log. Logged methods are accounted for in views constructed for the task (e.g. the deletion of \((n_1, n_2)\) in graph (b) in Figure 1). This permits a task to proceed through methods without blocking.

Moreover, if task \(t\) is about to complete, but concurrent tasks that must not observe its effects are still running, then instead of either committing \(t\)’s private log (which is wrong) or blocking \(t\) until these tasks finish (which is inefficient), TANGO appends \(t\)’s log of outstanding methods to that of a running task, \(t’\), where \(t’\) is the live task that immediately succeeds \(t\) in importance. \(t\) can then terminate, and \(t’\) becomes responsible for executing \(t’\)’s methods, ensuring that these are applied to the shared state in a serializable order.

### 3.4 Discussion: Scope and Limitations

Our approach generalizes several recent efforts to parallelize specific algorithms, including

- the helper-threads (HT) scheme [34, 45], where auxiliary threads offload work from the main thread by performing computations ahead of time, in parallel, either pessimistically [34] or speculatively (HT+TM) [45]; and
- the dynamic-independence (PMS) approach [8], which exploits the observation that in certain greedy algorithms, such as the algorithm for finding a maximal independent set (MIS), each iteration only depends on a subset of the previous iterations to run a task ahead of its designated time, as soon as the tasks it depends on have completed.

Both HT and PMS enforce pessimistic parallelization, letting background threads execute computations ahead of time when doing so is provably serializable. TANGO generalizes these two approaches in enforcing weaker preconditions for correct parallelization. From a practical standpoint, TANGO asks only for monotonicity annotations, and obtains comparable speedups to HT and PMS on the benchmarks motivating these approaches.

**Runtime Overhead** The main challenge faced by TANGO is to constrain the cost of constructing task-specific views. The practical limitation this places is that the effects of a task must have a compact description. This is true of REVERSE DELETE, where the substate manipulated by a given loop iteration is a single edge. TANGO incurs tolerable overheads over the 12 algorithmic benchmarks from our suite, the worst case being a factor of 1.35x compared to a sequential version of the benchmark (cf. Table 2).

**Applicability** As our analysis so far suggests, TANGO is geared toward computations where guards are nondeterministic with multiple admissible witnesses, and the intentions of a task refer to a small portion of the shared state. Two broad categories of applications satisfying this profile are

- pruning algorithms, such as REVERSE DELETE, the alpha-beta search algorithm [47] and decision-tree minimization [41], where a witness contained in a modification-free substate secures early deletion of an element; and dually,
- incremental algorithms, such as KRUSKAL MSF [15], task scheduling [15] and quick elimination [2], where a witness contained in a past version of the evolving solution secures early exclusion of an element.

We emphasize that though TANGO was built as a tool for loop parallelization, the principles underlying it are general and applicable also in other parallel contexts, such as concurrent data structures and reactive systems.

Our experiments (described in Section 6), where we consider benchmarks of varying characteristics, indicate that TANGO is highly effective in parallelizing applications with nondeterministic reads and granular intentions, like KRUSKAL MSF and REVERSE DELETE, and is also able to utilize some of the available parallelism in codes with deterministic reads. However, algorithms where the intentions of a task are initially broad, and undergo refinement only at a late point, are not good candidates for TANGO.
4. Concurrent Tasks, Specialization, Priority

In this section, we provide a complete technical description of our approach. We begin by stating a restrictive requirement for serializable execution of an operation in parallel with concurrent tasks, whereby that operation must commute with all possible future behaviors of the tasks. We then introduce specialization as a means of turning this restriction into an advantage: Instead of requiring commutativity over all behaviors, concurrent tasks can select specific behaviors that are mutually compatible and commit to these behaviors. Finally, we introduce a notion of priority, and show that this leads to a progress guarantee.

4.1 Execution Model

We assume a generic programming model with standard control structures (if, while, sequential composition, etc). There is a single shared state from an unspecified state space \( \Sigma \). There are a finite number of threads, each with a unique identifier \( \tau : \mathcal{T} \), that access the shared state in only two ways:

- **Guard** \( g : \Sigma \rightarrow \{ \text{true}, \text{false} \} \) is an observation that a thread makes on the shared state.

- **Method** \( m : \Sigma \rightarrow \mathcal{P}(\Sigma) \) is a modification that a thread makes on the shared state.

Guards and methods can be realized in many ways, depending on the particular language context. The most natural example is Dijkstra’s language of Guarded Commands [17]. Another example is a language that performs low-level memory operations, where \texttt{read} is a guard and \texttt{write} is a method. As discussed in Section 3.2, for this paper we have chosen a relational program representation, whereby the shared state is expressed as one or more relations, guards are boolean query commands, and the methods are inserts and remove.

