Isolation for Nested Task Parallelism

Jisheng Zhao
Rice University
jisheng.zhao@rice.edu

Roberto Lublinerman
Google, Inc.
rluble@google.com

Zoran Budimlić
Rice University
zoran@rice.edu

Swarat Chaudhuri
Rice University
swarat@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Abstract

Isolation—the property that a task can access shared data without interference from other tasks—is one of the most basic concerns in parallel programming. While there is a large body of past work on isolated task-parallelism, the integration of isolation, task-parallelism, and *nesting of tasks* has been a difficult and unresolved challenge. In this paper, we present a programming and execution model called *Otello* where isolation is extended to arbitrarily nested parallel tasks with irregular accesses to heap data. At the same time, no additional burden is imposed on the programmer, who only exposes parallelism by creating and synchronizing parallel tasks, leaving the job of ensuring isolation to the underlying compiler and runtime system.

Otello extends our past work on Aida execution model and the *delegated isolation* mechanism [22] to the setting of nested parallelism. The basic runtime construct in Aida and Otello is an *assembly*: a task equipped with a region in the shared heap that it *owns*. When an assembly A conflicts with an assembly B, A transfers—or *delegates*—its code and owned region to a carefully selected assembly C in a way that will ensure isolation with B, leaving the responsibility of re-executing task A to C. The choice of C depends on the nesting relationship between A and B.

We have implemented Otello on top of the Habanero Java (HJ) parallel programming language [8], and used this implementation to evaluate Otello on collections of nested task-parallel benchmarks and non-nested transactional benchmarks from past work. On the nested task-parallel benchmarks, Otello achieves scalability comparable to HJ programs without built-in isolation, and the relative overhead of Otello is lower than that of many published data-race detection algorithms that detect the isolation violations (but do not enforce isolation). For the transactional benchmarks, Otello incurs lower overhead than a state-of-the-art software transactional memory system (Deuce STM).

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Languages, Design

Keywords Isolation, Programming abstractions, Irregular parallelism, Contention

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509534

1. Introduction

The demand for parallel programming is now higher than ever: inexpensive multicore processors are ubiquitous, and the bottleneck is now software rather than hardware. And yet, it is increasingly clear that current foundations for parallel programming, such as locks and messages, are low-level, complex, error-prone, and non-modular. Consequently, many research groups in industry and academia are seeking high-level models and languages for parallel programming that are intuitive as well as scalable.

A critical challenge in the foundations of parallel programming models is that of *isolation*: the property that a task can access dynamic subsets of shared data structures without interference from other tasks. A programming model providing high-level guarantees of dynamic isolation is able to rule out the perennial dual headaches of deadlocks and data races, while also relieving the burden of low-level reasoning about memory consistency models.

Such high-level models of isolation have received much attention in the research community in the last decade. Of existing approaches to this problem, the most well-known

is the transactional memory model [20]. However, integration of these high-level models, including transactions, with *nested* parallelism remains a difficult and unresolved challenge. This is unfortunate because nested parallel tasks form a natural programming idiom in many real-life settings. For example, suppose we declare a task A to be isolated, but want to extract additional parallelism within it. A natural way to do so is to let A fork parallel tasks B and C. However, the tasks B and C may need to be isolated from each other, and may need to synchronize (join) at some point. In general, B and C might fork other parallel tasks as well. As yet, there is no transactional memory solution that guarantees isolation while supporting nested parallelism of this sort. For parallel programming models that do support nested-parallelism (e.g. OpenMP 3.0[27], Cilk [5]), locks are still the major approach for implementing mutual exclusion.

In this paper, we present a programming and execution model, called *Otello*, that allows parallel tasks that are arbitrarily nested, and make irregular accesses to objects in a shared heap, to run in isolation. In our programming model, parallel tasks are declared to be isolated by a single keyword, and the programmer only exposes parallelism by creating and synchronizing parallel tasks, respectively using structured `async` and `finish` constructs, as in Habanero Java [8] and X10 [10]. The underlying compiler and runtime system are responsible for ensuring isolation between tasks.

Otello's implementation relies on extensions to the Aida execution model and the *delegated isolation* mechanism introduced in [22] to support nested isolated task-parallelism. Specifically, the Otello runtime views each task as being equipped with a set of shared objects that it *owns*. We refer to this combination of a task and its owned objects as an *assembly*. An assembly can only access objects that it owns, thus guaranteeing isolation. When an assembly A needs to acquire ownership of a datum owned by assembly B (i.e., a *conflict* happens), A transfers—or *delegates*—its code and owned region to assembly C in a way that will ensure isolation with B, leaving the responsibility of re-executing task A to C. The choice of C, as well as the precise mechanics of delegation depends on the nesting relationship between A and B.

Aside from isolation, Otello offers some fairly strong progress guarantees. Specifically, Otello is deadlock and livelock-free; also, one can quantitatively bound the ratio of conflicts (cases when delegation needs to happen) to commits (cases when a control in a task reaches the task's end point) using a parameter based on the nesting depth of tasks. We note that guarantees like the last one exploit the structured nature of `async-finish` parallelism. In fact, the observation that structured parallel constructs simplify and clean up the semantics of nested isolated task-parallelism is a core contribution of this paper.

From the viewpoints of programmability, Otello's high-level isolation construct offers many benefits. When parallelizing a sequential program, the programmer only needs to pay attention to parallel decomposition without worrying about semantic issues arising from conflicting accesses performed by concurrent tasks. Isolation also helps support composability of parallel software, since an isolated library function call that creates internal tasks is guaranteed to be isolated from the task that made the call. The programmer can reason about interleavings of tasks rather than interleavings of instructions among tasks, thereby simplifying the problem of debugging parallel programs. Further, the programmer can assume a sequential consistency model when reasoning about task interleavings, even if the program executes on a platform that supports weaker memory models. Finally, the programmer can obtain all these benefits when using general nested parallel program structures.

As for performance, we have implemented Otello on top of the Habanero Java parallel programming language [8], and used this implementation to evaluate Otello on collections of nested-parallel benchmarks and transactional benchmarks. For the nested-parallel benchmarks, our results show that the low overhead of enabling 'isolation-by-default' in semantics Otello's isolation-by-default makes it a promising approach for adoption in practice. For 16-core executions, the average slowdown resulting from turning on Otello's isolation-by-default semantic across six nested-parallel benchmarks was 1.32×, with a maximum slowdown factor of 1.75×. This is significantly lower than the relative overhead of many published data-race detection algorithms [28] which only detects isolation conflicts do not enforce isolation [1]. Interestingly, one benchmark (spanning tree) showed a speedup for Otello over the default HJ implementation, due to the Otello version revealing more parallelism. For the transactional benchmarks, our results show that Otello incurs lower overhead than a state-of-the-art software transactional memory system (Deuce STM) [11]. For 16-core executions, the maximum speedup obtained for Otello relative to Deuce was 184.09×, the minimum speedup was 1.20×, and the geometric mean of the speedup across seven transactional benchmarks was 3.66×.

The rest of the paper is organized as follows. Section 2 summarizes the Habanero Java task parallel model, Aida delegated isolation model [22], and motivates the need for isolation in nested task parallelism. Section 3 introduces the core Otello model with details of its syntax, semantics, and properties. Section 4 introduces a more general programming language version of the core Otello model and describes our implementation of this language. Section 5 summarizes our experimental results. Section 6 discusses related work, and Section 7 contains our conclusions. Appendix A contains details on the formal semantics of Otello.

[1] The key reason for this difference is that a data race detector must monitor accesses at the location level to avoid false alarms, whereas isolation can be enforced at the object level.

2. Background

In this paper, we will use the Habanero Java (HJ) language [8] as a representative of nested task-parallel languages, and demonstrate how it can be extended with isolation using the Otello programming and execution model. The basic ideas in Otello of adding isolation to nested task parallelism can be applied to other nested task-parallel programming models including Cilk [5] and OpenMP 3.0 [27]. Section 2.1 provides a brief summary of HJ's `async`, `finish` and `isolated` constructs, Section 2.2 summaries Aida's delegated isolation model [22], Section 2.3 contrasts tasks in HJ with assemblies in Aida and Section 2.4 uses a simple example to illustrate the challenges that arise in ensuring isolated execution of parallel tasks.

2.1 Async, Finish and Isolated statements in HJ

The basic primitives of task parallelism relate to creation and termination of tasks. In HJ, these primitives are manifest in the `async` and `finish` statements, which were in turn derived from the X10 language [10].

async: The statement "**async** ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩ *asynchronously* (*i.e.,* before, after, or in parallel) with the remainder of the parent task. Following standard convention, we use "ancestor" to refer to the transitive closure of the parent relation on tasks. Figure 1 illustrates this concept by showing a code schema in which the parent task, T_0, uses an `async` construct to create a child task T_1. Thus, `STMT1` in task T_1 can potentially execute in parallel with `STMT2` in task T_0.

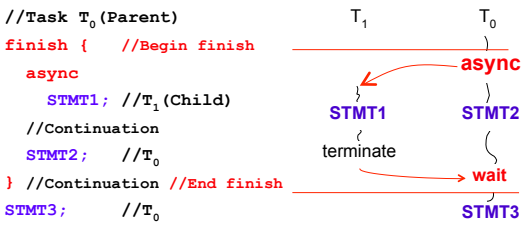

Figure 1. An example code schema with `async` and `finish` constructs

finish: `finish` is a generalized join operation. The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all `async` tasks created within ⟨*stmt*⟩ have completed, including transitively spawned tasks. Each dynamic instance T_A of an `async` task has a unique *Immediately Enclosing Finish* (IEF) instance F of a `finish` statement during program execution, where F is the innermost `finish` containing T_A [29]. The IEF of T_A must belong to an ancestor task of T_A. There is an implicit `finish` scope surrounding the body of `main()` so program execution will only end after all `async` tasks have completed.

As an example, the `finish` statement in Figure 1 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. If T_1 created a child `async` task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the `finish` scope before executing `STMT3`.

isolated: `isolated` ⟨*stmt1*⟩ denotes an isolated statement, HJ provides a "weak atomicity" semantics for isolated statements — any dynamic instance of an isolated statement is guarenteed to be performed in mutual exclusion with respect to all other potentially parallel dynamic instances of isolated statement. Thus far, the only viable approach to implement this isolation semantics with nested parallelism has been through the use of locks.

pr, guarantees that each instance of ⟨*stmt1*⟩ will be performed in mutual exclusion with all other potentially parallel interfering instances of `isolated` statement ⟨*stmt*⟩.

