© 2016 by Stephen Thomas Heumann. All rights reserved.

THE TASKS WITH EFFECTS MODEL FOR SAFE CONCURRENCY

BY

STEPHEN THOMAS HEUMANN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Vikram Adve, Chair and Director of Research
Professor Sarita Adve
Professor David Padua
Associate Professor Madhusudan Parthasarathy
Associate Professor Luis Ceze, University of Washington

Abstract

Today's state-of-the-art 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. I believe that a concurrent programming model should offer strong safety guarantees such as data race freedom, atomicity, and optional determinism, while being flexible enough to express the wide range of uses for concurrency in realistic programs, and offering good performance and scalability. In my thesis research, I have defined a new concurrent programming model called *tasks with effects (TWE)* that is intended to fulfill these goals. The core unit of work in this 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, the TWE model is able to guarantee strong safety properties such as data race freedom and atomicity.

I have implemented this programming model in a language called TWEJava and its accompanying runtime system. I have evaluated TWEJava's expressiveness by implementing several programs in it. This evaluation shows that it can express a variety of parallel and concurrent programs, including programs that combine unstructured concurrency driven by user input with structured parallelism for performance, a pattern that cannot be expressed by some more restrictive models for safe parallel programming.

I describe the TWE programming model and provide a formal dynamic semantics for it. I also formalize the data flow analysis used to statically check effects in TWEJava programs. This analysis is used to ensure that the effect of each operation is included within the covering effect at that point in the program. The combination of these static checks with the dynamic semantics

provided by the run-time system gives rise to the TWE model's strong safety guarantees.

To make the TWE model usable, particularly for programs with many fine-grain tasks, a scalable, high-performance scheduler is crucial. I have designed a highly scalable scheduling algorithm for tasks with effects. It uses a tree structure corresponding to the hierarchical memory descriptions used in effect specifications, allowing scheduling operations for effects on separate parts of the tree to be performed concurrently and without explicitly checking such effects against each other. This allows for high scalability while preserving the expressiveness afforded by the hierarchical effect specifications. I describe the scalable scheduling algorithm in detail and prove that it correctly guarantees task isolation.

I have evaluated this scheduler on six TWEJava programs, including programs using many fine-grain tasks. The evaluation shows that it gives significant speedups on all the programs, often comparable to versions of the programs with low-level synchronization and parallelism constructs that provide no safety guarantees.

As originally defined, the TWE model could not express algorithms in which the side effects of each task are dependent on dynamic data structures and cannot be expressed statically. Existing systems that can handle such algorithms give substantially weaker safety guarantees than TWE. I developed an extension of the TWE model that enables it to be used for such algorithms. A new feature of the TWEJava language is defined to let programmers specify a set of object references that define some of the effects of a task, which can be added to dynamically as the task executes. The static covering effect checks used by TWE are extended to ensure they are still sound in the presence of such dynamic effect sets, and the formal dynamic semantics for the TWE model are extended to support these dynamic effects.

I describe and implement a run-time system for handling these dynamic effects, including mechanisms to detect conflicts between dynamic effects and to abort and retry tasks when such conflicts arise. I show that this system can express parallel algorithms in which the effects of a task can only be expressed dynamically. Preliminary performance experiments show that my dynamic effect system can deliver self-relative speedups for certain benchmarks, but the current version of the system imposes substantial performance overheads. I believe that these overheads can be significantly

reduced, making TWE practical to use for programs that require dynamic effects; this is a focus of my ongoing and future work.

Dedicated to my grandparents.