When tasks are executed concurrently without synchronization, they do not always behave correctly. There may be executions in which one task interferes with another task.

**Example 4.1.** Assume an unsynchronized parallel run of the code in Figure 1 on graph (a) in Figure 2, where task \((n_2, n_5)\) is the first to execute. That task would then delete edge \((n_2, n_5)\) based on witness \([n_2, n_1, n_4, n_5]\), which is invalid, because in sequential execution of the code, both edge \((n_1, n_4)\) and edge \((n_1, n_2)\) would be deleted, rendering edge \((n_2, n_5)\) part of the MST.

For safe concurrency, tasks must be aware of the potential effects of concurrent tasks, as well as their assumptions on the shared state. As is standard, we use the notion of commutativity to quantify interference [35, 36]:

\[
m \triangleright_{\sigma} m' \iff [m ; m']\sigma = [m'] ; m\sigma
\]

If the commands of two threads (pairwise) commute, then it is safe for them to proceed in parallel. Intuitively, this is because their interleaved execution can be rearranged into a serial execution. However, our work takes a new tack: Our notion of conflict is based on what may happen in the future rather than what has happened in the past. To this end, we add two criteria:

**Condition 4.1.** A method \( m \) performed by a task \( \tau \) must not change the boolean value of every guard \( g \) that may ever be observed by a concurrently executing task \( \tau' \).

**Condition 4.2.** A method \( m \) performed by a task \( \tau \) must commute with every method \( m' \) that may ever be performed by a concurrently executing task \( \tau' \).

If Conditions 4.1 and 4.2 hold, then every execution of the concurrent tasks is serializable.

These two conditions are strong and thus somewhat surprising. We are requiring that a method commutes with every method/guard that any other task may ever perform. This is stronger than the pairwise synchronization conditions that hold over histories in traditional systems such as TM or lock-based mutual exclusion.

**Example 4.2.** For the code in Figure 1 and graph (a) from Figure 2, the \texttt{FindPath} guard over \((n_1, n_2)\) does not commute with the edges. Remove method over \((n_1, n_4)\) because of witness path \([n_1, n_4, n_5, n_2]\). Parallel execution of these two commands is possible, however, assuming the \((n_1, n_2)\) guard can only select witness \([n_1, n_3, n_4, n_5, n_2]\).

Our important insight, which we formulate in the following, is that the strict synchronization model due to Conditions 4.1 and 4.2 makes potential conflicts explicit. This lets tasks coordinate their present behaviors in order to avoid as much conflict as possible in the future. For this reason, as we show in Section 6, our approach is in fact competitive with many leading-edge concurrent programming algorithms.

4.2 Specialization and Preservation

Nondeterminism complicates commutativity judgments, because there are multiple possible behaviors for a task. Concurrent tasks can only take a next step if that step commutes with all of these behaviors. This is seen in Conditions 4.1 and 4.2, where \( m \) must commute with all possible commands that may be executed by concurrent tasks.

We turn this limitation of synchronizing nondeterministic tasks into an advantage through specialization: A task specializes by restricting its possible behaviors, which increases concurrency by relaxing the commutativity checks. We define specialization for both methods and guards as follows:

**Definition 4.1** (Specialization). Specialization of methods \( \preceq_M \) and of guards \( \preceq_G \) is as follows:

\[
m_1 \preceq_M m_2 \iff \forall \sigma. [m_1]\sigma \subseteq [m_2]\sigma
\]

\[
g_1 \preceq_G g_2 \iff \forall \sigma. [g_1]\sigma \Rightarrow [g_2]\sigma
\]

We will often drop the subscript on \( \preceq \) as it is clear from context. For the definition of dynamic specialization over a
specific state $\sigma$, we simply remove the universal quantifica-

tion.

Above a specialized command is a state transformer that may
have less nondeterminism than the original command. If a
specialized guard holds, then the original guard must hold,
but the specialization may have a particular implementation
that, for example, is focused on a portion of the state space.

**Example 4.3.** For the setting described in Example 4.2, if
we specialize the $\text{FindPath}$ guard over $(n_1, n_2)$ by restrict-
ing its evaluation to edges that are invariant over higher-
priority tasks (namely, by excluding $(n_1, n_4)$, as illustrated
in graph (b) in Figure 2), then we obtain witness path
$[n_1, n_3, n_4, n_5, n_2]$. This witness implies commutativity be-
 tween the guard and the edges. Remove command over edge
$(n_1, n_4)$.

The advantage with specialization is that a task can make
an informed choice of which behaviors to eliminate and
which behaviors to retain based on the behaviors of concur-
rent tasks. This permits inter-task coordination, where non-
determinism allows tasks to choose behaviors that are mutu-
ally compatible, thereby enabling safe parallel progress.