2.2 Aida

`Aida` [22] is a programming model which provides a notion of delegation among concurrent isolated tasks (known in Aida as assemblies). An assembly `A` is equipped with a region in the shared heap that it owns—the only objects accessed by `A` are those it owns, guaranteeing isolation. The region owned by `A` can grow or shrink flexibly—however, when A needs to own a datum owned by `B`, `A` delegates itself, as well as its owned region, to `B`. From now on, `B` has the responsibility of re-executing the task `A` set out to complete. Delegation as above is the only inter-assembly communication primitive in `Aida`.

However, `Aida` does not permit general nesting of isolated tasks. Instead, it requires all child tasks to be created at the very end of the parent task after all accesses to shared objects have completed and been committed.

2.3 Tasks vs. Assemblies

It is worth taking some time here to make clear the distinction between tasks and assemblies. A *task* is a piece of code intended to be executed asynchronously and (potentially) in parallel with other tasks. Habanero Java's *async* and Cilk's spawn are example of constructs that create tasks.

An *assembly* is a runtime concept used to implement delegated isolation. It is a collection consisting of the currently running task, all the data owned by the assembly, and all the tasks that have been delegated to this assembly through the delegation mechanism. Each assembly starts its life with a single task, and acquires data (either through acquiring individual objects or through delegation) and other tasks (through delegation) as it executes the original task and the tasks delegated to the assembly. Each assembly ends its life either when it finishes the execution of all the tasks it contains, at which point it releases all the data it owns and dies, or when it discovers a conflict with another assembly, at which point it delegates the tasks it contains and data it owns to that other assembly and then terminates.

Throughout this paper, we will use the term *task* when referring to chunks of code intended for asynchronous execution (specified by the programmer using `async`). We will

use the term *assembly* when referring to the runtime entities used to implement delegation, isolation and nesting.

2.4 Example

Though the `async` and `finish` constructs provide a general framework for expressing nested parallelism, a key challenge still remains for programmers — how to deal with *conflicts*[2] among tasks that execute in parallel?

As a simple example, consider the sequential depth-first algorithm shown in Figure 2 to compute a spanning tree of an undirected graph. It is relatively easy for a programmer to reason about parallel decompositions for such an algorithm. For instance, an examination of Figure 2 may reveal that the recursive calls to `child.visit()` could potentially execute in parallel. It is then straightforward to use `async` and `finish` constructs to obtain parallel HJ code based on this observation, as shown in Figure 3. However, when viewed as a pure HJ program, this is an incorrect algorithm because of the conflict when two tasks attempt to become the parent of the same node. This conflict arises from a potential data race between reads/writes to the same `child.parent` location by multiple tasks.

Fixing this problem is not easy. Figure 4 shows the Cilk code for a parallel depth-first-search algorithm that uses a global lock to avoid this conflict. The lock is used to implement a *critical section*. It is well known that the choice and placement of locks can be both tricky and fragile from a software engineering viewpoint since "under-locking" and "over-locking" can lead to data races or deadlock respectively. Further, the use of a single global lock can limit the amount of parallelism in the program.

Unlike Cilk, HJ uses an `isolated` construct [8] instead of locks. Though the use of `isolated` guarantees the absence of deadlocks, the programmer still has to decide where to place the `isolated` statements so as to expose sufficient parallelism without creating data races. Further, the current HJ definition does not permit new task creation in the body of an `isolated` statement. Similar constraints exist in other task parallel languages *e.g.,* X10 does not permit new tasks to be created within an `atomic` statement.

In this paper, we go beyond previous approaches and completely remove the programmer's burden of reasoning about `isolated` statements or locks. Core Otello treats all `async` tasks as being `isolated` by default (General Otello allows selected tasks, denoted by async-w for "weak async", to be non-isolated.). With isolation-by-default semantics, the code in Figure 3 will work correctly despite the potential for conflicts among accesses to the same `child.parent` location. It is the responsibility of the language implementation to ensure that any two conflicting tasks execute as though they had been performed in a sequential order that is consistent with the parallel program semantics.

[2] We use the standard definition of a conflict *viz.,* two parallel operations on a shared location such that one of the operations is a write.

```
1  void visit() {
2    for(int i = 0;
3        i < neighbors.length; i++) {
4      Node child = neighbors[i];
5      if(child.parent == null) {
6        child.parent = this;
7        child.visit();
8      }
9    }
10 } // visit()
11 . . .
12 root.visit();
13 . . .
```

Figure 2. Sequential depth-first algorithm in HJ

```
1  void visit() {
2    for(int i = 0;
3        i < neighbors.length; i++) {
4      Node child = neighbors[i];
5      if(child.parent == null) {
6        child.parent = this;
7        async child.visit();
8      }
9    }
10 } // visit()
11 . . .
12 finish root.visit();
13 . . .
```

Figure 3. Parallel algorithm in General Otello

```
1  Cilk_lockvar mylock; int** G; int* parent; int nodes;
2  cilk void dfs(int p) {
3      for (int i=0; G[p][i]!=-1; ++i) {
4          Cilk_lock(mylock);
5          int visited = (parent[G[p][i]] != -1);
6          if (!visited) parent[G[p][i]] = p;
7          Cilk_unlock(mylock);
8          if (!visited) spawn dfs(G[p][i]);
9      }  }
```

Figure 4. Cilk code for parallel depth-first-search (dfs.cilk)

3. Core Otello

In this section, we present the Otello programming and execution model. For easier exposition, we use a foundational version of Otello that focuses on the model's essential features. We call this model *Core Otello*. The main task abstraction in Otello (and Core Otello) is called an *(object) assembly* [21, 22], or a task that has *explicit* ownership of a region in the shared heap. An assembly can only access objects in the region that it owns, hence by definition, it embodies isolation. Now we present the formal syntax and semantics of the Core Otello language. Core Otello uses the same concurrency constructs as our implementation of Otello; however, the former makes several simplifying assumptions about the sequential language underlying the model. For example, statements in Core Otello do not call methods on objects, create new objects, or declare new local variables. Also, we let objects be untyped and assume that all objects are shared.

3.1 Language syntax

Otello is implemented on top of a framework of fork-join parallelism. In Core Otello as well as our implementation, the programmer creates assemblies by enclosing imperative code blocks within the construct "`async` $\{\ldots\}$". The construct `finish` $\{\ldots\}$ defines a scope at the end of which all assemblies created within the scope must join. Generalizing prior work on delegated isolation [22] as well prior work on "flat" models of `finish` statements studied in [4], we allow `finish` and `async` blocks to be arbitrarily nested.

Formally, let us assume a universe Var of variable names and a universe F of field names. The syntax of programs in Core Otello is shown in Figure 5. Here, programs are given by the nonterminal $Prog$, and the code executed by an assembly is given by the nonterminal $Block$. We assume that all variables appearing in the text of a program P are implicitly declared at the beginning of P.

Note that Core Otello imposes *some* restrictions on the structure of nested `async` and `finish` blocks. For example, the body of a `finish` block can only contain `async` statements. The body of an `async` statement can be sequential code, or a group of `finish` blocks followed by a group of `async` statements. Sequential code can only appear inside of an `async`. These restrictions are imposed in order to simplify the operational semantics of the Core Otello. A more general language called General Otello is described in Section 4.

$$
\begin{array}{lcl}
Prog & ::= & Finish \\
Finish & ::= & \texttt{finish}\,\{[Async;]^*\} \\
Async & ::= & \texttt{async}\,\{[Finish;]^*\,[Async;]^*\} \mid \texttt{async}\,\{Block\} \\
Block & ::= & Block\ Block \mid \texttt{v} := \texttt{u.f;} \mid \texttt{v.f} := \texttt{u;}
\end{array}
$$

Figure 5. Syntax of Core Otello. (Here $\texttt{u}, \texttt{v} \in Var$ and $\texttt{f} \in Fields$. For nonterminals X and Y, $[X]^*$ denotes zero or more successive occurrences of X, and $(X \mid Y)$ denotes syntax given either by X or Y.)

The source of parallelism in the constructs in Figure 5 arises from the two occurrences of $[Async;]^*$. From the Otello viewpoint, the programmer can assume that the `async` tasks in these two lists can execute in any order, can leave the determination of which tasks can safely be executed in parallel to the implementation.

3.2 Semantics: Delegation and Nesting

Conflicts between tasks (assemblies) are resolved using a runtime mechanism called *delegation*. This mechanism is a generalization of the delegation mechanism introduced in the Aida model [22], to a setting where parallel tasks can be arbitrarily nested. The core idea is as follows. Suppose an assembly A detects a conflict[3] while accessing an object O owned by a different assembly B. In that case, assembly A rolls back its effects on the heap and *delegates* itself to B, which executes it sequentially at a later point in time. Delegation could also result in the transfer of the region owned by A to a different assembly C, depending on the nesting relationship between A and B.

Delegation becomes especially challenging in the presence of nested parallelism. In this section, we sketch some of the scenarios that we must consider for delegation. We use the tree of finish scopes in Figure 6 as an example to guide our discussion. This tree represents a snapshot of a state of the program at any point in time. Each finish scope is a set of assemblies. Each assembly can contain at most one active finish scope at any point in time. Note that references to A, B, C are intended to denote general assemblies, whereas A_1, A_2, ... and F_1, F_2, ... refer to specific assemblies and finish scopes respectively in Figure 6. Note also that each assembly belongs to *exactly* one Immediately Enclosing Finish (IEF) (in our example, A_5 belongs to F_3). The *parent* of an assembly A is the assembly that created the IEF of A (in our example, the parent of A_5 is A_3).

Suppose an assembly A detects a conflict when accessing object O owned by assembly B. If assembly A and assembly B are both in the same IEF then A will delegate to B (assembly A_3 will delegate itself to assembly A_4 on Figure 6, for example). If they are not in the same IEF then either 1) assembly B is in some IEF that is nested within the IEF of assembly A, or 2) it is not.

When assembly B is in some IEF nested within the IEF of assembly A, then let C be the assembly that contains the finish scope of B and it has the same IEF as assembly A. A will then delegate itself to C. A will be executed sequentially **after** C completes, analogously to what happens in the absence of nesting. In our example, if A_2 detects a conflict with A_6, then A_2 will delegate itself to A_1.

When assembly A's IEF is nested within the IEF of B, then let C be the parent assembly of A, containing the IEF of A. A will delegate itself to C but will execute **at the closing** of its IEF. In our example, if assembly A_5 conflicts with assembly A_4, then A_5 will delegate itself to A_3, but the code of A_5 will be enqueued for execution after F_3.

