The Tasks with Effects Model for Safe Concurrency

Stephen T. Heumann Vikram S. Adve Shengjie Wang
University of Illinois at Urbana-Champaign
{heumann1,vadve,wang260}@illinois.edu

Abstract

Today's widely-used concurrent programming models either provide weak safety guarantees, making it easy to write code with subtle errors, or are limited in the class of programs that they can express. We propose a new concurrent programming model based on *tasks with effects* that offers strong safety guarantees while still providing the flexibility needed to support the many ways that concurrency is used in complex applications. The core unit of work in our model is a dynamically-created task. The model's key feature is that each task has programmer-specified *effects*, and a run-time scheduler is used to ensure that two tasks are run concurrently only if they have non-interfering effects. Through the combination of statically verifying the declared effects of tasks and using an effect-aware run-time scheduler, our model is able to guarantee strong safety properties, including data race freedom and atomicity. It is also possible to use our model to write programs and computations that can be statically proven to behave deterministically. We describe the tasks with effects programming model and provide a formal dynamic semantics for it. We also describe our implementation of this model in an extended version of Java and evaluate its use in several programs exhibiting various patterns of concurrency.

Categories and Subject Descriptors D.3.2 [*Software*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.3 [*Software*]: Language Constructs and Features—Concurrent Programming Structures; D.1.3 [*Software*]: Concurrent Programming

General Terms Languages, Verification, Design, Performance

Keywords Tasks, effects, task scheduling, concurrent and parallel programming, task isolation, data race freedom, atomicity, determinism

1. Introduction

Concurrency is used for many purposes in modern programs. To exploit the full capabilities of today's multicore processors, parallel algorithms must be used. But concurrency is also used for other purposes. This is perhaps particularly true of interactive programs, both on end-user devices and servers, where the behavior of the user or client is inherently concurrent with the program. In GUI programs, long-running operations should be run concurrently with user interface event processing in order to preserve responsiveness. It can also be convenient to express a full program as a set of of modules or actors [3] that can operate concurrently and communicate with each other. This can be a natural fit, for example, to the model-view-controller design of interactive programs.

Large programs often combine multiple types of concurrency. For example, an interactive application may separate long computations from the UI thread or use multiple concurrent modules, but also sometimes perform data-parallel computations. We believe a widely-applicable concurrent programming model should seek to support all of these forms of concurrency, since they are all widely used and are often combined within a single application.

Today, parallel and concurrent programs are commonly written using threads, with low-level mechanisms such as locks used for synchronization (or with carefully designed lock-free data structures). Such a programming model is flexible enough to express many forms of concurrency, but it does not guarantee any safety properties such as data race freedom, atomicity, deadlock-freedom, or determinism. It also provides little well-defined structure for the concurrent control flow and synchronization in programs, making it difficult to reason about them manually or automatically. In addition, complicated low-level details such as processor memory models [2] can affect the semantics of programs written in this style, further complicating the task of reasoning about them.

Many previous systems have attempted to address aspects of these problems. Some offer more structured parallel control and synchronization constructs, but sometimes with limitations that prevent them from expressing general, event-driven concurrency, and often without strong safety guarantees. Cilk [11] and Thread Building Blocks (TBB) [22], for example, provide structured parallelism constructs, but they do not offer checked guarantees of strong safety properties such as data race freedom.

Some other systems do seek to offer stronger guarantees. The Deterministic Parallel Java (DPJ) language [13, 14] offers a strong set of guarantees for programs that can be expressed in it. These include data race freedom, strong atomicity [1], deadlock freedom, and deterministic semantics with full sequential equivalence for parallel computations that do not explicitly use nondeterministic parallel constructs. These guarantees are very strong, but DPJ's parallelism model does not provide the flexibility that we seek. Most critically, DPJ is restricted to fork-join parallelism structures, which are not suitable for many concurrent programs.

In this paper, we propose a new model for concurrent programming, which gives strong safety guarantees while providing the flexibility needed to express a wide range of concurrent programs in it. We call our model *tasks with effects*.[1] It uses tasks that can execute concurrently as the fundamental units of work. Tasks are lighter-weight constructs than threads and support only limited operations for inter-task communication and synchronization. Concurrent work is launched by creating a new task, and it is possi-

PPoPP'13, February 23–27, 2013, Shenzhen, China.

[1] Two workshop papers gave preliminary descriptions of the tasks with effects model [20, 21].

ble for one task to await the completion of another. A scheduler is responsible for executing tasks in an efficient manner. Tasks provide a structured mechanism for concurrent control flow, while still preserving the flexibility to express a wide variety of concurrency patterns and parallel algorithms.

Several existing systems support task-based programming models, including Intel's TBB, Apple's Grand Central Dispatch and operation queues [6], Microsoft's Task Parallel Library in .NET [30], the `ForkJoinTask` framework in Java 7 [33], and the tasking operations in OpenMP 3.x [32]. However, they do not offer strong safety guarantees. It is possible for two concurrent tasks to perform conflicting accesses that give rise to data races or violations of intended atomicity properties, and it is generally the programmer's responsibility to manually reason that such accesses do not occur or are benign, or else to protect them using low-level synchronization mechanisms such as locks.

We propose instead to associate a checked effect specification with each task. The run-time system then schedules tasks so as to ensure that only tasks with non-interfering effects can run concurrently. Effect specifications can take many forms, but in this work we adopt the statically-checked effect system developed for Deterministic Parallel Java [13]. In this system, the compiler statically verifies that the memory accesses in each task or method are covered by its programmer-specified effects. By combining these static checks with our dynamic effect-based task scheduling system, we are able to guarantee the basic *task isolation* property that no two tasks with interfering effects may run concurrently with each other. This guarantee leads to a guarantee of data race freedom, and to a guarantee of atomicity for tasks or portions of tasks that do not create or wait for any other tasks.

We also define mechanisms based on *effect transfer* between tasks to further enhance the utility of our model. One mechanism is used to avoid a class of deadlocks, and also enables certain useful programming paradigms. Another form of effect transfer is used for nested parallel computations. It enables us to provide a compile-time guarantee of determinism for a class of deterministic programs and algorithms similar to those supported by DPJ. We are aware of no other programming model which provides equally strong safety guarantees while supporting the flexible control flow needed for general concurrent programs such as interactive applications and actor-like programs.

This paper makes the following contributions:

- We define the tasks with effects programming model, which supports flexible task-based concurrency while providing a strong set of safety guarantees.
- We describe the TWEJava language which implements this model, and describe our compiler and runtime system for it.
- We provide a formal dynamic semantics of tasks with effects, and describe how it guarantees task isolation, data race freedom, atomicity, and (for certain computations) determinism.
- We evaluate the expressiveness and performance of our language and implementation. We show that TWEJava can be used to write a variety of concurrent and parallel programs, including two interactive applications, and that we can achieve substantial parallel speedups.

The rest of this paper proceeds as follows. Section 2 presents the TWEJava language and describes the task-related operations used in it. Section 3 gives a dynamic semantics of tasks with effects. Section 4 describes the safety properties of our model. Section 5 discusses our implementation of TWEJava in a compiler and runtime system, and section 6 evaluates it on several benchmark programs. Finally, section 7 discusses related work and section 8 concludes.

```
1  public abstract class Task<type TRet, TArg, effect E> {
2    // Code to be run when task is executed.
3    public abstract TRet run(TArg arg) effect E;
4
5    // Execute a task at some point in the future
6    public final TaskFuture<TRet> executeLater(TArg arg);
7    // Spawn a subtask of the current task, with effect transfer
8    public final SpawnedTaskFuture<TRet, effect E> spawn(TArg arg);
9  }
10
11 public class TaskFuture<type TReturn> {
12   // Await completion and get return value (no effect transfer)
13   public TReturn getValue();
14   // Check if task is done
15   public boolean isDone();
16 }
17
18 public class SpawnedTaskFuture<type TReturn, effect E>
19       extends TaskFuture<TReturn> {
20   // Await completion and get return value, with effect transfer
21   public TReturn join();
22 }
```

Figure 1. Operations supported by TWEJava. The abstract method `run` must be implemented in concrete subclasses of `Task`, giving the code to be run as a task. The other operations, although using the syntax of Java methods, are in fact new task-related language operations supported by our compiler and runtime system.