**Theorem 4.3** (Preservation). Method specialization, as well
as specialization of monotonic guards, preserve the behav-
iors and serializability of the original system.

**Proof Sketch.** For a method, specialization yields less post-
states than the original method, but these are all admissible
due to containment. For monotonic guards, successful eval-
uation on a substate implies successful evaluation on the en-
tire state thanks to monotonicity. Local preservation of be-
haviors and serializability implies a system that is globally
serializable.

Of course we have to be careful when we specialize that we
have not performed trivial specialization, simply eliminat-
ing all behaviors. We address this concern in the remainder
of this section, where we show that if tasks are ordered by
priority, then at each point, at least one of the tasks (specif-
ically, the highest-importance one) is free of specialization
obligations, obviating the threat that all tasks specialize triv-
ially, thereby blocking execution.

### 4.3 Priority and Progress

While specialization—in conjunction with commutativity—
guarantees preservation, progress is still not ensured. The
difficulty is in verifying that at least one task can step
through a guard, which is not necessarily the case unless
the tasks are assigned priorities.

**Example 4.4.** Assume a parallel run of $\text{REVERSE DELETE}$
on graph (a) in Figure 2, where the first two iterations
have completed, and now the third and fourth iterations—
with respective edges $(n_4, n_5)$ and $(n_2, n_5)$—simultaneously
specialize, resulting in view (c). This is a stuck state, because
for both iterations, (i) the $\text{FindPath}$ guard fails, and (ii) the
view is partial, disabling progress through the guard when
its evaluation is negative.

In response to this problem of nonterminating execu-
tions, we introduce task priorities. We show that with this
strengthening, tasks can execute in parallel while ensuring
both progress and preservation. Intuitively, the power of pri-
oritization is in breaking the symmetry between tasks, such
that a higher-priority task need not account for the behavior
of lower-priority tasks. This yields a progress guarantee.

The rules we have presented thus far provide a serializ-
able concurrent model in which tasks specialize their nonde-
terministic choices in order to proceed in concert. We now
revise Conditions 4.1 and 4.2, obtaining new conditions that
ensure progress (provided the tasks terminate sequentially).

We first define priority:

**Definition 4.2** (Priority). Let $\ll: (T \times T)$ be a total order
on task identifiers.

We can now state the two conditions.

**Condition 4.4.** A method $m$ performed by task $\tau$ must not
change the boolean value of a guard $g$ that was observed by
another task $\tau'$ such that $\tau' \gg \tau$.

**Condition 4.5.** A method $m$ performed by task $\tau$ must com-
mute with every method $m'$ that may ever be performed by
another task $\tau'$ such that $\tau' \gg \tau$.

The formal detail of these conditions involves quantifying
over all possible executions, and for each execution, per-
forming a pairwise check whether $m$ can move to the left
of methods and guards of other concurrent tasks of higher
priority [35]. Intuitively, priority manifests as a discount: It
lets a task ignore both the assumptions (i.e. guards; Con-
dition 4.4) and the effects (i.e. methods; Condition 4.5) of
lesser-priority tasks, which means that at each point, at least
one task can progress without constraints.

**Example 4.5.** We return to Example 4.4 and fix it per Con-
ditions 4.4 and 4.5. The third iteration (with respective edge
$(n_4, n_5)$) is more important than the fourth one (with edge
$(n_2, n_5)$), which means that it can ignore the effects of the
fourth iteration. Thus, the graph view by the third iteration
after the first two iterations have terminated is complete, co-
inciding with the actual graph state. This permits progress
through the $\text{FindPath}$ guard in spite of its negative evalua-
tion. The same situation arises for the fourth iteration after
the third iteration completes.

**Theorem 4.6** (Progress). For tasks $\tau_1 \ldots \tau_n$ that each ter-
minate in a sequential setting and priority relation $\ll$, either
(i) all tasks have completed, or else (ii) there exists a task $\tau_i$
that can perform a method or a guard.

**Proof Sketch.** The highest-priority task (denoted $\tau_i$) can pro-
ceed without waiting for other tasks. Condition 4.4 ensures
that the values of all guards of the highest-priority task are not altered by concurrent tasks. Condition 4.5 ensures that if another task \( \tau_j \ll \tau_i \) executes a method \( m \) concurrently, then \( m \) commutes with all the operations of \( \tau_i \). Thus, there is necessarily an equivalent serial history in which \( \tau_i \) happens before \( \tau_j \).

5. The TANGO Protocol

We now describe the TANGO synchronization protocol, which instantiates the formal framework developed in Section 4.