When neither A nor B are nested within each other's IEFs, then let C be the parent assembly of A, containing the IEF of A. A will again delegate itself to C but will enqueue the code for A for execution **at the closing** of its IEF. In our example, if A_5 conflicts with A_7 (for example, when trying to acquire object O owned by A_7) then A_5 will delegate itself to A_3, and schedule itself for execution after F_3. Assembly A_3, when executing the code for A_5 after F_3, may again conflict with A_7 or one of its ancestors (when A_7 commits it will transfer the ownership of its objects to its parent assembly) when trying to acquire O, and delegate itself to its parent assembly (in the "cloud"), and so on, in the

[3] In the current version of Otello, we do not have any notion of read-only ownership of objects. Hence, a conflict includes scenarios where both assemblies intend to read the same object. An extension of the assembly abstraction with an separate category of read-only ownership is however easy to define, and will be implemented in future versions of Otello.

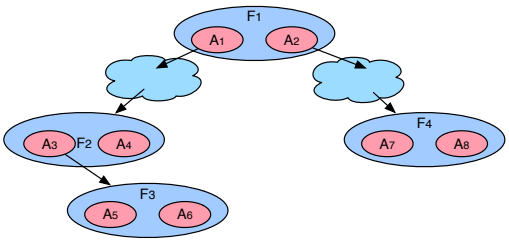

Figure 6. An example of a finish tree. F_2 and F_4 are descendants of F_1, with clouds representing arbitrary levels of nesting of finish scopes.

worst case resulting in A_1 delegating itself to A_2. However, if A_2 (and its whole subtree of assemblies and finish scopes, including A_7) finishes its execution before A_3 tries acquiring O again while executing the code for A_5, then A_3 will be able to acquire O and proceed with the execution.

Now we sketch the operational semantics of Core Otello. A more detailed semantics is presented in Appendix A.

The central data structure in Core Otello is the shared-memory *heap*, which maintains the state of all shared mutable data accessed by a program. We abstractly view a heap as a directed graph whose nodes are objects, and edges are pointers labeled with field names.

Now consider a piece of code $\texttt{K} = \texttt{S}_1; \ldots; \texttt{S}_\texttt{m}$ that can be executed by an assembly, where each $\texttt{S}_\texttt{i}$ is either an assignment, or a nested `async` or `finish`-block. A *closure* of `K` is a data structure that contains information about: (1) the part of `K` that still remains to be executed, and (2) the mapping of variables used in `K` to objects in the heap.

Now, let us assume a universe of assembly IDs. Consider an assembly with ID A that we are currently executing, and let G be the heap that A accesses. An *assembly state* of A is a data structure that contains information about: (1) the closure that A is currently executing; (2) the delegation queue of A—i.e., a list of closures that have been delegated to A, and will be executed by A in sequence once the current closure finishes executing; and (3) a *region*—a set of nodesowned by G.

The state of an entire Core Otello program at any given time (called a *concurrent state*) is defined as a tree-shaped structure where a node is an ID for a specific finish-scope, and each node F is associated with a collection of assembly IDs $\mathcal{T}(F)$. Intuitively, the assemblies in $\mathcal{T}(F)$ have F as their immediately enclosing finish.

Unlike in usual trees, edges here go from assembly IDs (associated with tree nodes) to tree nodes. There is a tree edge from an assembly ID A to a tree node F if the finish-scope represented by F is nested within the asynchronous task represented by A. At any point of an execution, each assembly ID is associated with a specific assembly state. These assembly states must be such that the regions owned by the corresponding assembly in them are disjoint. This guarantees that an object in our model is owned by at most one assembly. The objects in G that are not owned by any assembly are said to be *free*.

The dynamics of a Core Otello program is defined by a set of transitions between concurrent states. Most of these transitions select an assembly from the current concurrent state and execute the next statement in it. *For an assembly to be scheduled for execution, it must be a leaf in the tree that is the current concurrent state*. The need for delegation arises when an assembly tries to access an object owned by a different assembly.

Specifically, we sketch the more interesting classes of transitions:

- **Same-level conflict:** We have a class of transitions that define delegation among assemblies that are siblings of each other in a concurrent state. Let A and B be assemblies, and let Q_A and Q_B be the list of closures that A and B obligated to execute, respectively. (By convention, let us assume that the closures that A and B are *currently* executing are the first elements of these lists.)

  Let the first element of Q_A be χ; by our convention, this closure is currently under execution. Now suppose χ reads or writes an object u that is currently owned by the assembly B (i.e., a *conflict* happens). Consequently, A must now delegate its work and owned region to B. After the rule fires, the state of B becomes such that: (1) Its owned region includes all objects owned by A or B before the rule fired; (2) Its delegation queue is obtained by appending Q_A at the end of Q_B.

  One important issue is that A may have modified certain objects in its region while it was executing χ. In this case, before delegation, the runtime *rolls back* the effect of the code already executed by χ. However, because A is required to be a leaf in the tree that constitutes the concurrent state, and also because of the syntactic restrictions in Core Otello, this code is solely a collection of heap updates—it does not include any `async` or `finish` statements. The implementation of this rollback operation is therefore straightforward.

- **Conflict below:** We have a class of transitions capturing scenarios when an assembly A conflicts with an assembly B, and there is an assembly C that is a sibling of A and an ancestor of the node containing B. In this case, A delegates to C.

- **Conflict with ancestor:** We have a class of transitions that handle the case when A tries to access an object u owned by an ancestor B in the concurrent state. Because B is not a leaf node in the current finish-tree, it is currently "suspended"; therefore, A can safely "steal" u from B.

- **Conflict with unrelated:** These transitions handle the remaining conflict scenarios. Here, if $A \in \mathcal{T}(F)$ conflicts

with B, then A transfers its owned region to the parent C of the node F. The code executed by A is now to be executed after all the assemblies in $\mathcal{T}(F)$ finish executing, and is put in the closure queue for F.

- **Async creation:** These transitions define the semantics of assembly creation. The rule creates a new assembly with an empty set of owned objects. The assembly belongs to the same tree node (finish-scope) as the assembly that created it.
- **Finish:** These transitions define the semantics of "`finish`"-statements executed by an assembly A. Here a new tree node F' is created; A is the parent of F', and $\mathcal{T}(F') = \emptyset$. The code inside the `finish`-block is transferred to the closure queue of F.

3.3 Properties of Core Otello

Isolation The property of isolation demands that a concurrent task read or write shared-memory objects without interference from other tasks. In Core Otello, an assembly can only read or write objects in its own region; also, if A delegates work to B, the ability of B to read or write its objects is not compromised. Therefore, Core Otello guarantees isolation.

Deadlock-freedom There are only two synchronization operations in Core Otello: scoped joining of assemblies and delegation. As the former operation does not depend on data at all, it clearly does not introduce deadlocks (The tree structure also ensures that deadlock is not possible due to join operations.). As for delegation, the only plausible deadlock scenario involving *two* assemblies is the following: "Assembly A tries to delegate to assembly B, B tries to delegate to A, and neither can progress." This scenario, however, is impossible in Otello. If A and B try to simultaneously delegate to each other, then one of the two requests (let us say the one from B) will be nondeterministically selected and honored. This can lead to two outcomes: (a) The ownership of all objects owned by B would be transferred to A; or (b) The ownership of all objects owned by B would be transferred to the parent B' of the finish-tree node where B resides. In the former case, the request from A is no longer a conflict—the request from A to access u will succeed, and A will be able to progress. In the latter case, A may conflict with B', in which case the ownership of u may be transferred to the parent B'' of the finish-scope of B', and so on. However, the number of times such conflicts over u may occur is bounded by the number of hops between the root of the finish-tree and the node to which B belongs. Formally, let us define the *finish-depth* $Depth(P)$ of a program P inductively as follows:

- If $P = \texttt{finish}\{P_1 \ \ldots \ P_n\}$, where each P_i is an `async`-block, then $Depth(P) = 1 + \max_i Depth(P_i)$
- If P is a *Block*, then $Depth(P) = 0$
- If $P = \texttt{async}\{Q_1 \ \ldots \ Q_m \ P_1 \ \ldots \ P_n\}$, where each Q_i is a `finish`-block and each P_i is an `async`-block, then $Depth(P) = \max(\max_i Depth(P_i), \max_i Depth(Q_i))$.

In the above scenario, the maximum number of conflicts between A and other assemblies $B, B', B'', \ldots$ over u is bounded by $Depth(P)$. It is easy to generalize the above argument to cyclic deadlocks among n assemblies.

Livelock-freedom In Core Otello, two assemblies A and B would be livelocked if they constantly try to delegate to each other, none of them progressing. As an analogy, there is always a non-zero probability that such a livelock scenario may occur in a transactional memory system with repeated rollback-retry operations. However, in such a scenario, Otello would destroy one of the two assemblies, delegating its work—the other assembly would then be able to progress.

Bound on conflicts/commit ratio Finally, a key property of Core Otello is that in any execution of a program P, the number of conflicts (number of applications of the delegation rules in Figure 11) is bounded by $Depth(P)$ times the number of commits. [4] As $Depth(P)$ is small in practice, this property works as a performance guarantee in high contention-scenarios, where, in many state-of-the-art transactional memory systems, there may possibly be an unbounded number of aborts and too few commits. Naturally, our guarantee comes at a cost; due to delegation, some tasks in Otello may be performed sequentially, and in certain situations, this loss of parallelism may be a limitation.

4. Implementation

4.1 General Otello

Core Otello is a simple and elegant language that can be implemented using the techniques described below in Section 4.3, and that provides important correctness guarantees. However, in practice, such a core model is too minimalistic for programmers. Given this, we embed our core model in a fuller-featured parallel programming language called *General Otello*.

General Otello removes many of the syntactic restrictions present in Core Otello—for example, that *Block* can only appear inside an *Async*, and that *Finish* can only appear at the beginning of an *Async*. Also, General Otello permits non-isolated parallel tasks in addition to isolated ones (assemblies). As before, the latter are demarcated in the source code by keyword "`async`". A block of code executed as a non-isolated task is marked by the keyword "`async-w`" (read as "*weak* async"). Such non-isolated tasks allow programmers or compilers to completely eliminate the

[4] We note that commits are somewhat tricky to define in the setting of nested parallelism. This is because a task that has been "committed" may be have to be aborted subsequently because the parent of the task is aborted. In this paper, we count each time control reaches the end point of a task to be a commit.