2. The TWEJava language

We implement the tasks with effects model for safe, flexible concurrency in an extended version of Java, which we call TWEJava. (TWEJava programs can use almost all Java language features, but they should not use Java's thread-based concurrency mechanisms or lock-based synchronization, which TWEJava is designed to replace.) Figure 1 shows the new operations supported by TWEJava.

Figure 2 shows how our task system can be used in an image editing program, which we will use as as running example. It illustrates a simplified version of a programming pattern used in the ImageEdit program that we have implemented in TWEJava (see section 6). The example code shows a class `Image` representing an image, with the pixel values held in two arrays, `topHalf` and `bottomHalf`. We would like to support operations in parallel on these two halves of the image. (We adopt this arrangement for simplicity. In the actual ImageEdit application, it is possible to use finer-grained parallelism.) We also want to support a variety of operations to read and manipulate the image, which may be invoked as asynchronous tasks. This is useful, for example, when the user directs the program to perform a lengthy operation that should not block the user interface while it runs.

We show the task `increaseContrast` (lines 6–16), which can be executed to increase the contrast of the image. It relies on the separate method `increasePixelContrast` (lines 18–26) to actually update the pixel values in each array. This enables the `increaseContrast` operation to work on the top and bottom halves of the image in parallel, by spawning a child task to work on the top half while the parent task works on the bottom half.

Figure 3 shows the tasks created in this computation. The GUI system executes the `increaseContrast` task in response to user input. That task in turn spawns a child task so that the two halves of the image can be processed in parallel, and then joins that child task after it completes. Meanwhile, the GUI system might execute additional tasks in response to further user input. (In this example, we show the GUI system as a task, responsible for processing low-level input data and launching tasks in response to UI events. This architecture would be possible, but for ease of implementation we have so far used Java's Swing GUI framework, with wrappers to launch tasks in response to Swing events.)

```
1  class Image {
2    region Top, Bottom;
3    final int[]<Top> topHalf;        // pixel values
4    final int[]<Bottom> bottomHalf;
5    ...
6    public final Task<Void, Void, writes Top,Bottom>
7    increaseContrast =
8        new Task<>() {
9          public Void run(Void _) {
10           SpawnedTaskFuture<Void, writes Top> f =
11               increasePixelContrast(topHalf).spawn(null);
12           increasePixelContrast(bottomHalf).run(null);
13           f.join();
14           return null;
15         }
16       };
17
18   private static <region runtime R> Task<Void, Void, writes R>
19   increasePixelContrast(final int[]<R> pixels) pure {
20     return new Task<>() {
21       public Void run(Void _) {
22         modify values in pixels array
23         return null;
24       }
25     };
26   }
27 }
```

Figure 2. Example computation.

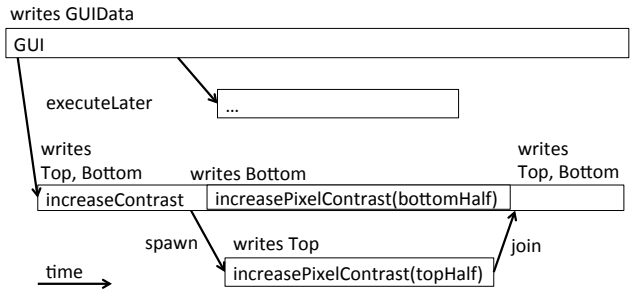

Figure 3. Tasks in example computation.

2.1 Tasks

In TWEJava, potentially concurrent work is made by creating a *task*, which will then be executed at some point when execution resources are available. It is possible to check if a task is completed or block awaiting its completion. A program written in TWEJava is started by invoking an initial task, and creating new tasks is the sole means of performing concurrent work. Tasks can also take parameters as input and return a result. (Wrapper classes support tasks with multiple parameters.) Three fundamental operations implement this basic tasking model: `executeLater` adds a task to the queue of tasks to be executed, `getValue` waits until a task is done and gives its return value, and `isDone` checks whether a task is done, without blocking.

Each type of task is specified by a subclass of the `Task` class, which takes type parameters giving its input and output types, and an effect parameter giving its effect (described below). An `executeLater` operation performed on a `Task` instance returns a `TaskFuture` object, which represents an actual execution of the task; `getValue` and `isDone` operations can be performed on this task future. In our example, the `increaseContrast` task is created by an `executeLater` operation in the GUI. (The `spawn` and `join` operations used within it will be described in section 2.5.)

The structure described above is similar to other existing task systems, but TWEJava has a key difference. In other systems, a task can generally be run at any time after it is queued for execution, without regard to what other tasks are running concurrently. Because of this, the programmer must take care to ensure that there are no data races between potentially concurrent tasks. This can be done by using synchronization mechanisms such as locks within tasks to guard access to shared data, or by carefully designing the pattern in which tasks are executed and joined in such a way that no two tasks accessing the same data might be executed concurrently. Using these mechanisms to guard against data races is often complex and error-prone, and traditional thread-based systems for concurrent programming generally do not provide a mechanism to automatically check that they have been used correctly.

Our system solves this problem by using *effects* to control the scheduling of tasks. Each task has an effect specification, which is checked at compile time to ensure that it accurately (conservatively) reflects the task's memory accesses. These effect specifications of tasks are in turn used at run time by the task scheduler, which will ensure task isolation—that is, that no two tasks with interfering effects can be running concurrently.

2.2 Effects and Regions

In order to perform effect-based scheduling of tasks, our system must know the effects of each task, and be able to check whether the effects of two different tasks interfere with each other. Intuitively, two tasks interfere if they could both access the same memory location and at least one of those accesses could be a write. Two tasks can only be run concurrently if their effects do not interfere, which is the core property enforced by the scheduler in our system.

In TWEJava, we use the effect system originally developed for the Deterministic Parallel Java (DPJ) language [13]. DPJ is an extended version of Java that uses type and effect annotations to enable the compiler to statically prove strong safety properties for programs written using its fork-join parallel constructs. In this work, however, we adopt its type and effect system for use in combination with our effect-based task scheduling system.

The DPJ type and effect system is based on a partitioning of memory into *regions*. The programmer can declare each object field and array cell to be in a specified region. Region-parameterized types and methods are also supported. This permits different instances of a class to have their fields in different regions by giving different region arguments when instantiating the class. In addition, nested hierarchies of regions are supported by using *region path lists* (*RPLs*), and *index-parameterized arrays* allow each element of an array to be placed in a distinct region. A wildcard $*$ can be used in RPLs to specify effects covering a set of regions.

Using this partitioning of memory into regions, the effects of any operation in the program can be specified in terms of read and write effects on memory regions. The programmer declares the effects of each method as part of its method signature. The compiler can then statically verify that the declared effects of each method actually cover the effects of every operation in it. The DPJ type and effect system also defines formally under what circumstances two effects can be proven to be non-interfering. In DPJ, this information is used purely statically to verify that programs using simple fork-join parallelism constructs have no interference of effect between portions of code that can run concurrently.

TWEJava adopts DPJ's region-based type and effect system, but couples it with a runtime representation of effects that is used by a run-time scheduler to guarantee noninterference of effect between concurrent tasks. This allows it to support a much wider range of programs than DPJ can handle, including those that are inherently nondeterministic and do not use a fork-join style of concurrency.

We also use an extension of the basic DPJ type system to support effect parameters to types (in addition to region parameters), which was introduced in [12]. This allows us to use an effect pa-

rameter E in our definition of the abstract class Task, which will be extended by each actual task defined in the user's code. The definition of each actual type of task will instantiate this parameter with that task's effects, and the compiler will then be able to ensure statically that the effects of the supplied run method for that task are actually covered by the effect parameter E. Thus, our runtime system can safely use that effect parameter as a (possibly conservative) summary of the actual effects of the task.

In our example code, we declare two region names, Top and Bottom (line 2). We then declare the cells of the topHalf and bottomHalf arrays to be in those two regions, respectively. The increaseContrast task is declared with the effects writes Top, Bottom, meaning it can read and write the pixel values in both halves of the image. The increasePixelContrast method has a region parameter R corresponding to the region containing the cells of the array passed to it. Since the declared effect of the task it returns is writes R, increasePixelContrast(topHalf) will produce a task with the effect writes Top.