5.1 Intention Analysis

In the concrete TANGO protocol, we model the effects of a task as the subtrees it may modify. This assumes that the state is decomposable into subtrees: \( \sigma = \bigcup_i s_i \) for atomic subtrees \( s_i \), with the assumption that the state space \( \Sigma \) is closed under subtrees. For a concrete state, the standard decomposition is into memory locations (e.g., variables and object fields), the (low-level) state description being a mapping from memory locations to their associated value [52].

In TANGO, the shared state is expressed as relations, and so state decomposition is in terms of qualified tuples. The possible effects of the two simple methods are straightforward, as follows:

\[
\text{mod}(\text{insert } r \ t) = \langle r, t \rangle \\
\text{mod}(\text{remove } r \ t) = \langle \neg r, t \rangle
\]

To induce correct task-centric state views, containing only portions of the shared state that are invariant over the effects of concurrent tasks, the runtime system must be aware of what these tasks may do. Computing the effects of a task on the fly, during a parallel run, is expensive. Instead, TANGO features a lightweight static data-flow analysis, tailored for loop structures.

The analysis is performed at compile time, and is a forward analysis starting at the loop head. When the analysis encounters a method, it stores its affected subtree as a symbolic access path (e.g., \( \langle \text{edges}, \text{edge} \rangle \) for the \text{Remove} and \text{Insert} statements in Figure 1, modeled as \text{remove} and \text{insert} respectively).

After the analysis reaches a fixpoint over the control structure of the loop body, each collected access path \( \langle r, t \rangle \) is processed as follows:

1. If \( \langle r, t \rangle \) is a constant access path (i.e., both the relation and the tuple are amenable to static resolution), then it flows into the may-modify set as is.
2. Otherwise, if the loop iterates over a sequence—and therefore the loop variable points to a tuple—and \( \langle r, t \rangle \) is identical to the (symbolic) tuple pointed-to by the loop variable, then again \( \langle r, t \rangle \) flows into the may-modify set directly. This is the case e.g., in Figure 1 with \( \langle \text{edges}, \text{edge} \rangle \). The runtime system can resolve symbolic tuples of this sort before the loop starts executing based on the elements in the argument sequence. Hence the relevance of this case.

3. A third situation is where the tuple as a whole is unavailable, but the analysis can statically resolve the relation \( r \). In this case, the abstract tuple \( \langle r, * \rangle \) flows into the may-modify set, indicating that a given iteration may mutate any tuple in relation \( r \).

4. The final case is where neither the relation nor the tuple are amenable to static resolution. This translates into \( \bot \), meaning that an iteration may modify any tuple in any relation.

The results of the compile-time intention analysis are made available to the runtime system as a may-modify oracle, mod, mapping tasks to symbolic tuple descriptions.

5.2 Runtime Protocol

The runtime TANGO protocol is described in Figure 3 according to the operations it synchronizes: method execution (\text{EXECUTEMETHOD}), task termination (\text{EXIT}) and guard evaluation (\text{TESTGUARD}). For clarity, we separate the formal description in Figure 3 into two contexts, treating the most important task, max \( T \), separately from the other tasks.

max \( T \) executes as if no other task is running in parallel. To avoid from interfering with max \( T \), as well as any other higher-importance task, another task \( \tau \) must therefore log its mutations instead of executing them directly. To ensure that its control choices—governed by guards—are serializable, \( \tau \) must either block until it becomes the most important task for a general guard \( g \), or ensure that \( g \) is successful if \( g \) is nondeterministic (otherwise retrying until either \( g \) becomes successful or \( \tau \) becomes max \( T \)). If \( \tau \) is about to complete prior to becoming max \( T \), then it can exit and transfer its log of outstanding methods to the (live) task succeeding it in importance. Otherwise, upon becoming max \( T \) (the event), \( \tau \) must first clear its own log of pending methods (\text{log}(\tau)) or (\text{log}(\tau)). When max \( T \) is about to terminate, it executes all pending methods transferred to it according to the order between the respective tasks.

We now relate the informal description above to Figure 3. The first context is where the argument task \( \tau \) ranks first in priority. In this case, methods and guards are evaluated directly over the shared state \( \sigma \) (the \([\cdot]\) notation). On exit, the task applies the outstanding methods in its log (due to other tasks) to the shared state.