(small, but not negligible) overhead of delegated isolation in cases where the programmer or compiler knows in advance that the tasks will be accessing completely disjoint parts of the data. However, this extra efficiency comes at the cost of some guarantees. General Otello does not make any guarantees about isolation between "`async-w`" tasks or isolation between "`async-w`" and "`async`" tasks; it relies completely on the underlying Habanero Java implementation to execute "`async-w`" tasks. In other words, it implements the *weak atomicity* semantics [24] for programs containing both "`async-w`" and "`async`". In relationship to current HJ, General Otello's "async-w" and "async" constructs can be viewed as equivalent to "async" and "async isolated" constructs in HJ. Further, HJ's "isolated" construct can be viewed as equivalent to "finish async" in General Otello.

The syntax of General Otello shown in Figure 7.

$$
\begin{array}{lll}
Prog & ::= & \texttt{finish}\,\{Stmt\} \\
Stmt & ::= & \texttt{async}\,\{Stmt\} \mid Stmt; Stmt \\
 & & \mid \texttt{async-w}\,\{Stmt\} \\
 & & \mid \texttt{finish}\{Stmt\} \mid Block
\end{array}
$$

Figure 7. General Otello. $Block$ is arbitrary sequential Java code

4.2 The General Otello compiler

Now we present a set of compiler transformations (Figure 8) for translating programs written in General Otello into a form that simplifies the runtime implementation described below in Section 4.3. While most of these transformations are self-explanatory, transformation 5 deserves special attention. This transformation moves spawning of an $async\{S_1\}$ that is followed by a statement S_2 to the end of the enclosing async by capturing the local context of S_1 (all the local variables that are free in S_1) at the original point of the spawn (coded as $C_1 = C(S_1)$), then spawning the async at the end of the enclosing async with the free local variables substituted with the corresponding variables from the captured context (coded as $S_{C=C_1}(S_1)$). Note that this may result in some loss of parallelism, since this transformation always moves an async to the end of the enclosing async, even if S_1 does not conflict with S_2 and the two could potentially run in parallel. Transformation 5 is the only one on Figure 8 that has a potential loss of parallelism, all the others preserve the amount of work that is potentially done in parallel in General Otello.

The reason for this transformation is implementation-dependent. Since in our implementation, the assembly that discovers the conflict always delegates itself to the assembly that already owns the object, we cannot allow the possibility of code in S_2 discovering a conflict with S_1 and attempting to delegate the outer $async$ to $async\{S_2\}$, since the outer $async$ contains the spawn of $async\{S_2\}$. A different implementation could always decide to delegate inner $async$ to the outer $async$ whenever a conflict occurs between the two, but would require a complex synchronization mechanism between the two assemblies as the outer $async$ would have to notify the inner $async$ that it needs to abort, rollback and delegate itself to the outer $async$, and than wait for that to happen before proceeding. Another approach is a compiler analysis that will determine if S_1 and S_2 are commutative and allow them to run in parallel, which is a subject for future work.

The syntax of General Otello in Figure 7 does not include control flow. To allow arbitrary control flow within a finish block that contains arbitrary nesting of finishes and asyncs we have implemented a simple runtime strategy. Whenever an $async$ is created inside of arbitrary control flow, the context is captured at the point of creation, but the actual spawning of the async is delayed until the end of the enclosing `finish` scope. At first glance this may look even more restrictive than the transformation 5 from Figure 8 that only moves the $async$ to the end of the outer $async$ and not the enclosing `finish` scope, but the end result will be the same, since the resulting code will not have any effectual code between the outer async and the end of the enclosing finish.

Combined, the transformations from Figure 8 and the runtime strategy in this section jointly ensure that all $async$s that are created within a `finish` scope are spawned at the end of that `finish` scope.

4.3 The General Otello runtime

In this section we present some of the key mechanisms in our implementation of the General Otello execution model.

Rollback log. When an assembly commits, rollback logs for all modified objects are merged with the rollback logs of its parent. These rollback logs are needed in case the parent assembly has to be rolled back, requiring the rollback of all its effects, including those caused by nested asyncs. If both the parent and the child have a rollback log for the same object, the child's log is discarded.

Ownership. When an assembly commits, the ownership of all the objects it acquired is also transferred to the parent assembly. This is to ensure proper nesting semantics, so that no other assembly can observe the state committed by the child assembly until the parent assembly commits. Other assemblies running in parallel to the parent assembly will still conflict with it if they try to acquire an object that was modified and committed by the child assembly.

Delegated Task Queues. There are two "types" of delegated task queues.

1. *Assembly Queues*. These are the queues that contain delegated tasks that are to be executed sequentially after the current task commits, as in Aida [22]. An assembly delegates itself by transferring the ownership to the target assembly, and appending its *Assembly Queue* to the target *Assembly Queue*.

1. $\mathcal{T}(finish\{S_1\}) \rightarrow finish\{\mathcal{T}_A(S_1)\}$
2. $\mathcal{T}_A(async\{S_1\}; S_2) \rightarrow async\{\mathcal{T}_{FA}(S_1)\}; \mathcal{T}_A(S_2)$
3. $\mathcal{T}_A(B_1; S_2) \rightarrow async\{finish\{async\{B_1\}\}; \mathcal{T}_{FA}(S_2)\}$
4. $\mathcal{T}_A(finish\{S_1\}; S_2) \rightarrow async\{finish\{\mathcal{T}_A(S_1)\}; \mathcal{T}_{FA}(S_2)\};$
5. $\mathcal{T}_{FA}(async\{S_1\}; S_2) \rightarrow finish\{async\{C_1 = C(S_1)\}\}; \mathcal{T}_{FA}(S_2); async\{\mathcal{T}_{FA}(S_{C=C_1}(S_1))\}$
6. $\mathcal{T}_{FA}(finish\{S_1\}; S_2) \rightarrow finish\{\mathcal{T}_A(S_1)\}; \mathcal{T}_{FA}(S_2)$
7. $\mathcal{T}_{FA}(B_1; S_2) \rightarrow finish\{async\{B_1\}\}; \mathcal{T}_{FA}(S_2)$

Figure 8. Compiler transformations to convert General Otello programs into Core Otello

2. *Finish Queues*. New in Otello are queues of delegated tasks that are to be executed sequentially at the end of a `finish` scope. Tasks might end up in this queue when a conflict forces an assembly to delegate itself to the parent assembly, as described above. The assembly delegates itself by transferring the ownership and rollback logs to the parent assembly, but transferring its assembly queue to the *Finish Queue* of the immediately enclosing finish.

Task Queues. Every `finish` scope contains a queue for assemblies spawned inside of it. This queue is different from the *Finish Queue*. If an assembly cannot be moved to the end of its outer assembly using transformation 5 from Figure 8 (i.e. if the spawning of the assembly is inside of some control flow), then the assembly is moved to the *Task Queue* of the enclosing finish. When the execution reaches the end of the finish, it first spawns in parallel all the assemblies from its *Task Queue*. These assemblies run in parallel and in isolation. If any of them discover a conflict, they may delegate themselves to their parent assembly and transfer their code and their assembly queue to the corresponding *Finish Queue*.

Object acquisition. When an assembly attempts to acquire an object, the object has to fall into one of the following categories:

1. *Free*. These are objects that are not owned be any assembly; i.e. their owner pointer is either null or points to a *dead* owner.
2. *Owned by ancestor*. The object is owned by an ancestor assembly. Ancestor assembly has an active `finish` scope with which the current assembly synchronizes.
3. *Owned by a sibling*. The object is owned by an assembly that belongs to the same `finish` scope as the current assembly.
4. *Owned* by an assembly that is not an ancestor or a sibling.

When an assembly requests an object that is either *free* or *owned by ancestor* the request is successful and the current assembly acquires the ownership of the object. Note that it is safe to "inherit" an object that is owned by an ancestor, as by design the ancestor is **not** executing meaningful code (code that accesses ownable objects) at that time. Also note that only one assembly may inherit this specific object, as all subsequent acquisition attempts by other assemblies will fall into the *owned by ancestor* or *owned by sibling* category. When the assembly finishes, it transfers the ownership to its parent, which will (eventually) result in the object ownership being transferred back to ancestor from which it was inherited.

When the object is *owned by a sibling* or *owned*, that constitutes a conflict. If the object is *owned by a sibling*, the current assembly delegates itself to the sibling that owns the object. The current assembly will rollback, transfer its ownerships to the sibling, and add its assembly queue to the end of the sibling's assembly queue.

Finally if the requested object is *owned* then the current assembly will delegate itself to its parent assembly, performing the following steps:

1. First the assembly rolls back its changes.
2. The assembly transfers the ownership of all its objects to its parent.
3. The assembly appends its assembly queue to the end of the corresponding *finish queue* of its parent.
4. The assembly synchronizes with the finish (as if ended normally) and ends.

Completing the assembly When a nested assembly finishes its execution (including all the assemblies in its assembly queue) it transfers the ownership of all its objects to its parent and **merges** its rollback log with the parent's rollback log, ensuring that a parent rollback includes a rollback of the effects of nested assemblies. The **merge** is performed so that the parent's rollback information supersedes the child; if both the parent and the child have rollback information for the same object, the child's rollback information for that object is discarded, otherwise the child's rollback information for the object is added to the parent's rollback log.

State backup and recover Otello uses a per-object based log mechanism that backs up the entire object when it is acquired by an assembly. For arrays, Otello backs up a chunk of the array or the entire array when any of the array elements is acquired by an assembly.

Most of the other implementation details are similar to what has been done in Aida [22], including a Union-Find data structure for quick test and transfer of object ownership,

and the nonblocking implementation of the delegation mechanism. However, as mentioned earlier, the Aida approach does not guarantee isolation with nested task parallelism, which is the main motivation for Otello.

It may be tempting to implement Otello using locks for conflict detection. However, such an implementation will likely incur prohibitively large overhead. In addition, without using reentrant locks, the implementation would also have to handle potential deadlocks. However, this is a direction that may be worth pursuing in the future if lightweight reentrant locks become available with hardware support.

5. Evaluation

In this section, we present an experimental evaluation of the Otello programming model implemented as an extension to the HJ compilation and runtime framework [8]. We used this implementation to evaluate Otello on collections of nested-parallel benchmarks and transactional benchmarks.

5.1 Experimental Setup

Our experimental results were obtained on a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory, running Red Hat Linux (RHEL 5) and Sun JDK 1.6 (64-bit version) (with a heap size was set to 12GB). The software transaction memory (STM) system used for some of the performance comparisons is version 1.3.0 of the Deuce STM [11] which implements the TL2 [12] algorithm.