Like DPJ, we use purely static checks to ensure that each method and task complies with its effect declaration and that region- and effect-parameterized types are used soundly. TWEJava never requires runtime checks associated with individual memory accesses, which avoids a major source of overhead in some other systems such as STMs. However, our system does need to have information on the effects of tasks available at run time so that it can be used by the scheduler. We make this information available by introducing a set of internal runtime classes that represent dynamic regions and effects, and internally adding extra fields to classes which hold the runtime instantiation of their region and effect parameters, as well as extra arguments to constructors and methods corresponding to the region and effect parameters passed to them.

The scheduler only directly needs information about the effect parameters of task objects, but these may depend on other region and effect parameters in scope at the places where task classes are declared and instantiated, making it necessary to also track those additional parameters at run time. To minimize the overhead of this run-time tracking, we require programmers to annotate region parameters that need to be tracked at run time. This allows us to avoid generating run-time tracking code for the many region parameters that are used only in the compiler's static analysis. (Failing to provide such an annotation where needed will cause a compile-time error.)

2.3 Effect-Based Task Scheduling

The key property that our run-time task scheduler must enforce is that two tasks with interfering effects will not be run concurrently. To do this, the scheduler will have to delay the execution of tasks that are created while another task with interfering effects is already executing. It may also delay tasks for other reasons, e.g. waiting until execution resources are available.

In Figure 3, the increaseContrast task with effects writes Top, Bottom is run while the GUI task with effect writes GUIData continues to execute. To determine if the new task may be run concurrently with the already-executing task, the scheduler will check if these two sets of effects interfere with each other. In this case, the region GUIData is disjoint from Top and Bottom, so the two tasks have non-interfering effects and may be run concurrently.

If a third task is run with executeLater while these two tasks are executing, its effects will be checked against those of both existing tasks. Thus, another task trying to access the image data in the regions Top and Bottom would have to wait until the increaseContrast task is done, but a task accessing different regions might be able to run concurrently. (The increasePixelContrast(topHalf) task is run with the spawn operation, which uses effect transfer to avoid the need for these run-time checks; see section 2.5.)

Considerable variation is possible in the design of an effect-aware task scheduler. Our initial prototype implementation uses an approach based on a linear queue of tasks, which is described in section 5.2. For greater performance and scalability, the effect checking could be structured around regions, so that tasks accessing unrelated regions do not need to be explicitly checked against each other. A scheduler may also provide additional properties related to fairness or task ordering, in addition to the basic property of noninterference. For interactive programs, it is valuable to preserve responsiveness through fairness properties that avoid delaying the execution of one task excessively while other tasks execute ahead of it. But for efficiency in many parallel codes, it would be desirable to use algorithms similar to Cilk's work-stealing scheduler [11], which preferentially execute recently-created tasks on each processor. We believe that the design of high-performance effect-based task schedulers is a valuable area for future work.

2.4 Effect Transfer When Blocked

The model we have described so far envisions the effects of each task remaining unchanged while it runs, and says that two tasks with interfering effects may not run concurrently. This will lead to deadlock if one task blocks waiting for another task that has yet to run and which has effects that interfere with those of the first task. For example, if task A creates task B using executeLater, then blocks on B using getValue, and the effects of tasks A and B interfere, deadlock results because B cannot begin execution until A is complete.

We wish to prevent this form of deadlock and enable certain useful programming patterns involving this sort of blocking, so we introduce a mechanism for *effect transfer* from a blocked task to the task it is blocked on. The key idea is that a getValue or a join operation (described later) "transfers" enough effects from the blocking task to the target task to allow the target task to begin execution. For example, if a task A is blocked on another task B using getValue, we record this fact and ignore any effect conflict between A and B in deciding whether B can be executed. We also extend this to indirect blocking through chains of blocking operations. Note that a task that blocks will always remain blocked until all the tasks it directly or indirectly blocks on are done. Therefore, this mechanism does not enable two tasks with conflicting effects to be actively running at the same time.

This form of effect transfer prevents the type of deadlock described above, and it also allows some useful programming patterns. One of these is for one module of the program with effects on a certain region to launch and block on a task in another module, which may "call back" to the first module by launching and blocking on another task whose effects interfere with those of the first task. Another useful programming pattern enabled by this mechanism is similar to a locked or atomic block in other programming models. One task can launch a second task with a superset of its effects, and then use a getValue operation to wait for the second task. This transfers the first task's effects to the second task (allowing it to access the same regions as the first task), and leaves the second task to wait until it can acquire access to the regions covered by its other effects, which may correspond to a shared resource.

2.5 Effect Transfer for Nested Parallelism

Our system supports an additional form of effect transfer which is particularly suitable for nested parallelism, as used in fork-join style computations. It is a mechanism to transfer some of the effects of a parent task to a newly-created child task, and later transfer those effects back to the parent task when the child task completes. We call these operations spawn and join, respectively. A child

task created with `spawn` may run immediately, since "ownership" of its effects is transferred directly from the parent to the child task, and thus no other tasks with conflicting effects may be running concurrently.

In Figure 2, these mechanisms are used to operate in parallel on the two halves of the image. We use the `spawn` operation to run the `increasePixelContrast(topHalf)` task (line 11). This transfers the effect `writes Top` directly from the parent `increaseContrast` task to the new child task, which means the new task can be enabled for execution immediately. The parent task also continues executing concurrently, with its remaining effect `writes Bottom`. The `increasePixelContrast(bottomHalf)` operation is run as a method within the parent task, which is possible since its remaining effect `writes Bottom` covers the effect of the method call. After that computation finishes, the parent task joins the spawned child task. This `join` operation also transfers the child task's effect `writes Top` back to the parent task. After this, both halves of the image will have been updated, so any other task that waits for the `increaseContrast` task to finish will know that the full operation is complete.

2.5.1 Spawning and joining child tasks

The `spawn` operation executes a new task, whose effects must be entirely covered by the effects of the parent task calling `spawn`. It immediately transfers those effects to the spawned task, which allows that task to be enabled for execution immediately, without going through the normal effect-based scheduling process required when using `executeLater`. Since the effects are transferred directly from the parent task to the child task, data in regions covered by those effects cannot be modified by any other task in the interim, so the child task reading that data is guaranteed to see the values last seen or written by the parent task.

The `join` operation permits effect transfer back to the parent task at the end of a child task. Apart from effect transfer, `join` behaves like `getValue`: it will await the completion of the joined task, and return the result value produced by it, if any. The difference is that `join` will transfer effects directly from a completed task to the task that joins it. This permits the task that called `join` to perform subsequent operations covered by the effects of the joined task. One application of this is that data written by the completed child task can be read by its parent task after the child task is done.

Only tasks executed with `spawn` are joinable, and this is reflected by the fact that `spawn` returns a `SpawnedTaskFuture`, which supports the `join` operation. Furthermore, only the parent task that spawns a task may join it, and a task may only be joined once (violating these rules causes an exception to be thrown). Also, the system implements an implicit join operation prior to returning from each method for all the tasks spawned by that method that have not already been explicitly joined. These measures ensure that all `spawn`ed tasks get joined, and that all the effects transferred from a method with `spawn` are returned to it through `join` operations before the method returns. This simplifies our static effect analysis, since a method never "gives up" effects from the perspective of its callers.

2.5.2 Covering Effect Analysis for Effect Transfer

Implementing effect transfer makes the static analysis of covering effects more complex, since a `spawn` or `join` operation subtracts or adds effects to the task in which it is executed, thereby changing the covering effects applicable to subsequent code in that task. This would be easy to address if we used dynamic checks to determine whether the effect of each memory operation is covered by the current effects of the task in which it appears, but we want to use a static analysis to determine this in order to minimize runtime overheads and detect as many errors as possible at compile time.

To do so, we added a dataflow analysis algorithm in the compiler to conservatively compute the *current covering effect* applicable to each expression in the program. The current covering effect at the beginning of a method is given by its declared method effect summary. When `spawn` operations are encountered, the statically declared effects of the spawned task are subtracted from the current covering effect, and a `SpawnedTaskFuture` parameterized by the effects of the spawned child task is returned. At `join` operations, the effects given by the static type of the joined `SpawnedTaskFuture` are added to the current covering effect. At control flow join points, a minimum of the current covering effects from the different control flow paths is used. Using an iterative dataflow analysis, we can thus conservatively compute the current covering effect applicable to each expression in the program. The effects of each expression can then be compared against its current covering effect to ensure the expression's effects will be covered.