In the complementary context where \( \tau \) is not the highest-priority task, methods are recorded into the task-private log, rather than performed. When the task is about to terminate, it delegates its outstanding log to the task succeeding it in priority. Evaluation of a general guard must block until the task becomes first in priority. A monotonic guard, however, is amenable to premature evaluation over the portion of the shared state that is invariant over concurrent tasks (\( \sigma \setminus \bigcup_{\tau' > \tau} \text{mod}(\tau') \)), where we account for logged meth-
5.3 Prototype Implementation

TANGO is implemented both in Java and in C# (.NET). The TANGO library provides a parameterized implementation of the Relation ADT, featuring efficient linearizable implementations of the primitive relational operations. TANGO also provides common specializations of the relation type (UnaryRelation, BinaryRelation, etc.), as well as a collection of useful guards (including transitive reachability, as in FindPath, and existential tests). Other ADTs, such as Map, Set, List and Graph, are implemented atop the Relation ADT. As stated earlier, the C# version of TANGO is built over the LINQ syntax features, enabling concise coding of loops with side effects, as exemplified in Figure 1. Finally, TANGO’s algorithm for statically approximating task intentions makes use of the WALA framework.

6. Experimental Evaluation

In this section, we describe two sets of experiments that we conducted over TANGO. The first experiment compares TANGO with several specialized parallelization schemes on 12 algorithmic benchmarks. We then report on our experience in integrating TANGO into a commercial application.

6.1 Algorithmic Benchmarks

Our benchmark suite for the first experiment consists of 12 benchmarks: 8 greedy algorithms and 4 dynamic programming (DP) algorithms. The main characteristics of the benchmarks are summarized in Table 1. Our choice of benchmarks was driven by two considerations: First, the chosen algorithms exhibit varying levels of available parallelism, which enables proper investigation of when and why our approach would work. This is also the reason why for all the greedy benchmarks we used two different input profiles, one permitting more parallelism than the other. Second, all the benchmarks in our suite are popular, being of practical interest and applicability. Some of the benchmarks have also been parallelized in the past, which enables comparing our approach with other techniques.

Experimental Setting Our experiments are based on the C# version of TANGO. The experiments were conducted on a 16-core IBM X3550 M2 machine with 100GB of RAM running Windows Server 2008 R2 64-bit with Microsoft .NET Framework v4.0.30319. Inputs were generated randomly, as detailed in Table 3. Each configuration of benchmark, input profile and parallelization technique was run 11 times. The reported numbers are the average of the 10 last executions (excluding the first, cold run).
Our choice of a machine with a high volume of RAM was deliberate, so as to minimize the impact of garbage collection on performance. This is particularly effective in the .NET setting, where contrary to the Java Virtual Machine (JVM), the Common Language Runtime (CLR) does not set any predefined limit on heap size, and instead depends on the host operating system for memory allocation.

In addition to TANGO and the HT and PMS approaches, presented in Section 3, we included the following parallelization techniques in our evaluation:

**Top-down Parallelization (TD)** [50] This scheme parallelizes DP algorithms in a top-down manner, where solutions to subproblems are cached in a lock-free hash table, and the order in which subproblems are computed is random.

**Work Distribution (BSP)** [13] This framework works for associative operators, such as min and max. Each thread applies the operator to a portion of the candidate values, and broadcasts its local candidate to all other threads.

**Lazy Speculation (OPT)** [51] This form of optimistic synchronization tests for conflicts only at transaction commit time. The transaction operates on a privatized version of the shared state.

For 8 of the benchmarks, we compared TANGO with the technique that was originally proposed to parallelize that benchmark. Our choice of parallelization technique for the remaining 4 benchmarks (GREEDY COLORING, RULE PRUNING, REVERSE DELETE and \(\epsilon\)-NET) was guided by...
their similarity to the other 8 benchmarks. The available parallelism in Greedy Coloring has the same structure as MIS, being due to the sparse dependencies between the iterations. Rule Pruning and e-net are similar to Kruskal MSF in that a helper thread can determine, ahead of time, the redundancy of an element. Finally, for Reverse Delete, speculation is the only technique out of the above that yields a sound parallelization solution.

**Performance Results** The speedup and overhead numbers appear in Table 2. Speedup is calculated as the ratio of sequential running time to parallel running time using 16 threads. Overhead is the ratio of execution time using 1 thread to sequential execution time. For benchmarks with two inputs profiles, the left number corresponds to the low-parallelism input. The speedup trends going from 1 to 16 threads are shown in Figures 4–15, where there are either one pair or two pairs of plotted graphs in each figure, visualizing Tango compared to its competing approach over one or two workloads. Detailed workload characteristics are provided in Table 3.

The main performance penalty of Tango lies in inducing task-centric views, where a view is computed by factoring in the effects of concurrent tasks. On average, across all benchmark and workload configurations, this leads to a slowdown of 1.18x (i.e., less than 20%) compared to the sequential implementation. In the worst case (on Greedy Coloring 10Kx500K), Tango incurs an overhead of 1.35x. We view Tango’s overhead as tolerable, and expect that with more engineering, this overhead can be mitigated.