We chose six benchmarks for the nested task-parallel collection. One of these benchmarks is the `spanning tree` example discussed in Section 2.4. The remaining five are HJ ports of the OpenMP 3.0 BOTS benchmarks [13] `health`, `floorplan`, `strassen`, `fft` and `nqueens`. These HJ ports are not new to this paper; they have also been used in earlier performance evaluation, e.g. [3] and [26]. Also, the HJ versions of these benchmarks are fundamentally the same as the OpenMP versions; the primary change (in addition to translating C code to Java code) is that the OpenMP 3.0 `task`, `taskwait` and `critical` directives were replaced by `async`, `finish` and `isolated` statements in HJ, respectively. The Otello versions of these benchmarks are identical to the HJ versions with one minor syntactic change: eliminating the `isolated` keyword when moving from HJ to Otello, since all async's were assumed to be isolated by default in the Otello version.

To understand precisely how the Otello and HJ versions of these codes differ, consider the code listed below, which shows the HJ version of `nqueens` benchmark:

```
1  void nqueens_kernel(final byte n, final byte j,
2                      final byte[] a, final int depth) {
3    if (n == j) {
4      isolated { // eliminated in Otello version
5        this.total_count ++;
6      }
7      return;
8    }
9    finish {
10     for (byte i = (byte) 0; i < n; i++) {
11       final byte i0 = i;
12       async {
13         final byte[] b = new byte[j+1];
14         System.arraycopy(a, 0, b, 0, j);
15         b[j] = i0;
16         if (ok((byte)(j+1), b))
17           nqueens_kernel(n,(byte)(j+1),b,depth);
18       }
19     }
20   }
21 }
```

The `isolated` statement in line 4 protects shared variable `this.total_count`. HJ currently uses a global lock to implement isolation. In the Otello version, the `isolated` keyword in line 4 is eliminated, since all async tasks run in an isolated mode by default in Otello.

For the transactional benchmarks, we used seven benchmarks from JSTAMP [16] (a Java port of the STAMP benchmark suite [25]), implemented in the Deuce STM [11]. The seven benchmarks are `Bayes`, `Genome`, `Vacation`, `SSCA2`, `Intruder`, `KMeans` and `Labyrinth3D` [5].

5.2 Comparing Otello with standard HJ

To evaluate the runtime overhead of the isolation-by-default mechanisms in Otello, we compared the execution times of isolated-by-default Otello and the explicit-isolated-as-needed HJ versions of the six nested task-parallel benchmarks. Figure 9 presents results for these benchmarks run using different numbers of worker threads. In general, Otello shows better scalability than `HJ` but also fairly high overhead when ran on a small number of threads. For 16-core executions, the maximum slowdown resulting from turning on Otello's isolation-by-default semantic was 1.75×, and the geometric mean of the slowdown across six nested-parallel benchmarks was 1.32×. Interestingly, the minimum slowdown was 0.76× *i.e.,* for this benchmark (`spanning tree`) the execution time with Otello's isolation-by-default approach was faster than that of the original HJ program due to the uncovering of more parallelism through speculative parallelism with delegated isolation.

In summary, for the nested-parallel benchmarks, our results show that Otello can achieve comparable scalability to standard HJ programs (without isolation-by-default semantic), with a relative overhead that is much lower than that (for example) of many published data-race detection algorithms [28].

5.3 Comparing Otello with Deuce STM

In this section, we compare the performance of Otello with the non-nested transactional parallelism approach in Deuce STM. Figure 10 presents the performance comparison between Otello and Deuce STM, a state-of-the-art Java-based STM implementation. Otello offers better performance and

[5] We omitted the `MatrixMul`, and `Yada` benchmarks, since `MatrixMul` has a simple critical section that can be implemented as a reduction and `Yada` exhibits a bug when running with multiple Java threads.

Benchmark Suite.	Name	Has Isolation	Input Size
BOTS	**strassen**	No	nodes: 1,000,000
	fft	No	elements: 1,000,000
	health	Yes	largest size
	floorplan	Yes	size: 15
	nqueens	Yes	12
HJ Bench	**spanning tree**	Yes	nodes: 100,000 neighbors: 100
JSTAMP	**Bayes**	Yes	vars: 32 records: 4096 numbers: 10 percent: 40% edges: 8 inserts: 2
	Genome	Yes	gene length: 16,384 segment length: 64 segments: 1,677,216
	Vacation	Yes	clients: 16 queries: 90 relations: 1,048,576 transactions; 4,194,304
	SSCA2	Yes	scale: 20 partial edges: 3 max path length: 3 incr edges: 1
	Intruder	Yes	attacks: 10 length: 16 numbers: 1,000,000
	KMeans	Yes	nodes: 65536 clusters: 40
	Labyrinth3D	Yes	128 × 128 grid

Table 1. Benchmarks and relevant input size.

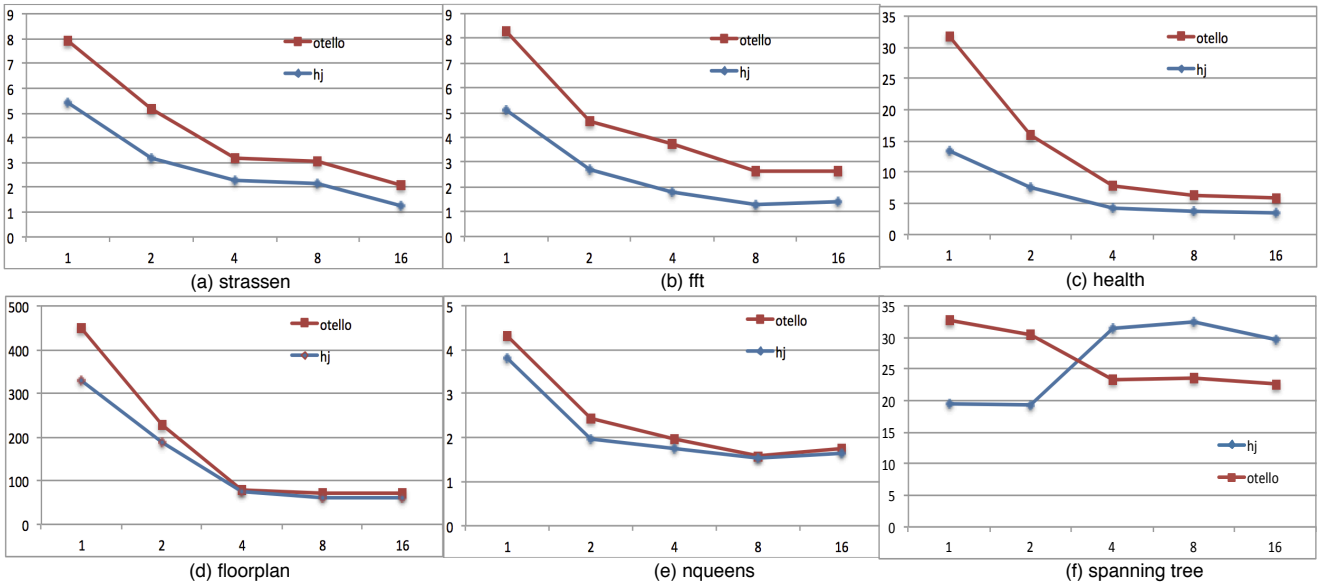

Figure 9. Performance comparison between Otello and HJ. "hj": HJ implementation; "Otello": Otello implementation; X-axis: number of parallel threads (cores) used; Y-axis: execution time in seconds.

scalability than Deuce STM across all benchmarks with one exception (`Bayes` on 8 threads).

The performance difference is especially evident in `KMeans` and `Labyrinth3D`, which are array-intensive programs, where Deuce STM's contention management backs up all array elements, while the Otello implementation performs backup of a chunk of the array, rather than a per-element backup (see Section 4). For 16-core executions, the maximum speedup obtained for Otello relative to Deuce was 184.09×, the minimum speedup was 1.20×, and the geometric mean of the slowdowns was 3.66×.

6. Related Work

We use Table 2 to guide the discussion in this section. This table qualitatively classifies programming models according to the following attributes:

- *Programmability/expressiveness:* how easy is it to express a wide range of parallel programming patterns in the model?
- *Expertise:* how much expertise in concurrent programming does the programmer need?
- *Correctness guarantees:* does the model provide correctness guarantees such as deadlock-freedom, livelock-freedom, and progress guarantees?
- *Scalability:* how well does performance scale with an increase in the number of processor cores and hardware threads?
- *Overhead:* how much overhead does the model impose relative to a sequential implementation?

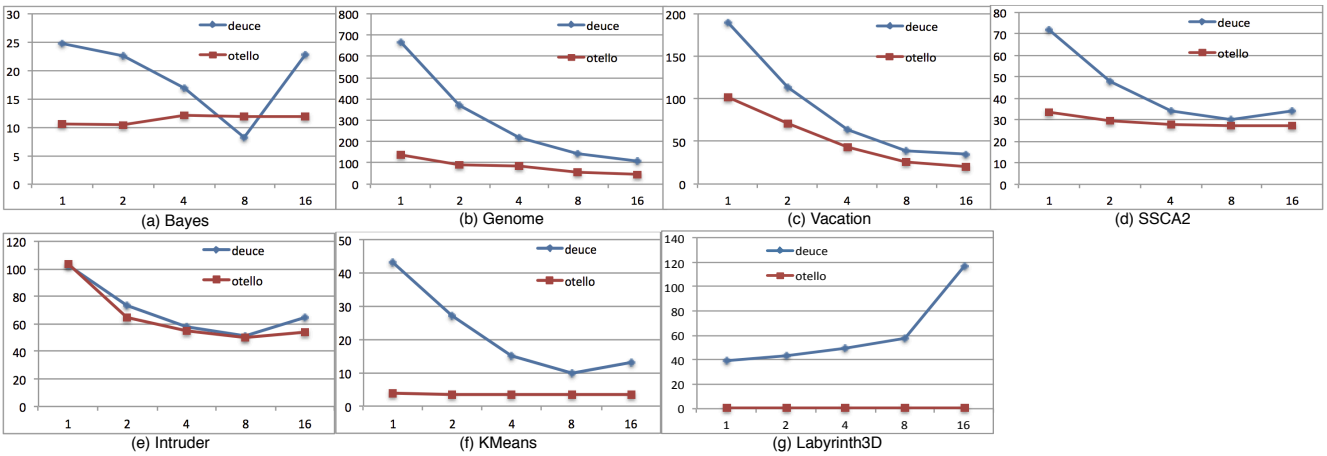

Figure 10. Performance comparison between Otello and STM for JSTAMP benchmarks. "deuce": deuce STM implementation; "otello": Otello implementation; X-axis: number of parallel threads (cores) used; Y-axis: execution time in seconds.