In our example the covering effect of the `increaseContrast` task is initially `writes Top, Bottom`. When it spawns a child task (line 11), its covering effect then becomes `writes Bottom`, since the `writes Top` effect has been transferred to the spawned child task. When that task is joined (line 13), the covering effect of the parent task once again becomes `writes Top, Bottom`.

One detail that must be accounted for in this analysis is that effect-parameterized types in the static program code are in general only a conservative summary of the actual effects of tasks at run time, and they may contain wildcard elements in their region specifiers. The actual effects of the `Task` object used at run time may be smaller than the effects given in the static type, e.g. by omitting some of the effects that are included in the static type or replacing effects on RPLs containing wildcards (which can cover a set of regions) with effects on a fully-specified RPL designating a single region in that set. We generally use a conservative static analysis: `spawn`s are treated as transferring away all the effects in the static type of the spawned task, including ones with wildcards. Subsequent operations in the parent task may not interfere with those transferred-away effects, which conservatively ensures that they cannot interfere with any of the actual effects of the child task at run time.

As an exception in this conservative analysis, however, we allow `spawn` operations even if we cannot be certain at compile time whether or not the effects of the spawned task will actually be covered at run time. In this case, we generate code to keep track of the run-time covering effects in the method containing the `spawn` operation (updated only when a `spawn` or `join` operation is performed). An exception will be thrown if the effects of the spawned task are not actually covered at run time. This limited dynamic checking is useful for cases where we do not have full information on the effects of spawned tasks at compile time. For example, a loop may spawn tasks to operate on different elements of an index-parameterized array, but our compiler cannot determine statically whether each of the elements is distinct, so this mechanism effectively enables the check to be performed dynamically instead.

For `join`s, we need to be sure that the actual run-time effects of the task being joined are not less than those specified in the static type. To do this, we statically treat `join`s as performing effect transfer only if the effect parameter of the joined task's static type is fully-specified (i.e. contains no wildcards). We also adopt the typing rule that an effect-parameterized type `A` is only treated as a subtype of another effect-parameterized type `B` if either the corresponding effect parameters are exactly equivalent, or the effect parameters in `B` are *not* fully-specified. This ensures that fully-specified effect parameters in the static types of `SpawnedTaskFuture`s exactly match the actual parameters used when instantiating the task object at run time, so we may safely use those parameters in the static analysis of `join` operations.

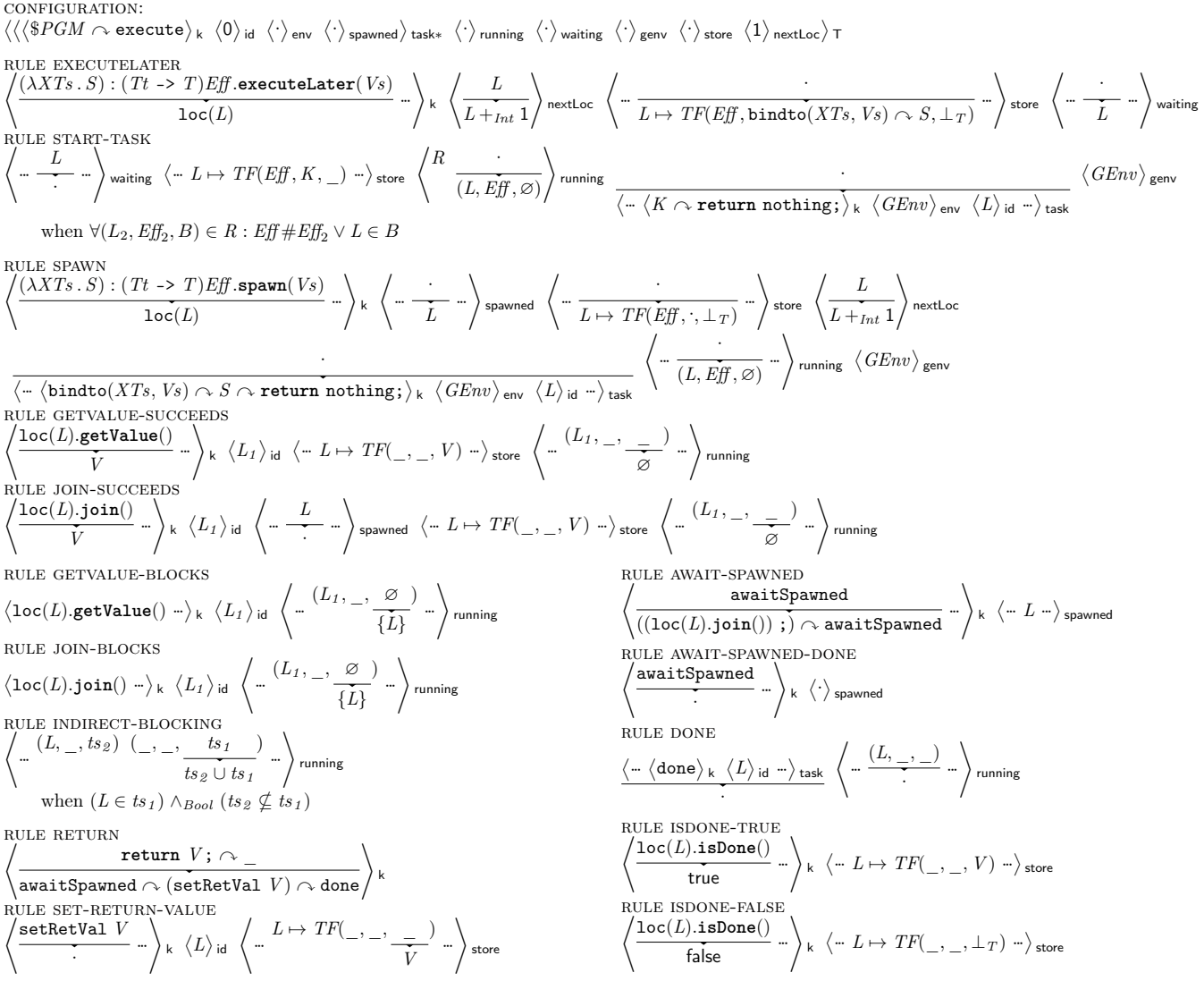

Figure 4. Dynamic semantics of tasks with effects.

3. Dynamic Semantics of Tasks with Effects

We have formalized the dynamic semantics for the core operations of the tasks with effects model in the context of a basic imperative language. A program in this language consists of a set of global variable declarations and task declarations (which are similar to function declarations in a traditional language, but include an effect specification for each task). Here we present and describe only those semantic rules related to tasks, which are shown in Figure 4.

These rules are written using the K semantic framework [36], which is based on rewriting logic and operates on a configuration of nested cells which corresponds at any point to the current state of the execution. (Although the K framework is less common than the standard approach for operational semantics, it has significant advantages, especially in that it is more modular and flexible.) Each rule may apply when it can match the configuration elements on the top of it, and when it applies any elements with a horizontal line under them are replaced by what is below the line. K supports lists, sets, and maps, and a rule may match a single element from these structures, either anywhere in them or at the front of a list; in these cases, the remainder of the structure is denoted by ellipses. A dot represents the identity element of these structures, and an underline is a 'don't-care' element that can match anything. K rules also obey a locality principle, saying that a rule matching two subcells that appear within the same outer cell must match only two subcells within the same *instance* of that outer cell.

At the top of Figure 4, we show the initial configuration of the program. It consists of a *task* cell (of which there may later be more than one, indicated by the *); a *running* cell which will hold a set containing information on running tasks; a *waiting* cell which will contain a set of IDs of tasks waiting to execute; a *genv* cell holding the global environment (mapping identifiers to locations in the store); a *store* cell which will map locations (integers) to various objects; and a *nextLoc* cell giving the next available location in the store. Each *task* cell contains code to be executed in its *k* subcell; an ID in its *id* subcell (corresponding to a location in the store); the current environment in its *env* subcell; and a set of IDs of spawned child tasks in its *spawned* subcell. The initial configuration will pass the program code to a special operation `execute` (not shown) which initializes the store and global environment based on the declarations in the program and then runs the task named `main`.