Tango’s memory consumption is correspondingly efficient. In profiling runs over the benchmarks in our suite, the memory footprint of Tango was no more than 25% higher than the sequential version. This is not surprising. Tango maintains may-modify logs only for running tasks (as opposed e.g., to STM, where read/write logs for terminated tasks are still needed by concurrent tasks to determine conflict). This means that at any point during parallel execution, the number of active logs is bounded from above by the number of processors (16 in our case). Thus, if the may-modify information of a given task can be represented succinctly, then the memory overhead of Tango becomes constrained.

This is indeed the case with all our benchmarks, wherein the effects of a task can be summarized concisely. In Reverse Delete, for instance, task modifications are restricted to a single graph edge. At the other extreme, in the “pathological” case of Dijkstra SP, task modifications span the entire state, which can again be stated concisely. This explains the overhead data in Table 2.

Finally, for bottom line performance, Tango achieves a speedup of over 3x on 6 of the benchmarks, being comparable to its competing specialized technique (if not faster) on 10 out of the 12 applications (Dijkstra SP and Jarvis March being the exceptions). Tango’s speedup on Dijkstra SP and Jarvis March is negative. On Dijkstra SP, even the specialized technique of [45] achieves negligible speedup. On Jarvis March, however, Tango is significantly worse than the technique of [13], which obtains a speedup of up to 7x. In contrast, Tango outperforms lazy speculation on Reverse Delete by a factor of > 2.

**Discussion** The speedup statistics reported above are consistent with our expectations. Tango favors nondeterminism in read operations, as well as granular and predictable write operations. The classic example is Reverse Delete (cf. Section 3). This explains the poor results on Jarvis March and Dijkstra SP, where a task can only communicate conservative intentions until a late point in its execution, and read operations are deterministic.

The reason why the BSP approach works well for Jarvis March is that it exploits intra-operator, rather than inter-operator, parallelism. Instead of parallelizing the entire task of extending the convex hull, BSP parallelizes only the inner loop searching for the next point in the hull. We could do the same, applying Tango to the (inner) search task, but preferred to remain consistent with the other benchmarks.

Greedy Coloring and the DP algorithms (Edit Distance, Knapsack, Seq. Alignment and Partition) are more compatible with Tango, though still not its ideal clients. While a task has narrow intentions as it commences (which is good), read operations are deterministic, leaving limited opportunity for parallel guard evaluation. Hence, the speedups obtained by Tango fall below the competing approaches on these benchmarks. The remaining 5 benchmarks (Reverse Delete, e-net, Rule Pruning, MIS and Kruskal MSF) feature both nondeterministic reads and granular intentions, and thus behave well under Tango.

Though Tango outperforms competing approaches only in specific cases, we find that its advantage comes from (i) its generality compared to specialized parallelization techniques devised for specific domains and algorithms, as well as (ii) its usability as an encapsulated library in two popular general-purpose languages, which requires only minimal annotation effort from the programmer.

We conclude with two remarks on the effect of prioritization on performance. First, most of our benchmarks perform ordered iteration over the input collection, and thus force a specific prioritization scheme. The remaining algorithms (e.g., Greedy Coloring) are unordered, but are normally implemented atop one or more heuristic ordering schemes (e.g., prioritizing a node inversely to its degree in Greedy Coloring). We did the same. We can report, however, that in preliminary experiments we performed that involved manipulation of concurrent data structures (like graphs and Maps) by random clients, where prioritization was decided ad hoc according to the starting time of a task, the variance we observed between executions was relatively low. We defer further analysis of the effect of the prioritization scheme on performance, and optimization thereof, for future work.
Figure 4. Speedup on Dijkstra SP

Figure 5. Speedup on Edit Distance

Figure 6. Speedup on \( \epsilon \)-Net

Figure 7. Greedy Coloring

Figure 8. Jarvis March

Figure 9. Knapsack

Figure 10. Kruskal MSF

Figure 11. MIS

Figure 12. Partition

Figure 13. Reverse Delete

Figure 14. Rule Pruning

Figure 15. Seq. Alignment

Figure 16. Speedup results

Figure 17. Architecture of the commercial engine, including illustration of its flow
6.2 Case Study: Commercial Security Testing Tool

For the second study, we considered a testing module contained in IBM AppScan Standard Edition, a commercial product for dynamic detection of web security vulnerabilities. We report on our experience in applying TANGO systematically to “heavy” (sequential) loops in the code. The methodology and hardware configuration for this experiment are the same as in the previous one.

Tool Description The module’s architecture is outlined in Figure 17. The input to the testing engine is an HTML page. Outgoing links are extracted, and “injection points”, such as parameter, cookie and session values, are identified. Next, for each candidate injection point, the engine checks for security vulnerabilities by sending further requests for the respective link, this time substituting the original input value with different test payloads (e.g. a script block, when checking for a cross-site scripting vulnerability). The response received from the subject application is then validated to confirm whether the attack had succeeded, in which case the discovered vulnerability is reported to the user.