Parallel prog. model	**Expressiveness** (higher is better)	**Expertise** (lower is better)	**Safety** (higher is better)	**Scalability** (higher is better)	**Overhead** (lower is better)
STM [15]	High	Very Low	implementation dependent	Low	High
Java with fine-grained locking	High	High	Very low	High	Low
Cilk reducers [5]	Medium	Medium	Low	High	Low
Chorus [21]	Medium	Low	High	Medium	Medium
Galois [7, 18, 19]	Medium	Low	Medium	High	Low
Aida [22]	High	Low	High	High	Low
Otello [this paper]	High	Very Low	High	High	Low

Table 2. Comparison of several parallel programming models.

For simplicity, we focus our comparison on current approaches to parallel programming with isolation and nesting.

Otello provides a high-level minimalistic programming model similar to *Transactional Memory* [15], with a single construct (`async`) to define blocks of code to be executed concurrently and in isolation. *All* parallel tasks in Core Otello are isolated by default, the programmer only needs to worry about what tasks should run in parallel, and all the conflict detection and resolution is done automatically by the implementation. Otello guarantees livelock-freedom, unlike transactional memory where livelock freedom is probabilistic and implementation-dependent.

The *assemblies* in *Chorus* [21] and *Aida* [22] provide a high-level mechanism for irregular data parallelism and have been the main inspiration for assemblies in Otello. The Chorus assemblies, however, are restricted to *cautious applications* [23], while neither Chorus nor Aida support nested parallelism.

The delegation mechanism in *Aida* [22] is the main inspiration for the delegation mechanism in Otello. The delegation in Aida, however, only considers the flat model, where all tasks belong to the same finish scope. The main contribution of Otello is an execution model that performs task delegation in the presence of arbitrary finish and isolated task nesting.

There has also been a lot of work in the transactional memory community on enabling different forms of nesting of transactions; however, very few efforts allow the integration of nested transactions with task parallelism. Agrawal et al. [1] propose a design to support Cilk-style nested parallelism inside transactions, but they do not have an implementation or evaluation. OpenTM [2] allows transactions within OpenMP constructs, but does not allow nested parallelism within transactions.

NePaLTM [30] is the only TM system we are aware of that integrates nested transactions with task parallelism (OpenMP). Otello is built on top of, and adds nested isolated tasks to HJ, a more flexible and dynamic programming model that allows creation of more general task graphs than OpenMP. None of the STM systems mentioned use delegation as the mechanism for conflict resolution, nor do they offer an upper bound on conflict-to-commit ratio in heavily-congested scenario as Otello does.

Programming models such as OpenMP [9] and Cilk [5] provide efficient support for *reductions* in deterministic parallel programs. However, those constructs (or the constructs

offered by DPJ [6]) are not applicable to the nondeterministic, irregular parallel programs supported by Otello.

Orthogonal to Otello is prior work on data-race detection [14]. Otello gives the option of proceeding (with well-defined semantics) in the presence of data races, while the other models do not. If there is a desire to warn the user about interference among parallel accesses to certain classes of objects, past work on data-race detection can be applied to Otello for that purpose.

Herlihy and Koskinen proposed a form of checkpointing and continuations [17] to handle partial commits and rollback of transactions. We implemented a similar mechanism in Otello as an optimization to allow partial commits of long-running asyncs to avoid superficial conflicts.

7. Conclusions and Future Work

Isolation has long been one of the most basic concerns in parallel programming. Despite the recent attention paid to software and hardware approaches to transactional memory, there is no transactional memory solution in past work that guarantees isolation by default for all code while supporting nested parallelism. In this paper, we presented a programming and execution model called Otello that supports an isolation-by-default approach. In our model, the programmer only exposes parallelism by creating and synchronizing parallel tasks (using `async` and `finish` constructs), leaving it to the Otello compiler and runtime system to ensure isolation among parallel tasks. Otello's programming model offers several high-level correctness guarantees while requiring minimal programmer expertise. The practicality of Otello stems from the novel compiler and runtime techniques presented in this paper. We used these techniques to implement Otello on top of the Habanero Java parallel programming language, and to evaluate Otello on collections of nested-parallel benchmarks and transactional benchmarks. For the nested-parallel benchmarks, the maximum slowdown resulting from turning on Otello's isolation-by-default support was 1.75×, and the geometric mean of the slowdown across six nested-parallel benchmarks was 1.32× which is significantly lower than the relative overhead of many published data-race detection algorithms. For the transactional benchmarks, our results show that Otello incurs lower overhead than a state-of-the-art software transactional memory system (Deuce STM). For 16-core executions, the maximum speedup obtained for Otello relative to Deuce was 184.09×, the minimum speedup was 1.20×, and the geometric mean of the speedup across seven transactional benchmarks was 3.66×.

Future work includes developing compiler and runtime optimizations to further reduce the overhead of ensuring isolation by default, exploration of new strategies for delegation, and supporting greater integration of Otello with additional HJ constructs for task parallelism, including futures, data-driven tasks, and phasers [29].

Acknowledgments

This work was supported in part by NSF award CCF-0964520. Any opinions, ndings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. We would like to thank members of the Habanero group at Rice for valuable discussions related to this work, and contributions to the Habanero Java infrastructure used in this research. We are grateful to the anonymous reviewers for their comments and suggestions. Finally, we would like to thank Keith Cooper for providing access to the Xeon system used to obtain the performance results reported in this paper.

References

[1] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *Proceedings of PPoPP'08*, Salt Lake City, UT, USA, pages 163-174.

[2] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM transactional application programming interface. In *Proceedings of PACT'07*, pages 376387.

[3] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Interprocedural strength reduction of critical sections in explicity-parallel programs. In *Proceedings of PACT'13*, 2013.

[4] Ganesh Bikshandi, Jose G. Castanos, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, and Tong Wen. Efficient, portable implementation of asynchronous multi-place programs. In *Proceedings of PPoPP'09*.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work-stealing. In *Proceedings of FOCS'94*, 1994.

[6] Robert L. Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of POPL'11*, 2011.

[7] Martin Burtscher, Milind Kulkarni, Dimitrios Prountzos, and Keshav Pingali. On the scalability of an automatically parallelized irregular application. In *Proceedings of LCPC'08*, pages 109 123.

[8] Vincent Cave , Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the New Adventures of Old X10. In *Proceedings of PPPJ'11*, 2011.

[9] Robit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. Parallel programming in OpenMP. *Morgan Kaufmann Publishers Inc.*, 2001.

[10] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA'05*, New York, NY, USA, 2005.

[11] Deuce STM - Java Software Transactional Memory. http://www.deucestm.org/.

[12] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of DISC'06*, pages 194208, 2006.

[13] Alejandro Duran et al. Barcelona OpenMP Tasks Suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of ICPP'09*, 2009.

[14] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of PLDI'09*, pages 121 133. ACM, 2009.

[15] Tim Harris, James R. Larus, and Ravi Rajwar. Transactional Memory, 2nd Edition. *Morgan and Claypool*, 2010.

[16] JSTAMP. Jstamp. http://demsky.eecs.uci.edu/software.php.

[17] Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In *Proceedings of SPAA'08*, pages 160168, jun 2008.

[18] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How much parallelism is there in irregular applications? In *Proceedings of PPoPP'09*, pages 314.

[19] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of PLDI'07*, pages 211222, New York, NY, USA, 2007. ACM.

[20] J. Larus and R. Rajwar. Transactional Memory. *Morgan and Claypool*, 2006.

[21] Roberto Lublinerman, Swarat Chaudhuri, and Pavol Cerny. Parallel programming with object assemblies. In *Proceedings of OOPSLA'09*, New York, NY, USA, 2009. ACM.

[22] Roberto Lublinerman, Jisheng Zhao, Zoran Budimlic , Swarat Chaudhuri, and Vivek Sarkar. Delegated isolation. In *Proceedings of OOPSLA'11*, NY, USA, 2011.

[23] Mario Me ndez-Loj, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingalil. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of PPoPP'10*, pages 3 14, New York, NY, USA, 2010. ACM.

[24] Vijay Menon, Ali reza Adl-tabatabai, Steven Balensiefer, Richard L. Hudson, Adam Welc, Tatiana Shpeisman, and Bratin Saha. Practical weak-atomicity semantics for java stm. In *Proceedings of ACM Symposium on Parallellism in Algorithms and Architectures*, 2008.

[25] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *Proceedings of IISWC'08*, pages 3546, 2008.

[26] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. A transformation framework for optimizing task- parallel programs. *ACM Trans. Program. Lang. Syst., 35(1):3*, 2013.

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

[28] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of PLDI'12*, pages 531 542.

[29] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of ICS'08*, pages 277 288.

[30] Haris Volos, Adam Welc, Ali-reza Adl-tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. NePaLTM: Design and implementation of nested parallelism for transactional memory systems. In *Proceedings of ECOOP'09*, jun 2009.

Appendix A: Formal Semantics of Otello

Heaps The central data structure in Core Otello is the shared-memory *heap*, which maintains the state of all shared mutable data accessed by a program. We abstractly view a heap as a directed graph whose nodes are objects, and edges are pointers labeled with field names. A *region* R in a heap G is a subset of the nodes of G.

Assembly states Let us first define a *variable state* of program P over a heap G. Such a state is a function μ that maps the variables of P either to objects in G, or to the symbol `null`. We emphasize that an assembly is not required to own the objects to which its variables point. However, once it tries to read from or write to such an object u, it acquires ownership of u, or alternately delegates to the owner of u.

Consider the code $\mathtt{K} = \mathtt{S_1}; \ldots; \mathtt{S_m}$ that can be executed by an assembly, where each $\mathtt{S_i}$ is either an assignment, or a nested `async` or `finish`-block. We interpret such a block as a list $[\mathtt{S_1}, \ldots, \mathtt{S_m}]$. A *closure* of K is a triple $\langle \mathtt{K_1}, \mathtt{K_2}, \mu \rangle$, where $\mathtt{K_1}$ and $\mathtt{K_2}$ are the sublists of K that, respectively, have not and have been executed (we have $\mathtt{K} = \mathtt{K_2}; \mathtt{K_1}$), and μ is a variable state of P.