Note that we present here only a dynamic semantics, which presupposes that the program has passed all static checks, including type checking and checking that the current covering effects at each point in each task correctly cover all the memory accesses it may perform. (Dynamic computations of current covering effects are not needed in this formalism, because the effects of each task are fully defined statically and there is no provision for dynamic instantiation of region or effect parameters.) These semantics are agnostic to the specific effect system used, but a formalism of the DPJ type and effect system used in TWEJava is presented in [13].

3.1 Starting Tasks

The first major class of rules in our semantics relates to starting tasks. The EXECUTELATER rule implements the `executeLater` operation. It will apply once the `executeLater` operation is the next piece of code to execute, after a task name in the code has been evaluated to a lambda expression (comparable to a `Task` object in TWEJava) and its arguments have been evaluated to values (simple rules not shown). The EXECUTELATER rule will allocate a new location L in the store, and store a *TF* tuple (corresponding to a `TaskFuture` in TWEJava). This tuple contains the effect of the task, the code to be executed when it is run, and the task's return value (initially $\perp_T$, indicating it has not yet been set). The rule adds the ID (location) of this task to the set of tasks waiting to run, and the result of the operation is a reference to that location.

The START-TASK rule is then responsible for actually starting one of the tasks in the *waiting* set. When it applies, it will create a new *task* cell in the configuration, containing the code of the new task to be run. (This cell may exist side-by-side with other *task* cells.) The rule also adds a tuple (L, *Eff*, $\varnothing$) to the *running* cell. This indicates that the task L is now running, and holds its effects and an initially-empty set of tasks that it is blocked on. Finally, the key element of this rule is the condition relating to the existing contents R of the *running* cell. This will contain information about all the other currently-running tasks, and we use it to ensure our model's basic property of task isolation. Specifically, the new task L cannot be started unless for every already-running task L_2, either the effects of L are non-interfering with those of L_2 (denoted by $\#$) or L is in the set of tasks on which L_2 is blocked. This latter case implements our mechanism for effect transfer when blocked.

The SPAWN rule is similar to a combination of the EXECUTELATER and START-TASK rules, since it allows a task to start immediately without the need for the effect checking in the START-TASK rule. One addition, however, is that the ID of the spawned task is added to the *spawned* set of its parent task, which keeps track of child tasks that have been spawned and not yet joined.

3.2 Awaiting Completed Tasks and Blocking

The next group of rules relates to the potentially blocking operations `getValue` and `join`. They both can be applied to a reference to a location containing a *TF* tuple. The GETVALUE-SUCCEEDS rule addresses the case where the task in question is complete, and as such has a return value V stored in its *TF* tuple. In this case, the result of the operation is that value. Since the task that executed the `getValue` operation (L_1) will no longer be blocked, we empty the blocked-on set in its *running* tuple. The JOIN-SUCCEEDS rule is similar, but also requires that the task being joined was in the current task's *spawned* set, and removes it. This reflects the fact that a task can only be joined once, and only by the task that spawned it. (If a `join` operation violates these rules, the task that executes it will hang in our formalism. In TWEJava, an exception is thrown.)

The next two rules, GETVALUE-BLOCKS and JOIN-BLOCKS, handle the case where the task L may not yet be done. These rules put L in the blocked-on set for the task L_1 that does a `getValue` or `join` operation on L. This potentially allows L to be started based on effect transfer, using the START-TASK rule. The INDIRECT-BLOCKING rule propagates entries in the blocked-on sets when there is a chain of blocked tasks, allowing effect transfer to be applied in the case of indirect blocking. (In the TWEJava implementation, this propagation is fully performed at the time a `getValue` or `join` operation is evaluated.)

3.3 Finishing Tasks and Checking If Tasks are Done

The next group of rules relates to finishing a task. The RETURN rule handles a `return` statement (which may be in the program's code, or the `return nothing;` that we insert at the end of each task when starting it, in case it does not explicitly return a value). The rule says to first await any spawned children of the current task that have not yet been joined, then set the task's return value in its *TF* tuple (which will signal that the task may be considered done), and finally erase its *task* cell and its entry in the *running* set. The next several rules implement these operations.

Finally, the last two rules implement the `isDone` operation. A task is considered done once its return value has been set to a value. If it is still undefined (indicated by $\perp_T$), then the task is not done.

4. Safety Properties

Our model guarantees strong safety properties, including our basic task isolation property, plus data race freedom and atomicity properties stemming from it. We also avoid a significant class of deadlocks and can prove that many computations are deterministic.

4.1 Task isolation

The task isolation property of our system is that no two tasks may be actively running concurrently with interfering covering effects. The basic check used to guarantee this is to record the effects of each running task in the *running* set, and compare the effects of new tasks against the effects of all existing tasks before allowing them to start in the START-TASK rule.

There are two cases where we can start tasks even though they might appear to have effects interfering with those of another running task. One is that a task A may be allowed to start while a task B with conflicting effects is in the *running* set if A is in the blocked-on set for B. In this case, our rules guarantee that B cannot resume execution until A has completed, so we allow A to run based on our first effect transfer mechanism.

The other case is the `spawn` operation. In this case, our covering effects analysis ensures that the spawned task's effects are subeffects of its parent task's effects (so they may not conflict with anything that the parent's effects do not) and that the parent task will not execute any operations that conflict with the effects of the spawned task between where it is spawned and where it is joined.

4.2 Data race freedom

Data race freedom follows from the combination of the task isolation property and the guarantee provided by our static checks that the specified effects of each task cover all its memory accesses.

The formalism in section 3 implicitly uses a sequentially-consistent memory model, but in fact the tasks with effects model requires memory updates to be visible only between operations ordered by a limited set of happens-before edges. Our model imposes some order on any two tasks with interfering effects. This gives rise to happens-before edges between the end of one task and the start of any subsequent task with interfering effects, analogous to those between a lock release and subsequent acquisition in other systems. A full happens-before relation for our model is given by the transitive closure over these edges as well as edges for task creation, waiting or checking for task completion, and the sequential program order within each task. Any two accesses to a

memory location where at least one is a write will be ordered by this happens-before relation.

4.3 Atomicity

A task or portion of a task that does not create or wait for any other tasks behaves atomically. It has fixed effects that cover all the memory locations it can access, and the scheduler will ensure that no other tasks performing conflicting accesses run concurrently with it, which ensures it is atomic. This atomicity property also extends to portions of tasks that contain task creation operations, in the sense that the semantics are equivalent to those given by creating the new tasks only at the end of the parent task or just before the next `getValue` or `join` operation in it.

Atomicity does not always extend across `getValue` or `join` operations, as our mechanism for effect transfer when blocked may allow other tasks with conflicting effects to run before the blocking operation completes. However, this potential for non-atomicity is limited to running the task(s) that are directly or indirectly blocked on, and it does not occur in cases where those tasks have definitely finished prior to the `getValue` or `join` operation. Also, a deterministic computation (discussed below) effectively executes atomically, as it is semantically equivalent to a sequential execution with no task-related operations. As in languages with explicit atomic constructs, it remains the programmer's responsibility to identify sections of code that should behave atomically and write the code in a way that ensures they do so, e.g. by not using `getValue` or `join` operations within such sections.

4.4 Deadlock avoidance

Our model avoids deadlocks in the case that a task A directly or indirectly blocks on another task B whose effects conflict with A's effects, using the effect transfer mechanism discussed in section 2.4. While we do not prevent all deadlocks, we believe this class of deadlocks is significant, and we found our effect transfer mechanism to be useful in practice, particularly in the interactive FourWins program (see section 6).

4.5 Determinism

Many parallel algorithms are deterministic. That is, they always produce the same output given the same input state. Since this is an expected property of many algorithms, detecting violations of it is a useful way of finding bugs. Moreover, knowing that a program or an algorithm is deterministic makes it much easier to reason about: the user of the program or algorithm knows that it will always produce the same output given the same input, so they need not be concerned that different parallel interleavings of operations may produce different results. Determinism also makes a program or algorithm much simpler to debug, since one knows that the same result will be produced every time it is run with a given input.

DPJ [13] can provide a compile-time guarantee of determinism using the combination of its type and effect system and simple parallelism constructs supporting only fork-join patterns of parallelism. We provide a similar static guarantee of determinism for deterministic algorithms or programs written in TWEJava. All programming patterns for which DPJ can give a guarantee of determinism can also be expressed and proven deterministic using the tasks with effects model. Our model also allows us to give a static guarantee of determinism for certain computations in a program while still allowing the rest of the program to use the full flexibility of TWEJava (including non-fork-join concurrency structures), and guaranteeing our other safety properties for the whole program. Thus, our feature for guaranteed determinism can be used within programs that could not be expressed with DPJ.