Acknowledgments

I am grateful to many people for their support and guidance during my PhD and their contributions to the research described in this thesis:

- My advisor, Vikram Adve, supported, encouraged, and guided me throughout the PhD process and contributed numerous ideas to my research.
- My collaborators on the DPJ project, and in particular its leader Rob Bocchino, laid a strong foundation of ideas that I have built on in this work. These include both the general principle of the importance of disciplined parallelism with strong safety guarantees, and also the specific region-based type and effect system which I adopted from DPJ into TWEJava. They also created the DPJ compiler supporting that type and effect system, which I built upon to develop the TWEJava compiler. Moreover, the DPJ project provided me with my first experience of graduate-level computer science research, and it exposed me to many interesting ideas that helped to shape my research interests.
- Danny Dig encouraged me and gave me several opportunities to share my research and contribute to the community.
- My doctoral committee members and numerous other people at Illinois provided valuable feedback about my research.
- Alexandros Tzannes helped with the benchmarks and evaluation of TWEJava, and gave his advice on various aspects of my research.
- My undergraduate collaborators Shengjie Wang and Lance Strait helped to port several benchmark programs in order to evaluate TWEJava.

- Finally, I wish to thank my parents and the rest of my family for their love and support.

I also wish to thank Intel, Microsoft, and the NSF for funding my research assistantship at various times during my PhD. I was funded for some time by the Universal Parallel Computing Research Center (UPCRC) at Illinois, a center supported by Intel and Microsoft, and later by the Illinois-Intel Parallelism Center (I2PC) supported by Intel.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

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. Servers use concurrency to respond to multiple client requests. It can also be convenient to express a full program as a set of of modules or actors [6] 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. A server may also combine concurrency used to handle multiple client requests with parallelism that may be needed to quickly process an individual request. I 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 [4] can affect the semantics of programs written in this style, further complicating the task of reasoning about them.

These weaknesses of today's widely-used concurrent programming models contribute greatly to the difficulty of writing correct concurrent programs, acting as a drag on programmer productivity and providing many opportunities for subtle errors. Programmers commonly have to reason manually about whether memory accesses are correctly synchronized to avoid data races, and whether the granularity of synchronization is appropriate to ensure that operations that are expected to behave atomically actually do so. In designing this manual synchronization, programmers also must be careful not to give rise to deadlocks.

A particularly important problem is that a program's behavior may vary from run to run in unexpected ways because the interleaving of operations may be different in every run, even in cases where the programmer intends the behavior to be deterministic. This makes it much more difficult to reason about the behavior of concurrent programs than about sequential ones. In combination with the need for manual, error-prone concurrency and synchronization design described above, it also provides ample opportunity for rare concurrency errors that may manifest themselves only occasionally when the program is run. These errors can be very difficult to debug because they cannot be consistently reproduced, and can easily escape into production code because they are rarely or never triggered in test environments.

It today's environment, where multi-core processors are ubiquitous and thus parallelism must be used to exploit the full computational power even of mobile devices, these difficulties of concurrent programming are an important source of errors and a limiting factor in programmer productivity. In light of this, I believe that defining concurrent programming models with strong safety guarantees is an important way to increase programmer productivity and reduce the risk of programming errors.

Many previous systems have attempted to address aspects of these problems, or offer more restricted programming models that avoid them. Functional programming models like Concurrent Haskell [74] and ML [76] avoid many classes of concurrency errors by using side-effect-free functions,

and can elegantly express various parallel algorithms. However, the lack of side effects on mutable data hinders them from efficiently expressing a wide range of computations, as a copy of a potentially large data structure must be made whenever any changes to it are necessary. Dataflow languages such as Sisal [37] have similar limitations, requiring computations to be expressed in terms of a dataflow graph containing explicit dependences from the outputs of one node to the inputs of another, with no notion of mutable shared memory.

Data-parallel language such as CMF [81], HPF [50], and C* [45] can express programs with data-parallel loops while avoiding most of the safety problems discussed above (by giving generally sequential-equivalent semantics). However, these languages are focused on structured parallel computations with parallel loops, and are not suitable for expressing general, potentially unstructured concurrent programs.