An *assembly state* over a heap G is a tuple of the form $N = \langle \langle \mathtt{K_1}, \mathtt{K_2}, \mu \rangle, R, Q \rangle$, where $\langle \mathtt{K_1}, \mathtt{K_2}, \mu \rangle$ is the closure currently being executed, R is the region of G currently owned by A, and Q is a list where $head(Q)$ is the current closure, and $tail(Q)$ is the list of closures that A will execute in sequence once the current closure is committed.

Two assembly states N_1 and N_2 are *disjoint* if the heap regions R_1 and R_2 referenced in N_1 and N_2 have no nodes in common.

Concurrent state A *concurrent state* of a Core Otello program is defined as a tree-shaped structure where a node is an ID for a specific finish-scope, and each node is associated with a collection of *assembly IDs*. There is a tree edge from an assembly ID A to a node F if the finish-scope represented by F is nested within the asynchronous task represented by A. At any point of an execution, each assembly ID is associated with a specific assembly state.

Formally, let us assume a universe of *assembly IDs* and a universe of *node IDs* (we assume these two sets to be disjoint). We define a *finish-tree* to be a pair $\mathcal{T} = (\mathcal{V}_\mathcal{T}, \lambda_\mathcal{T}, \dashrightarrow_\mathcal{T})$, where $\mathcal{V}_\mathcal{T}$ is a set of node IDs, $\lambda_\mathcal{T}$ is a map that associates each $F \in \mathcal{V}$ to a set of assembly IDs, and $\dashrightarrow_\mathcal{T}$ is a set of *edges*. An edge in a finish-tree has the form (A, F), where $F \in \mathcal{V}$ and $A \in F'$ for some $F' \in \mathcal{V}$. If $A \dashrightarrow_\mathcal{T} F$, then A is said to be the *parent* of F. We denote by $\dashrightarrow_\mathcal{T}^+$ the "transitive closure" in $\mathcal{T}$, defined as the least set of pairs (A, F) such that, for some F', A', we have (A, F'), $A' \in F'$, and $A' \dashrightarrow_\mathcal{T}^* F'$. If $A \dashrightarrow_\mathcal{T}^* F'$, then A is an *ancestor* of F.

It is required that:

- For any two distinct $F_1, F_2 \in \mathcal{V}$, $\lambda(F_1) \cap \lambda(F_2) = \emptyset$.
- The structure $\mathcal{T}$ is tree-shaped. This means that, first of all, there is no sequence $F_0, A_0, F_1, A_1 \ldots, F_k, A_k$ such that: (1) for all i, $F_i \in \mathcal{V}$, $A_i \in F_i$ and $A_i \dashrightarrow_\mathcal{T} F_{i+1}$, and (2) $A_k \dashrightarrow_\mathcal{T} F_1$. Second, there exists a root node F_0 such that for each node $F \neq F_0$ in $\mathcal{T}$, there exists an assembly ID A such that $A \in F_0$ and $A \dashrightarrow_\mathcal{T}^+ F$.

Abusing notation, we write $F \in \mathcal{T} = (\mathcal{V}_\mathcal{T}, \lambda_\mathcal{T}, \dashrightarrow_\mathcal{T})$ if $F \in \mathcal{V}_\mathcal{T}$, and $\mathcal{T}(F)$ to denote the set $\lambda_\mathcal{T}(F)$. Assemblies A_1 and A_2 are said to be *siblings* $A_1, A_2 \in \mathcal{T}(F)$ for some node F in $\mathcal{T}$. We say that an assembly A is a *leaf* in $\mathcal{T}$, and write $Leaf(A, \mathcal{T})$ if there is no node F such that $A \dashrightarrow_\mathcal{T} F$.

A *concurrent state* of a program P is now defined as a tuple $\langle G, \mathcal{T}, \Sigma, \Delta \rangle$, where G is a heap, $\mathcal{T}$ is a finish-tree, Σ is a function that assigns an assembly state $\Sigma(A)$ to each assembly ID A in $\mathcal{T}$, and Δ is a function that associates each node F in $\mathcal{T}$ with a list of closures. For any two distinct assembly IDs A_1 and A_2, we require $\Sigma(A_1)$ and $\Sigma(A_2)$ to be disjoint. In other words, an object in our model is owned by at most one assembly. The objects in G that do not belong to the regions referenced by $\Sigma(A)$, for assembly IDs A, are said to be *free*; we denote the set of all free objects by $Free(\Sigma)$.

Transitions Transition relation $\longrightarrow$ over states defines the operational semantics of Core Otello. The rules defining this relation are shown in Figures 11 and 12. Here, $\Sigma[A_1 \mapsto N_1; \ldots; A_k \mapsto N_k; A \mapsto \bot]$ denotes a map Σ' such that $\Sigma'(A_i) = N_i$, $\Sigma'(A)$ is undefined, and $\Sigma'(A') = \Sigma(A')$ for all other A' in the domain of Σ (a similar notation is defined for the map Δ). Also, $\mathcal{T}[F \mapsto \mathcal{A}]$ and $\mathcal{T}[A \dashrightarrow F]$ respectively denote a tree $\mathcal{T}'$ where $\mathcal{T}'(F) = \mathcal{A}$ and that is identical to $\mathcal{T}$ otherwise, and a tree $\mathcal{T}'$ that is obtained by adding a node F and an edge $A \dashrightarrow_\mathcal{T} F$ to $\mathcal{T}$.

The transitions in $\longrightarrow$ update the concurrent state of a program P. Most of these transitions select an assembly from the current finish-tree and execute the next statement in it. For an assembly to be scheduled for execution, it must be a leaf in the current finish-tree. The need for delegation arises when an assembly tries to access an object owned by a different assembly.

Specifically, the rule CONFLICT-SAME-LEVEL is used to define delegation among assemblies that are siblings of each other. In the premise of the rule, A_1 and A_2 are assemblies, respectively at states N_1 and N_2, and Q_1 is the list of closures that A_1 is obligated to execute. Further, the first element $head(Q_1)$ of this list is already under execution. The code for this closure is of the form $\mathtt{K_1'}; \mathtt{K_1}$. Of this code, $\mathtt{K_1'}$ has already been executed; the statement that is to be executed now is $head(\mathtt{K_1})$.

However, the statement $head(\mathtt{K_1})$ reads or writes an object u that is currently owned by the assembly A_2 (i.e., a *conflict* happens). Consequently, A_1 must now delegate its work and owned region to A_2. After the rule fires, the state of A_2 becomes N'. Note that the closure queue of A_2 is now $append(Q_2, Q_1)$.

Since A_1 may have modified certain objects in its region while it was executing $head(Q_1)$, the runtime rolls back

(CONFLICT)
$$\frac{N_1 = \langle\langle \mathtt{K}_1, \mathtt{K}_1', \mu_1\rangle, R_1, Q_1\rangle \quad head(\mathtt{K}_1) \in \{(\mathtt{x.f} := \mathtt{t}), (\mathtt{t} := \mathtt{x.f})\} \quad N_2 = \langle\langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_2, Q_2\rangle \quad \mu_1(\mathtt{x}) = u \quad u \in R_2}{conflict(N_1, N_2, u)}$$

(CONFLICT-SAME-LEVEL)
$$\frac{F \in \mathcal{T} \quad A_1, A_2 \in \mathcal{T}(F) \quad Leaf(A_1, \mathcal{T}) \quad \Sigma(A_1) = \langle\langle \mathtt{K}_1, \mathtt{K}_1', \mu_1\rangle, R_1, Q_1\rangle \quad \Sigma(A_2) = \langle\langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_2, Q_2\rangle \quad conflict(\Sigma(A_1), \Sigma(A_2), u) \quad N' = \langle\langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_1 \cup R_2, append(Q_2, Q_1)\rangle \quad \Sigma' = \Sigma[A_1 \mapsto \bot; A_2 \mapsto N']}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle rollback(G, \mathtt{K}_1'), \mathcal{V}, \mathcal{E}, \Sigma', \Delta\rangle}$$

(CONFLICT-BELOW)
$$\frac{F, F' \in \mathcal{T} \quad A_1, A_3 \in \mathcal{T}(F) \quad A_2 \in \mathcal{T}(F') \quad A_3 \dashrightarrow_{\mathcal{T}}^{+} F' \quad Leaf(A_1, \mathcal{T}) \quad \Sigma(A_1) = \langle A_1, \langle \mathtt{K}_1, \mathtt{K}_1', \mu_1\rangle, R_1, Q_1\rangle \quad \Sigma(A_2) = \langle\langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_2, Q_2\rangle \quad conflict(\Sigma(A_1), \Sigma(A_2), u) \quad N_3 = \langle\langle \mathtt{K}_3, \mathtt{K}_3', \mu_3\rangle, R_3, Q_3\rangle \quad N' = \langle\langle \mathtt{K}_3, \mathtt{K}_3', \mu_3\rangle, R_1 \cup R_3, append(Q_3, Q_1)\rangle \quad \Sigma' = \Sigma[A_1 \mapsto \bot; A_3 \mapsto N']}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle rollback(G, \mathtt{K}_1'), \mathcal{T}, \Sigma', \Delta\rangle}$$

(CONFLICT-ANCESTOR)
$$\frac{F \in \mathcal{T} \quad A_1 \in \mathcal{T}(F) \quad A_2 \dashrightarrow_{\mathcal{T}}^{+} F \quad Leaf(A_1, \mathcal{T}) \quad \Sigma(A_1) = \langle\langle \mathtt{K}_1, \mathtt{K}_1', \mu_1\rangle, R_1, Q_1\rangle \quad \Sigma(A_2) = \langle\langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_2, Q_2\rangle \quad conflict(\Sigma(A_1), \Sigma(A_2), u) \quad N_1' = \langle\langle \mathtt{K}_1, \mathtt{K}_1', \mu_1\rangle, R_1 \cup \{u\}, Q_1\rangle \quad N_2' = \langle\langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_1 \setminus \{u\}, Q_2\rangle \quad \Sigma' = \Sigma[A_1 \mapsto N_1'; A_2 \mapsto N_2']}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G, \mathcal{T}, \Sigma', \Delta\rangle}$$