To request that the compiler check and enforce the determinism of a certain task or method, the programmer can annotate it as `@Deterministic`. In code that has this annotation, the compiler will enforce that the only task-related operations used in the code are the `spawn` and `join` operations described in section 2.5. Also, code annotated as deterministic may only call other deterministic methods and spawn other deterministic tasks.

These restrictions ensure that the code invoked from a deterministic task or method (including through the creation of other tasks) accesses memory only as specified by its declared effects. Moreover, there is a defined order by which control of each region covered by those effects is transferred between tasks, as determined by `spawn` and `join` operations. (Note that the form of effect described in section 2.4 will never be needed for `join` operations within a deterministic computation, and thus will not occur.) Therefore, for a given input state of the memory in regions covered by the effects of the deterministic task or method, there is a deterministic output state that will not vary between executions of the code. This state is the same as the state produced if the code were executed sequentially with each task's code run at the point where the task is spawned. These deterministic computations are also deadlock-free.

5. Compiler and Runtime System

Our implementation of TWEJava consists of a compiler and a runtime system, which we briefly describe here.

5.1 Compiler

The compiler is based on the DPJ compiler, which checks that effect declarations are correct and that types are used correctly. Our extended version also supports the new features of TWEJava described in Section 2. These include generating code to record effect parameters and some region parameters for use at run time; performing a data flow analysis to determine the covering effects for each operation (accounting for operations that do effect transfer); and checking the use of the `@Deterministic` annotation.

To enable interoperation with existing Java code and libraries that do not have region and effect annotations (including the Java standard libraries), the compiler allows methods without effect annotations to be called within methods that have effect annotations. This produces a warning, but that warning can be suppressed for individual methods. Since we have not written an extensive standard library for TWEJava, we take advantage of this capability to use Java standard library features such as the Swing GUI system, I/O routines, and math functions. The compiler cannot give a full guarantee about the correctness of code making such calls, so the programmer has to manually reason about it, but that reasoning can be encapsulated by writing annotated wrapper methods that internally call unannotated library routines.

5.2 Runtime System

Code generated by our compiler can be run using our runtime system, which implements the various task-related operations in TWEJava. We use an effect-based scheduler to enforce our model's key property of task isolation. Our current prototype implementation uses a queue of tasks protected by a single lock to manage the effect-checking phase of task scheduling. The effect-based scheduler enables a task for execution only once it is safe to do so based on its effects. Once a task is enabled for execution by our scheduler, it is handed off to a version of the Java `ForkJoinPool` framework, which is responsible for actually executing tasks using a thread pool.

When attempting to execute a task, our implementation generally works by scanning from a task's position forward toward the head of the queue (which includes both running and waiting tasks), checking if the task's effects conflict with those of each task ahead of it. If a conflicting task is found when attempting to schedule a

task, then the later task is marked as waiting for the earlier one to complete. This approach will generally run conflicting tasks in the order that they were enqueued, but there is also a mechanism for prioritizing tasks that a running task is blocked on.

We show below that with this relatively simple scheduling approach we can achieve substantial speedups on a range of benchmarks. However, the tasks with effects model could also be implemented with other more scalable scheduling approaches. In particular, if we associated information about enqueued tasks with regions, then tasks with effects on unrelated regions would not need to have their effects explicitly compared against each other, and we could also avoid the need for a single global task queue lock.

6. Evaluation

We have carried out an evaluation of the tasks with effects model and our TWEJava language by porting several concurrent programs to it and writing one new one from scratch. We are principally concerned with demonstrating that TWEJava and the tasks with effects model can express a variety of concurrent programming styles used in real-world applications, but we also show that substantial parallel speedups can be achieved with our current TWEJava implementation.

6.1 Expressiveness

We ported four existing concurrent programs to TWEJava and wrote one new application in it. The first ported program is an interactive Connect Four game implementation called FourWins, which was ported from an original code that used JCoBox [37], an actor-like concurrent programming system for Java. The FourWins code is structured in terms of modules that behave similarly to actors, including the game state, board state, game controller, GUI view, and human and computer players. These modules communicate by sending messages between each other, sometimes, but not always, blocking until the message is processed. Our general approach in most parts of this code was to introduce a region holding the data for each module, and to define a number of types of tasks corresponding to each message that may be sent to that module. Those tasks have either read or write effects on the module's region, as needed. This code also includes a parallel computation in the computer player's AI, to explore the tree of possible future moves. That recursive parallel computation consumes most of the execution time, and it is the portion for which we report performance results below. We note that the complex concurrency structure of this program, with code from multiple actors running concurrently and sending messages between each other, cannot be expressed in many more restrictive parallelism models that require structured parallelism (e.g. fork-join) or involve a single conceptual flow of control.

The other interactive GUI application we implemented is an image editing application called ImageEdit, which we wrote from scratch in TWEJava. It allows the user to open one or more images and apply various image editing filters to them. Each of the images is displayed in a separate window and updated as filters are applied to it. Each image has a region associated with it, and the actual pixel data for the image is broken up into a 2-D grid of blocks, with the data for each block placed in a separate region using index-parameterized arrays. (By default, and in our benchmarks, a block is simply a group of adjacent lines totaling about 100,000 pixels, but the user may specify other block dimensions.) Concurrency is possible both by doing concurrent operations on different images and by operating in parallel on one image at the level of blocks. ImageEdit currently includes filters for Gaussian blur, sharpening (unsharp mask), detecting edges in the image (based on the Canny edge detection algorithm [15]), darkening or brightening the image, and converting it to grayscale. All of the filters can use parallelism at the level of blocks, sometimes using several computation steps in sequence with parallelism in each step. The only non-parallel step in any of them is a short final step in the edge detection filter to identify edges in the input image that cross between two different blocks. Computation in this program is driven by user input events, so the program as a whole does not follow the fork-join computation model required by systems like DPJ. It could be written in other task-based concurrency models that do not use effects, but these would not provide the strong safety guarantees of TWEJava and would require the programmer to manually ensure that tasks performing conflicting memory operations cannot run concurrently.

The other three benchmarks were previously written in DPJ [13], and we ported our versions from the DPJ versions, following a similar pattern of regions and effects. These are the force computation from a Barnes-Hut n-body simulation; a k-means clustering algorithm (originally adapted from STAMP); and a Monte Carlo financial simulation, originally from the Java Grande parallel benchmarks. These three benchmarks allow us to evaluate the impact of the run-time scheduling overheads in our system by comparing against the original DPJ versions, which do not have any overheads related to effect-based scheduling at run time.

The Barnes-Hut force computation involves a parallel loop over a set of bodies, computing and adding up the forces on each body due to the other bodies. We create one task per thread using the `spawn` operation, each operating on a portion of the total set of bodies, which is divided using an index-parameterized array. The resulting computation is deterministic and has good parallelism.

The Monte Carlo simulation includes a deterministic parallel loop to compute an array of results, followed by a reduction step that updates globally shared data. In the DPJ version, this reduction step used DPJ's commutative annotation, which represents an unchecked assertion from the programmer that two invocations of a certain method are commutative and that it internally uses the necessary locking to correctly synchronize concurrent invocations. In the TWEJava version, this commutative method is replaced by a task, and our system automatically guarantees that this task behaves atomically. Thus, TWEJava offers a stronger safety guarantee than DPJ, since it does not require the programmer to correctly insert manual locking operations. As with Barnes-Hut, we create one task per thread in the parallel loop.

The k-means computation involves a parallel loop with a reduction step. In the original STAMP code, this reduction step is an atomic block, but in the DPJ version it is a commutative method with internal locking. In TWEJava, it is a task. As in Monte Carlo, the DPJ version relied on unchecked, manual locking, so TWEJava offers a stronger safety guarantee than DPJ. The structure of the reduction computation in k-means requires that we create many reduction tasks, independent of the number of threads.

We were able to express all the parallelism that was present in the original codes that we ported. Both the `executeLater`/`getValue` operations and structured parallelism with `spawn` are used in our benchmarks. The former are necessary for unstructured parallelism such as messaging between actors or modules, and for defining tasks that behave like atomic or synchronized blocks, while the latter can be used in parallel loops or recursive parallel computations.