Some other languages 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 [20] and Thread Building Blocks (TBB) [52], for example, provide structured parallelism constructs, but they do not offer checked guarantees of strong safety properties such as data race freedom. Moreover, fork-join programming models such as Cilk do not support the general, event-driven form of concurrency required by interactive applications or actor-style programs.

Some other systems do seek to offer stronger guarantees. The Deterministic Parallel Java (DPJ) language [22, 23] 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 I seek. Most critically, DPJ is restricted to fork-join parallelism structures, which are not suitable for many concurrent programs. Because it can only express fork-join parallelism, it excludes cases like pipelined computations or algorithms with more general task graphs [5]. It also is not suitable for expressing programs where potentially concurrent pieces of work are triggered by external, asynchronous input (e.g. from the user or a client) while the program is running.

In addition, DPJ relies on a purely static type system to enforce its correctness requirements, and many algorithms cannot be checked with DPJ's type system, e.g. graph-based algorithms, or parallel algorithms that require array reshuffling or tree rebalancing.

I believe that concurrent programming models should offer strong safety guarantees similar to DPJ, including data race freedom, atomicity, and optional determinism, but be flexible enough to express the wide range of uses for concurrency in realistic programs, and be implementable with low performance overheads and high scalability. In my thesis research, I have worked to define and implement a new concurrent programming model called *tasks with effects (TWE)* that can accomplish these goals.

1.2 The Tasks with Effects Programming Model

The tasks with effects programming model is designed to give strong safety guarantees while providing the flexibility needed to express a wide range of concurrent programs in it. 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 possible for one task to await the completion of another. A scheduler is responsible for executing tasks in an efficient manner. Tasks provide a valuable abstraction for concurrency, 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 [10], Microsoft's Task Parallel Library in .NET [68], OpenMP [71], and the `ForkJoinTask` framework in Java 7 [72]. However, these systems 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.

I propose instead to associate a checked effect specification with each task, describing (possibly conservatively) the side effects of executing the 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 my work so far I have adopted the statically-checked effect system developed for Deterministic Parallel Java [22]. 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 a dynamic effect-based task scheduler, my system can 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.

The tasks with effects model also includes mechanisms that allow the compiler to give a static guarantee that certain structured parallel computations are deterministic, which can help the programmer to more easily reason about those computations and ensure they do not contain any unexpected sources of nondeterminism. This guarantee can apply to specific computations with a larger program that is not completely deterministic (which is commonly the case for many interactive applications or servers). In this case, the whole program can be expressed with a unified task-based programming model, and the other guarantees of the TWE model will apply to the program as a whole.

The tasks with effects model provides strong safety guarantees while supporting the flexible control flow needed to express general concurrent programs such as interactive applications and actor-like programs. I am aware of no other concurrent programming model that can provide a similar level of flexibility to express general concurrent programs while giving the strong safety guarantees that TWE provides.

1.3 Contributions

My work has focused broadly on defining the tasks with effects programming model and building a practical, efficient implementation of it. It has several major components:

- I have defined the tasks with effects (TWE) programming model, which provides the flexibility to express a wide range of concurrent programs while giving strong safety guarantees. I have given a formal dynamic semantics for the key run-time operations of the tasks with effects

model, and also formalized the static data flow analysis that is used to verify that the effect of each operation in a program will be covered by the effects of the task running it at the point when it is executed. The combination of these static and dynamic components gives rise to the TWE model's strong safety properties.

- I have designed and implemented the TWEJava language as a practical implementation of the TWE model, built as an extension of Java and incorporating a hierarchical region-based effect system adapted from DPJ.

- I have designed a high-performance, scalable scheduling algorithm for tasks with effects. It uses a scheduling tree designed to take advantage of the hierarchical structure of TWE effects. I have implemented this algorithm in the TWEJava runtime system.

- I have evaluated the expressivity and performance of the TWEJava system using several benchmark programs ported to or written in TWEJava. The evaluation shows that the TWE model can express a variety of parallel and concurrent programs and (with the tree scheduler) provide generally good performance and scalability for them.