Parallelization Process To parallelize the subject testing tool, we first reviewed the tool’s profiling data to identify “heavy” sequential loops. Manual inspection of the candidate loops disclosed three candidates where TANGO is immediately applicable. These loops perform greedy pruning, which renders the guard governing the observable effect of an iteration nondeterministic and monotonic.

The first loop (Reflection) iterates over input points, mapping each point to its respective reflection contexts, and eliminating redundant targets based on context overlap. The other two loops (Testing-{1,2}) fire test payloads, checking for each payload whether it is compatible with the tool’s online model of the site’s server-side defenses, and if so, sending it, as the AppScan documentation explains in detail. The first loop builds a coarse model of the site’s defenses, which the second loop refines if the first loop fails.

Having settled on the parallelization targets, we next applied two transformations: First, we decorated the loops with the Monotonic attribute, and second, we replaced ADT implementations accessed within the loops with their counterparts from the TANGO library (e.g. the TANGO Set implementation). These transformations proved straightforward, requiring little time and few code changes.

Performance Results To measure the impact of our parallelization transformations, we compared between the original version of the tool, where the candidate loops are executed sequentially, and a version where the loops are parallelized using TANGO. The data inputs were sampled randomly from the tool’s test suite, and the experiments were conducted in the same way as before.

The performance results are summarized in Table 4.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Reflection</th>
<th>Testing-1</th>
<th>Testing-2</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction</td>
<td>68%</td>
<td>32%</td>
<td>41%</td>
<td>58%</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.47x</td>
<td>3.12x</td>
<td>2.43x</td>
<td>1.73x</td>
</tr>
</tbody>
</table>

Table 4. Speedup statistics for TANGO on the commercial testing module

This table shows the fraction of time TANGO required to execute each of the loops compared to its sequential counterpart. For example, 32% for the Testing-1 loop means that if the average sequential execution time of the loop is t seconds, then executing this loop under TANGO requires 0.32·t seconds, which constitutes a speedup of 3.12x. Overall running time under TANGO of the entire code—including the code outside the subject loops, which was executed sequentially also under TANGO—was cut down to 58% of the sequential running time, improving performance by more than 40%.

Discussion TANGO proved effective for the Testing-1 and Testing-2 loops, yielding respective speedups of 3.12x and 2.43x. The main reason is that for most data inputs, parallel guard evaluation succeeds: The partial model due to early payloads suffices for parallel rejection of a large number of future payloads, which translates into noticeable performance improvement. In the Reflection loop, however, there is less parallelism to exploit. The majority of discovered reflection contexts are independent, obviating most attempts for (early) concurrent pruning of redundant attack points.

7. Related Work

In this section, we survey other parallelization approaches that have features in common with TANGO. We refer the reader to [10] and references therein for a discussion of specification, verification and testing tools designed for nondeterministic codes.

Language Features NESL [6, 7] is a data-parallel programming language based on ML. Similarly to TANGO, NESL enables concise coding of a wide range of algorithms from different fields, including sequences and strings, graphs and computational geometry. Pregel [40] is a distributed programming framework, providing a simple API for programming graph algorithms while managing the details of distribution invisibly, including messaging and fault tolerance. Green-Marl [31] also specializes in graphs, focusing on graph analysis algorithms. The Green-Marl language captures the high-level semantics of the algorithm, allowing the Green-Marl compiler to apply nontrivial parallelization transformations.

None of the above languages leverages nondeterminism for parallelism. Another difference is that TANGO is implemented as a library for general-purpose languages (currently: Java and .NET). As illustrated in Figure 1, TANGO’s LINQ frontend exposes declarative, SQL-like syntactic constructs for concise parallelization of challenging loops. This is close in spirit to Dryad [32] and DryadLINQ [33], which...
provide a general-purpose distributed execution engine for data-parallel applications. These systems, however, are constrained to side-effect-free loops, where TANGO is able to parallelize loops with irregular dependencies.

Harris and Fraser [24] use conditional critical regions (CCRs) for concurrency control. The user guards an atomic region by a boolean condition, with calling threads blocking until the guard is satisfied, which is reminiscent of the TANGO flow. The key difference is that [24] ensures the atomicity of a CCR via STM, whereas TANGO applies pessimistic synchronization, leveraging guard monotonicity. The retry construct in STM Haskell [25] allows a task to backtrack from its current behavior and select an alternate path that is hoped to have less conflict. However, retry does not involve coordination between threads, so one is still at the mercy of nondeterministic luck and what has been done in the past.