6.2 Performance

We measured the performance of our benchmark codes on a machine with four Intel Xeon E7-4860 processors (40 total cores, 80 hardware threads using Hyper-Threading) and 128 GB of memory, running Scientific Linux 6.3 with kernel 2.6.32 and 64-bit Oracle JDK 7u9. Figures 5 and 6 report the speedups achieved in the parallel portion of each code. For ImageEdit, we report speedups for

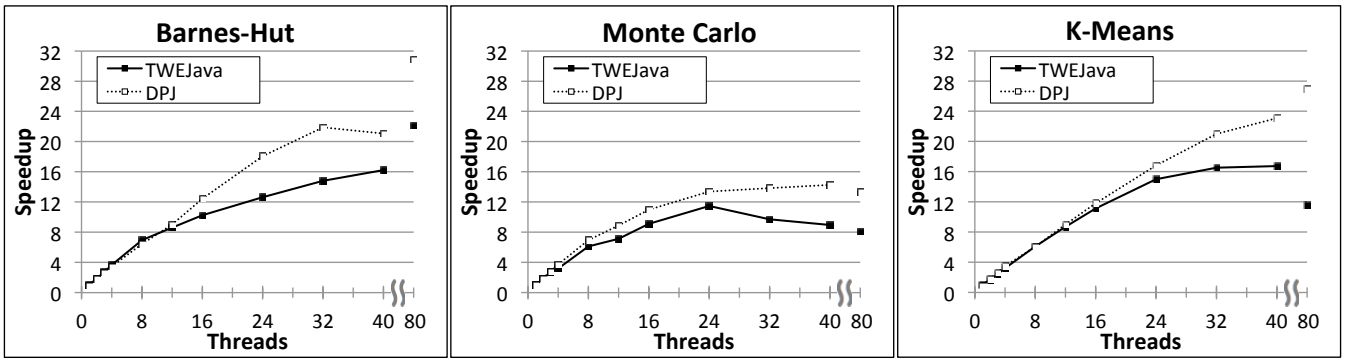

Figure 5. Parallel speedups of benchmarks ported from DPJ, showing performance of TWEJava and DPJ versions. These speedups are for the parallel portion of each code and are relative to the DPJ code compiled and run in sequential mode, in which the DPJ parallelism constructs are erased by the compiler, creating a sequential program with no run-time overheads related to parallelism constructs.

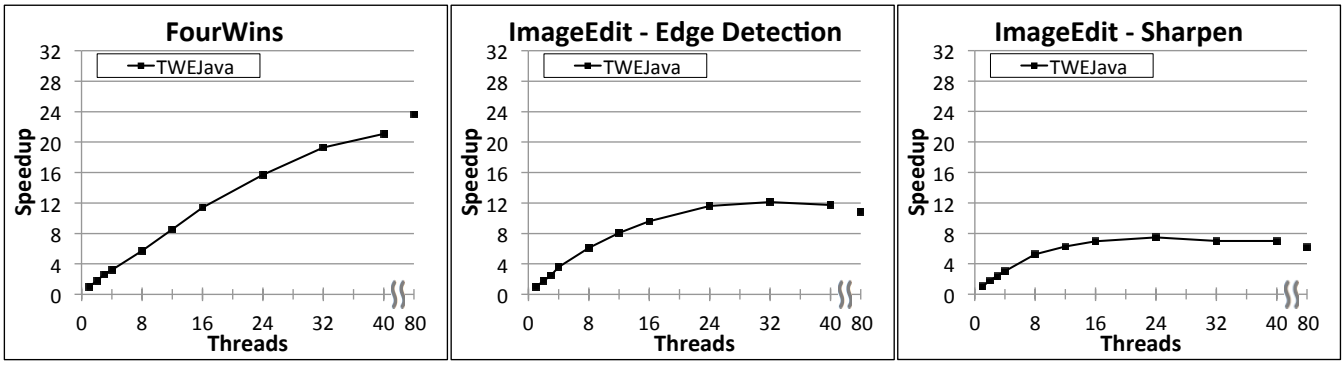

Figure 6. Speedups for the FourWins AI computation and two filters in the ImageEdit application. We did not have pure sequential versions of these programs available for comparison, so we give speedups relative to the TWEJava codes run using one worker thread and configured so that the major potentially-parallel computations in the codes each run as a single task, thereby minimizing task-related overheads.

both the edge detection filter and the sharpening filter. We also compared the parallel running times to DPJ for the codes where there is a DPJ version. The multi-threaded DPJ version internally executes tasks on a thread pool, but it does not have the overhead of run-time effect-based task scheduling, and previous work has shown it is generally quite efficient [13].

Each of our TWEJava benchmarks achieves significant speedups, with maximum speedups on the various benchmarks ranging from 7.5x to 23.6x. The Barnes-Hut and FourWins benchmarks continue scaling substantially up to 80 threads (with the gains going from 40 to 80 threads attributable to Hyper-Threading). The other benchmarks show good scaling at lower numbers of threads, but do not continue scaling above 24 to 32 threads.

The benchmarks for which we have DPJ versions perform very similarly to the DPJ versions up to at least eight threads, but show worse scaling at high numbers of threads. Several types of overhead in the TWEJava system may be responsible for these performance differences. The overheads of our effect-based run-time task scheduling system include the need to check the effects of tasks against each other to see whether they conflict, and the need to track some region parameters and all effect parameters at run time, rather than erasing them during compilation. These overheads can become larger with larger numbers of threads, because there will generally be more tasks active at once in such configurations. Another important factor limiting the scalability of our current implementation is that our effect-based scheduler uses a single queue protected by a single lock, so all the effect-based scheduling operations in the system are essentially serialized. Also, the DPJ runtime system can use recursive subdivision to split the iterations of parallel loops into tasks, while in TWEJava we converted these constructs to loops that sequentially spawn off child tasks. This may also contribute to the inferior scalability of the TWEJava codes. It would be possible to implement this sort of recursive subdivision in the tasks with effects model, but TWEJava currently does not have convenient language constructs for it.

We believe the overheads of effect-based task scheduling are particularly important factors in explaining the inferior scaling of our version of KMeans compared to the DPJ version on large numbers of threads, because the TWEJava version uses a task rather than a locked block for the reduction step. This is called a large number of times, regardless of the number of threads (550,000 times in our benchmark configuration). Since task scheduling is a heavier-weight procedure than simple locking around a short block, and particularly since (as noted above) the scheduling of each task is effectively sequentialized in our current implementation, this leads to poorer scalability for the TWEJava version of the code.

In the case on ImageEdit, one factor limiting the speedups achieved is that each time the image is updated, some sequential operations are necessary to actually change the image displayed in the GUI, which is implemented with Java's Swing framework and therefore needs to do GUI operations on a single thread, in accordance with Swing's architecture. This is a larger factor for the

sharpening operation than for the edge detection operation, since the core parallel computation for sharpening is faster than for edge detection. We believe this at least partially accounts for the poorer scalability of sharpening compared to edge detection, as well as the overall scalability limits of the ImageEdit computations.

While our system has run-time overheads related to task scheduling and dynamic tracking of region and effect parameters, it still delivers significant parallel speedups, sometimes comparable to the DPJ versions of the codes (particularly on relatively low numbers of threads). We believe the scalability and performance of our system could be improved by implementing a scheduler that does not use a single lock and a compiler and scheduler that work together to minimize the number of dynamic effect comparisons (e.g. by avoiding the need for run-time checking of covering effects when child tasks are spawned in a loop). However, we think our current implementation without these optimizations still gives good enough performance to be used in many applications, particularly in desktop and mobile systems with relatively low numbers of cores.

7. Related Work

Traditional multithreaded systems such as Java or Posix threads are more flexible than TWEJava in the sense that they allow almost any desired concurrency and synchronization structure to be expressed, but they provide no guarantees about the absence of concurrency errors, and also have few or no facilities that simplify reasoning about such errors. Some systems, including OpenMP [32], Cilk [11], Threading Building Blocks (TBB) [22] (except for the "lower-level" task interfaces), and Java's `ForkJoinTask` [33] are more structured and easier to reason about than traditional threads. However, these systems still do not provide any correctness guarantees such as data race freedom or determinism. The programmer still has to reason manually to ensure that data sharing patterns are correct and synchronization is present when needed. These systems simplify such reasoning by limiting programs to use a particular parallelism structure (e.g. fork-join), but in doing so, can no longer express the forms of concurrency required by many programs such as interactive applications, servers, and actor-style programs. TWEJava is able to express all these kinds of programs and yet provides strong correctness guarantees.