- I have extended the TWE model to support programs where the effects of tasks can only be computed dynamically, as the task is executing. This is an important class of programs that the initial version of the TWE model could not support. To add support for them, I define a mechanism that permits the effects of tasks to be dynamically modified in certain ways. I have extended the TWE formalism to support this mechanism, and also produced an initial implementation of it for TWEJava. This initial implementation gives self-relative speedups for some programs, although it has significant overheads that I am currently working to reduce.

My work on defining the TWE model and implementing the initial version of TWEJava (using a relatively naive effect-based task scheduler, not my current high-performance design) is described in a paper presented at PPoPP 2013 [48], with portions also described in two earlier workshop papers [47, 46]. My work on the scalable, tree-based scheduling algorithm for TWE was presented

at PACT 2015 [49].[1]

1.3.1 The Tasks with Effects programming model

The first major area of contributions is to define and formalize the tasks with effects programming model, in which tasks with effect specifications are scheduled by an effect-aware run-time scheduler. This model gives a guarantee of *task isolation*: that two tasks with conflicting effects will not be run concurrently.

This model could potentially be implemented using a variety of different effect systems, and effect specifications might be dynamically checked, statically checked, or unchecked. In my work so far, I have focused on using effect specifications that are statically checked to ensure they cover the actual memory effects of each task (possibly conservatively). When statically checked effect specifications are used, the combination of these static checks and the guarantee of task isolation given by the run-time task scheduler leads to a guarantee of data race freedom, and to a guarantee that portions of tasks that do not wait for or create tasks are atomic.

In addition to the basic features of creating and waiting for tasks, the TWE model that I have defined also includes mechanisms based on *effect transfer* between tasks. 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 a system based on the TWE model to provide a compile-time guarantee of determinism for a class of deterministic programs and algorithms similar to those supported by DPJ, avoiding the need for run-time effect checking in those cases.

In additional to informally describing the TWE mode, I have provided a formal dynamic semantics of tasks with effects (expressed using the K semantic framework). I have also formalized the data flow analysis used to statically compute a conservative approximation of the covering effect at each point in the program. This is used to ensure that the effect of each operation will always be covered by the effect of that task executes it at the point when it executes. I describe how

[1] Portions of this dissertation are based on material that appeared in these PPoPP 2013 and PACT 2015 papers. The PPoPP 2013 paper is copyright 2013 ACM (`http://doi.acm.org/10.1145/2442516.2442540`) [48]. The PACT 2015 paper is copyright 2015 IEEE (`http://dx.doi.org/10.1109/PACT.2015.25`) [49].

the combination of these static and dynamic check guarantees the strong safety properties of the TWE model, including task isolation, data race freedom, atomicity, and (for certain computations) determinism.

The tasks with effects programming model and its dynamic semantics are described in chapter 3, along with its implementation in the TWEJava language. Chapter 4 describes the data flow analysis to compute covering effects.

1.3.2 The TWEJava language

I have designed the TWEJava language, which implements the TWE model using an extended version of the Java language, and implemented it through a compiler and run-time system. TWEJava uses a region-based effect system based on DPJ's, and its compiler (based on the DPJ compiler) statically verifies the effect specifications of methods and tasks, as well as recording information about the effects of tasks for use at run time.

The TWEJava run-time system is responsible for running the tasks composing a TWEJava program, ensuring that task isolation is maintained (i.e. no two tasks with conflicting effects are run concurrently). My initial implementation of the run-time system used a relatively naive task scheduler design to accomplish this; I have since replaced it with a much higher-performance task scheduler, as described next.

1.3.3 An efficient scheduling algorithm for tasks with hierarchical effects

I have defined a high-performance, scalable algorithm for scheduling tasks with effects, and implemented it in the TWEJava run-time system. It takes advantage of the hierarchical structure of effects in TWEJava by using a tree structure to track the effects of running tasks. This tree structure is used to limit the number of effect comparisons that need to be done to detect conflicting effects. It also enables the use of fine-grained locking in scheduler operations, so operations on different parts of the tree can proceed concurrently.