Communication and Visibility The TIC programming model [49] extends standard STM by allowing threads to observe the effects of other threads at selected points, thereby supporting operations that wait or perform irreversible actions inside transactional code. Lesani and Palsberg [38] describe a semantics that combines STM with message passing through tentative message passing, which keeps track of dependencies between transactions to enable undo of message passing in case a transaction aborts. Certain TM contention managers, such as the greedy contention manager [4, 21], let a transaction wait for a conflicting transaction or abort it depending on their status and priority.

Gueta et al. [23] describe a methodology for utilizing foresight information for effective synchronization in general-purpose systems. They apply their approach to safe composition of ADT operations (i.e., without deadlocks and atomicity violations). Their implementation consists of static analysis of the client code to compute which ADT operations it may use at different points in the execution, combined with lazy runtime synchronization over a “smart” implementation of the ADT, which leverages the semantics of the ADT’s operations to increase parallelism. TANGO also bases synchronization on foresight, but with the different focus of parallelism due to latent nondeterminism, where we introduce task specialization as a synchronization primitive. TANGO also bases synchronization on foresight, but with the different focus of parallelism due to latent nondeterminism, where we introduce task specialization as a synchronization primitive.

Dataflow programming [3, 16] makes data dependencies between tasks explicit by representing a program as a directed graph, where nodes represent units of work and arcs denote dependencies. Recently, there have been attempts to integrate dataflow abstractions into STM to enable controlled access to mutable shared state while constraining the number of false conflicts [19, 48]. TANGO takes a pessimistic approach, while enforcing fine-grained parallelization of applications with irregular dependencies, such as graph algorithms. Also, the purpose of communication in TANGO is to publish task intentions, rather than propagate data values.

Botincan et al. [9] present a technique for synchronization synthesis, which allows fine-grained parallelization of resource usage by using separation logic, rather than pointsto analysis. The parallelization transformation inserts grant and wait barriers to transfer resources between concurrent tasks, which is reminiscent of the TANGO communication mechanism. Our focus, however, is on nondeterministic choice. TANGO focuses a task on a particular way of evaulating a nondeterministic action, which permits fine-grained synchronization, reducing contention, similarly to the idea of multiple granularity locking [20, 43].

Beyond Data Independence TANGO is able to enforce parallelism in the presence of data dependencies, as we illustrated in Section 3 for REVERSE DELETE. Another way of transcending data dependencies is value speculation [46], where a task is run speculatively based on a guess of the data values that would flow into its input parameters at its designated point of execution. The Alter framework [53] identifies and enforces optimistic parallelism that violates certain dependencies while preserving overall program functionality. This is governed by user annotations allowing reordering of loop iterations or stale reads. The novel feature of TANGO in being able to exploit available parallelism in the presence of data dependencies is its reliance on the notion of specialization.

The Bloom language [1] leverages theoretical results in monotonic logic to achieve eventual consistency in distributed programs. Bloom programs describe relationships between distributed data collections that are updated monotonically (i.e., in an incremental fashion). Conway et al. [14] generalize Bloom to a lattice-parameterized system, Bloom⁺. TANGO is similar to Bloom in leveraging monotonicity, but has wider applicability, supporting general-purpose languages and arbitrary state manipulations, beyond incremental updates (as in REVERSE DELETE).

8. Conclusion and Future Work

This paper takes a first step in exploiting parallelism due to nondeterminism. Existing approaches lack the necessary coordination mechanisms to guide concurrent tasks toward nonconflicting nondeterministic choices. We develop a novel concurrency paradigm that reduces conflict by letting nondeterministic operations specialize based on the potential future behavior of concurrent tasks. We have implemented our approach in TANGO, a synchronization protocol that extracts parallelism from pure monotonic queries with available nondeterminism. TANGO generalizes several existing parallelization schemes, and achieves encouraging performance results in our experiments.
In the future, we would like to introduce speculative capabilities into TANGO. The idea is to let tasks make nondeterministic choices heuristically and optimistically, where bad choices, resulting in conflict with other (higher-importance) tasks, are rolled back. (Inverse operations are startrightward to implement in the relational setting.) Speculation has attached costs (mainly logging and wasted work), but at the same time, it carries several important advantages: First, it obviates the need to approximate and monitor task intentions, which could potentially lower the runtime overhead of TANGO; and second, speculation is more permissive than our current model, which could expand the scope of TANGO beyond monotonic queries and improve its performance on irregular benchmarks.

As for TANGO’s current synchronization model, we intend to enhance it to automatically infer monotonicity annotations, which is facilitated by the relational representation. We are also currently investigating how to perform partial specialization already at compile time by synthesizing mutual-exclusion-based synchronization for conflicting pairs of executions.

References