RCCJava [18] can ensure data race freedom, but it does not provide structured concurrency constructs or guarantee other safety properties such as determinism. SharC [5] allows flexible concurrent control flow while providing a guarantee of data race freedom, but it also does not provide structured concurrency constructs and cannot guarantee stronger properties like determinism. CoreDet [8], Kendo [31], Grace [9], and DMP [16] allow multithreaded programs to be executed with a deterministic execution order that does not vary from run to run, but they do not provide structured parallelism constructs, and the deterministic execution order they provide is not related in an obvious way to the program code and may change if the code or input changes, which limits their utility as tools for reasoning about program behavior.

Many parallel and concurrent programming systems provide various correctness guarantees but have weaker expressive power than TWEJava. These include Jade [35], Prometheus [4], DPJ [13, 14], OoOJava [23], Dynamic Out-of-Order Java (DOJ) [17], Pāñini [27], SvS [10], Legion [7], and Ke et al.'s system for parallelization with dependence hints [25]. Several of these systems, including Jade, Prometheus, OoOJava, DOJ, and Pāñini, guarantee deterministic semantics (often with equivalence to a unique sequential program) but these systems are unable to express inherently nondeterministic algorithms, or programs where concurrency is due to external requests or user input and the input and its timing may affect the program's results. SMPSs [34] is also designed to provide sequential-equivalent semantics and uses a form of effect annotations for task scheduling, but these annotations are not verified, so the programmer is responsible for ensuring that the annotations are correct in order to ensure proper program behavior. Several systems, including (at least) Jade, SvS, Legion, DOJ, SMPSs, and Aida [28], have used effects in some form to guide run-time scheduling decisions, but TWEJava provides the ability to express programs not supported by any of these other languages and gives stronger safety guarantees than some of them.

DPJ, Legion and SvS can express nondeterministic programs, but not programs requiring flexible concurrency structures, identified above. DPJ supports programs with both deterministic and nondeterministic algorithms, and provides the strongest parallel correctness guarantees we know of, but because it is limited to fork-join parallel structures, it is not suitable for many concurrent programs. TWEJava supports a much broader class of programs than DPJ, and provides almost as strong correctness guarantees: its primary weakness compared to DPJ is that it only provides limited protection from deadlocks. Like DPJ, Legion cannot express programs with general concurrency and synchronization patterns because there are no mechanisms for explicit "join" synchronization between tasks (tasks block for other tasks only due to interfering effects, enforced by the scheduler) and the effects of a parent task must be a superset of the effects of its child tasks. Legion also provides significantly weaker correctness guarantees than DPJ or TWEJava, although it allows more dynamic assignment of data to regions, and explicit program management of locality via region maps. SvS executes tasks according to a statically-defined task graph, which limits the language to a narrower range of concurrent applications than TWEJava. SvS allows both deterministic and nondeterministic algorithms, and guarantees data race freedom to such programs. One key difference is that SvS *infers* potential conflicts due to implicit sharing of data between tasks, and uses an approximate run-time analysis of the memory possibly accessed by a task. While these features reduce the annotation burden on the programmer, they increase the likelihood of spurious dependences ("false positives") that prevent two tasks from executing in parallel. TWEJava does not suffer from such false positives when checking for effect interference between tasks.

Transactional memory systems [19] use speculative execution to enforce correctness guarantees such as atomicity. These systems guarantee that atomic blocks declared by the programmer execute in isolation from each other, performing rollback and retry if necessary. To date, implementations have often relied on software transactional memory (STM). STM systems generally have high overheads, stemming from the need to track memory accesses and check for conflicts, combined with wasted computation when rollbacks occur. In contrast, TWEJava only requires conflict checks (on task effect summaries) before a task begins execution and never rolls back partially-completed tasks. Also, to avoid exorbitantly high overheads, many STM systems only guarantee isolation between two atomic blocks (weak isolation). In these systems, statements outside atomic blocks may still race with other statements inside or outside atomic blocks, so there is not a full guarantee of data race freedom.

Several other systems also use optimistic parallelism. Galois [26] focuses on irregular algorithms and requires the programmer to specify which operations are semantically commutative and define inverse methods for use on rollback. Non-deterministic algorithms in DPJ also use atomic blocks implemented via an STM system, which has fairly poor absolute performance [14]. Aida [28] also focuses on irregular parallelism. It guarantees the absence of data races, deadlock and livelock, via a mechanism called "delegated isolation," where a task that conflicts with another concurrent task is rolled back and then "delegates" all its computation and data

to the latter task. Galois, DPJ and Aida are all limited to highly structured, fork-join concurrency.

A more flexible style of concurrent programming is *actors* [3]. In the basic actor model, a concurrent system is composed of several actors, each potentially having local state, but no shared state between the actors. Actors communicate by sending messages to other actors, and computation is done at each actor in response to the messages received. Each actor processes only one message at a time, so all concurrency is due to the simultaneous execution of different actors. Actor-style programs are natural to express using our system: a region can be defined to correspond to each actor, and tasks with effects on that region can be thought of as equivalent to messages sent to and processed by that actor. Several actor-like programming models for shared memory systems [24, 29, 37] broaden the basic actor model to include some form of shared state between actors, but these systems are generally less flexible than our effect system, and in some cases do not guarantee data race freedom. Our system, when used to write actor-style programs, can express both shared state between actors and internal concurrency within actors, while guaranteeing data race freedom as well as, where desired, deterministic, sequential-equivalent semantics for parallel algorithms used within an actor.

8. Conclusion

We have described and defined the semantics of a new concurrent programming model based on tasks with effects, and presented a language called TWEJava that implements it. TWEJava can express a wide range of concurrent and parallel programs, while delivering very strong safety properties including task isolation, data race freedom, atomicity, and optionally determinism. We have implemented several concurrent programs in TWEJava and shown that our present implementation can give substantial parallel speedups.

Acknowledgments

This work was funded by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp., Special Issue on Shared-Mem. Multiproc.*, pages 66–76, December 1996.

[3] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, 1986.

[4] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *PPOPP*, 2009.

[5] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *PLDI*, 2008.

[6] Apple. Concurrency Programming Guide. `http://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgrammingGuide/`, Dec. 2012.

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12*, 2012.

[8] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[10] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: Techniques for efficiently managing shared state. In *PLDI*, 2011.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPOPP*, 1995.

[12] R. L. Bocchino and V. S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *ECOOP*, 2011.

[13] R. L. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.

[14] R. L. Bocchino, S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[15] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 8(6):679–698, June 1986.

[16] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[17] Y. h. Eom, S. Yang, J. C. Jenista, and B. Demsky. DOJ: Dynamically parallelizing object-oriented programs. In *PPoPP*, 2012.

[18] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, 2000.

[19] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition (Synthesis Lectures on Comp. Arch.)*. Morgan & Claypool, 2010.

[20] S. Heumann and V. Adve. Disciplined concurrent programming using tasks with effects. In *HotPar*, 2012.

[21] S. Heumann and V. Adve. Tasks with effects: A model for disciplined concurrent programming. In *WoDet*, 2012.

[22] Intel. Intel Thread Building Blocks Reference Manual. `http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf`, Aug. 2011.

[23] J. C. Jenista, Y. h. Eom, and B. C. Demsky. OoOJava: software out-of-order execution. In *PPOPP*, 2011.

[24] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In *Principles and Practice of Programming in Java (PPPJ)*, 2009.

[25] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *OOPSLA*, 2011.

[26] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[27] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *Generative Programming and Component Engineering (GPCE)*, 2010.

[28] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *OOPSLA*, 2011.

[29] Microsoft. Axum. `http://msdn.microsoft.com/en-us/devlabs/dd795202`.

[30] Microsoft. Task Parallel Library (TPL). `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.

[31] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[32] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011.

[33] Oracle. Java Platform, Standard Edition 7 API specification. `http://download.oracle.com/javase/7/docs/api/`.

[34] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *IEEE International Conference on Cluster Computing*, 2008.

[35] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS*, 20(3):483–545, May 1998.

[36] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6), 2010.

[37] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP*, 2010.