(CONFLICT-UNRELATED)
$$\frac{F, F' \in \mathcal{T} \quad A_2 \in \mathcal{T}(F) \quad A_1 \in \mathcal{T}(F') \quad A_3 \dashrightarrow_{\mathcal{T}} F' \quad A_1 \not\dashrightarrow_{\mathcal{T}}^{+} F \quad A_2 \not\dashrightarrow_{\mathcal{T}}^{+} F' \quad Leaf(A_1, \mathcal{T}) \quad \Sigma(A_1) = \langle\langle \mathtt{K}_1, \mathtt{K}_1', \mu_1\rangle, R_1, Q_1\rangle \quad \Sigma(A_2) = \langle\langle \mathtt{K}_2, \mathtt{K}_2', \mu_2\rangle, R_2, Q_2\rangle \quad \Sigma(A_3) = \langle\langle \mathtt{K}_3, \mathtt{K}_3', \mu_3\rangle, R_3, Q_3\rangle \quad conflict(\Sigma(A_1), \Sigma(A_2), u) \quad N' = \langle\langle \mathtt{K}_3, \mathtt{K}_3', \mu_3\rangle, R_1 \cup R_3, Q_3\rangle \quad \Sigma' = \Sigma[A_1 \mapsto \bot; A_3 \mapsto N'] \quad \Delta' = \Delta[F' \mapsto append(\Delta(F), Q_1)]}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle rollback(G, \mathtt{K}_1'), \mathcal{T}, \Sigma, \Delta'\rangle}$$

(LOCAL-ACCESS-1)
$$\frac{Leaf(A, \mathcal{T}) \quad \Sigma(A) = \langle\langle \mathtt{K}_1, \mathtt{K}_2, \mu\rangle, R, Q\rangle \quad head(\mathtt{K}_1) \in \{(\mathtt{x.f} := \mathtt{t}), (\mathtt{t} := \mathtt{x.f})\} \quad \mu(\mathtt{x}) = u \quad u \in R \text{ or } u \in Free(\Sigma) \quad \mu \xrightarrow{head(\mathtt{K}_1)} \mu' \quad N' = \langle\langle tail(\mathtt{K}_1), \mathtt{K}_2; head(\mathtt{K}_1), \mu'\rangle, R \cup \{u\}, Q\rangle \quad G \xrightarrow{head(\mathtt{K}_1)} G' \quad \Sigma' = \Sigma[A \mapsto N']}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G', \mathcal{T}, \Sigma', \Delta\rangle}$$

(LOCAL-ACCESS-2)
$$\frac{Leaf(A, \mathcal{T}) \quad \Sigma(A) = \langle\langle \mathtt{K}_1, \mathtt{K}_2, \mu\rangle, R, Q\rangle \quad head(\mathtt{K}_1) \text{ has form } (\mathtt{x} := \mathtt{t}) \quad \mu \xrightarrow{head(\mathtt{K}_1)} \mu' \quad N' = \langle\langle tail(\mathtt{K}_1), (\mathtt{K}_2; head(\mathtt{K}_1)), \mu'\rangle, R, Q\rangle \quad \Sigma' = \Sigma[A \mapsto N']}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G', \mathcal{T}, \Sigma', \Delta\rangle}$$

(ASYNC)
$$\frac{F \in \mathcal{T} \quad A \in \mathcal{T}(F) \quad Leaf(A, \mathcal{T}) \quad \Sigma(A) = \langle\langle \mathtt{K}_1, \mathtt{K}_2, \mu\rangle, R, Q\rangle \quad head(\mathtt{K}_1) = \mathtt{async}\{\,\mathtt{K}_3\,\} \quad N' = \langle\langle tail(\mathtt{K}_1), \mathtt{K}_2; head(\mathtt{K}_1), \mu\rangle, R, Q\rangle \quad A'' \text{ is a fresh assembly ID} \quad \chi = \langle \mathtt{K}_3, \epsilon, \mu\rangle \quad N'' = \langle \chi, \emptyset, \{\chi\}\rangle \quad \Sigma' = \Sigma[A \mapsto N'; A'' \mapsto N''] \quad \mathcal{T}' = \mathcal{T}[F \mapsto \mathcal{T}(F) \cup \{A''\}]}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G, \mathcal{T}', \Sigma', \Delta\rangle}$$

Figure 11. Operational semantics of Core Otello.

$$\text{(NEXT-CLOSURE)}\ \frac{F \in \mathcal{T} \quad A \in \mathcal{T}(F) \quad \mathit{Leaf}(A,\mathcal{T}) \quad \Sigma(A) = \langle \epsilon, R, Q\rangle \quad \Sigma' = \Sigma[A \mapsto \langle \mathit{second}(Q), R, \mathit{tail}(Q)\rangle]}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G, \mathcal{T}, \Sigma', \Delta\rangle}$$

$$\text{(EMPTY-QUEUE)}\ \frac{\begin{array}{c} F \in \mathcal{T} \quad A \in \mathcal{T}(F) \quad \mathit{Leaf}(A,\mathcal{T}) \\ \Sigma(A) = \langle \epsilon, R, \emptyset\rangle \quad \mathcal{T}' = \mathcal{T}[F \mapsto \mathcal{T}(F) \setminus \{A\}] \quad A' \dashrightarrow_{\mathcal{T}} F \\ \Sigma(A') = \langle\langle \mathtt{K}_1', \mathtt{K}_2', \mu'\rangle, R', Q'\rangle \\ N' = \langle\langle \mathtt{K}_1', \mathtt{K}_2', \mu'\rangle, R \cup R', Q'\rangle \quad \Sigma' = \Sigma[A \mapsto \bot, A' \mapsto N'] \end{array}}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G, \mathcal{T}, \Sigma', \Delta\rangle}$$

$$\text{(FINISH)}\ \frac{\begin{array}{c} F \in \mathcal{T} \quad A \in \mathcal{T}(F) \quad \mathit{Leaf}(A,\mathcal{T}) \quad \Sigma(A) = \langle\langle \mathtt{K}_1, \mathtt{K}_2, \mu\rangle, R, Q\rangle \\ \mathit{head}(\mathtt{K}_1) = \mathtt{finish}\{\,\mathtt{K}_3\,\} \quad F' \text{ is a fresh node ID} \quad \mathcal{T}' = \mathcal{T}[F' \mapsto \emptyset; A \dashrightarrow F'] \\ \chi = \langle \mathtt{K}_3, \epsilon, \mu\rangle \quad \Delta' = \Delta[F' \mapsto \{\chi\}] \end{array}}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G, \mathcal{T}', \Sigma, \Delta'\rangle}$$

$$\text{(NEXT-CLOSURE-FINISH)}\ \frac{F \in \mathcal{T} \quad \mathcal{T}(F) = \emptyset \quad \Delta(F) \neq \emptyset \quad A \text{ is a fresh assembly ID} \quad N' = \langle \epsilon, \emptyset, \Delta(F)\rangle \quad \Sigma' = \Sigma[A \mapsto N']}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G, \mathcal{T}[F \mapsto \{A\}], \Sigma', \Delta[F \mapsto \emptyset]\rangle}$$

$$\text{(END-FINISH)}\ \frac{F \in \mathcal{T} \quad \mathcal{T}(F) = \emptyset \quad \Delta(F) = \emptyset}{\langle G, \mathcal{T}, \Sigma, \Delta\rangle \longrightarrow \langle G, \mathcal{T}[F \mapsto \bot], \Sigma, \Delta[F \mapsto \bot]\rangle}$$

Figure 12. Operational semantics of Core Otello (Continued).

the effect of the code $\mathtt{K}_1'$ before delegation. We denote by $\mathit{rollback}(G, \mathtt{K}_1')$ the result of atomically applying this rollback on G.

Note that because A_1 is required to be a leaf in the finish-tree $\mathcal{T}$ and also because of the syntactic restrictions in Core Otello, the code $\mathtt{K}_1'$ is solely a collection of heap updates—it does not include any `async` or `finish` statements. The implementation of this rollback operation is therefore straightforward.

The rule CONFLICT-BELOW describes delegation when A_1 conflicts with A_2, and there is an assembly A_3 that is a sibling of A_1 and an ancestor of the node containing A_2. In this case, A_1 delegates to A_3.

The rule CONFLICT-ANCESTOR handles the case when A_1 tries to access an object u owned by an ancestor A_2 of its node. Because A_2 is not a leaf node in the current finish-tree, it is currently "suspended"; therefore, A_1 can safely "steal" u from A_2.

The rule CONFLICT-UNRELATED handles the remaining conflict scenarios. Here, if $A_1 \in \mathcal{T}(F)$ conflicts with A_2, then A_1 transfers its owned region to the parent A_3 of the node F. The code executed by A_1 is now to be executed after all the assemblies in $\mathcal{T}(F)$ finish executing, and is put in the closure queue for F.

The rule LOCAL-ACCESS-1 formalizes read and write accesses by an assembly A to objects not owned by others. The local state of A changes to μ' from μ on executing $\mathit{head}(\mathtt{K}_1)$ (this is denoted by $\mu \xrightarrow{\mathit{head}(\mathtt{K}_1)} \mu'$); the heap changes from G to G' (this is denoted by $G \xrightarrow{\mathit{head}(\mathtt{K}_1)} G'$). If the object is not already in A's region, it is added. However, there is no delegation. The rule LOCAL-ACCESS-2 formalizes actions that update an assembly's variable state, but do not modify the heap.

The rule ASYNC defines the semantics of assembly creation. The rule creates a new assembly A' with an empty set of owned objects. The assembly A' belongs to the same tree node (finish-scope) as the assembly that created it.

NEXT-CLOSURE: when an assembly finishes executing a closure (we denote a closure that has executed all its code by ϵ)—i.e., the closure is *committed*, it selects for execution the second item in its closure queue (the first item is the closure that just finished executing); if Q has just one element, then $\mathit{second}(Q)$ is defined to be ϵ.

EMPTY-QUEUE: when the queue of closures for an assembly becomes empty, the assembly can be removed from the concurrent state of the current finish-tree node.

The rule FINISH defines the semantics of "`finish`"-statements executed by an assembly A. Here a new tree node F' is created; A is the parent of F', and $\mathcal{T}(F') = \emptyset$. The code inside the `finish`-block is transfered to the closure queue of F.

Executions and termination Consider a program $P = \mathtt{finish}\{\mathtt{K}\}$ in Core Otello. An *initial state* of P is a concurrent state $\pi_0 = \langle G, \mathcal{T}, \Sigma, \Delta\rangle$, where $\mathcal{T}$ consists of a single node F, $\mathcal{T}(F) = \emptyset$, Σ is the empty map, and $\Delta(F) = \chi$ for $\chi = \langle \mathtt{K}, \epsilon, \mu\rangle$ for some μ.

A *terminating state* of a Core Otello program is a concurrent state $\pi_n = \langle G, \mathcal{T}, \Sigma, \Delta\rangle$ where $\mathcal{T}$ is the empty tree. A Core Otello program is said to *terminate* if for all initial states π_0, there is a terminating state π_n such that $\pi_0 \longrightarrow^* \pi_n$.