In addition to implementing this algorithm in the TWEJava run-time task scheduler, I have also provided a formal definition of the algorithm with pseudocode for all its key operations. Based

on this definition of the algorithm, I have proved that it enforces the task isolation property which is key to TWE's safety guarantees.

Chapter 5 describes this scheduling algorithm and its implementation.

1.3.4 Evaluation of performance and expressiveness

I have ported or written several programs in TWEJava, and used them to evaluate the expressiveness and performance of the TWEJava language and implementation. The evaluation shows that TWEJava can be used to write a variety of concurrent and parallel programs, including interactive applications that combine unstructured concurrency with structured parallelism.

My evaluation also shows that programs written in TWEJava can achieve good parallel speedups. The initial TWEJava scheduler could give significant speedups for some programs with relatively coarse-grain parallelism, but was limited in the speedups it could give for programs using finer-grain parallelism. My newer tree-based scheduler, on the other hand, can give good speedups for such programs. It gives significant speedups on all the programs tested, often comparable to versions of the programs with low-level synchronization and parallelism constructs that provide no safety guarantees.

Chapter 6 presents these evaluations, including the evaluation of expressiveness and performance evaluations with both the old and new schedulers.

1.3.5 Support for tasks with dynamic effects

I have extended the TWE model and TWEJava to support tasks that can determine some of their effects only dynamically as they are running. To do this, I define a feature that allows tasks to add to their effects (with certain restrictions) while they are executing.

I have extended the formal dynamic semantics of the TWE model to support this sort of dynamic addition of effects. In addition, I have added support for it in the static and run-time implementation of the TWEJava language. This has involved extending the TWEJava run-time system to support the dynamic addition of effects (restricted for efficiency to certain forms, in particular those using object references). The system needs to detect if these dynamically-added

effects conflict with the effects of other tasks in the system, and I have implemented a mechanism to do this by tracking dynamic effect sets at each node of the scheduler tree. The system can also abort and retry a task when there is a conflict. In addition to the new run-time implementation, I have described and implemented a new static data flow analysis to conservatively track what effect have been dynamically added at each point in the program, allowing the TWE system's static effect checker to be used even in the presence of dynamic effects.

I have completed an initial implementation of the dynamic effect system and evaluated it on some relatively small programs. It can offer some self-relative speedups, but it currently has significant run-time overheads. I believe those overheads can be reduced, which would make the dynamic effect system practical to use with a wide range of programs. This is a subject of my ongoing and future work.

Chapter 2

Background: Effects, Regions, and the DPJ Effect System

The Tasks With Effects model relies on effects for its basic operations, both during compile-time effect checking and during run time with effect-based task scheduling. For this reason, before presenting the TWE model in detail, I will first define and describe what effects are, as well as the natural representation of memory effects in terms of regions. Since my implementation of the TWE model in TWEJava adopts the hierarchical, region-based effect system from DPJ, I also provide an overview of that effect system, which should be sufficient to understand this work. For a full formalization of this effect system, see Robert Bocchino's thesis [24].

2.1 Effects

The TWE model relies on being able to define the *effect* of each operation in a program. This is meant to capture whatever side effects the operation may have. In practice, we focus on accesses to shared memory, although accesses to other kinds of shared data or even other types of side effects could potentially be captured.

Two effects A and B are *non-interfering* (denoted $A \# B$) if any two operations with those two effects could proceed concurrently, and any ordering (including any possible interleaving of sub-operations) would produce the same results. In the case of accesses to shared memory, we can generally say that two accesses are non-interfering if they are to different memory locations or are both reads.

An effect A is *included* in another effect B (denoted $A \subseteq B$) if $B \# C \implies A \# C$ for all possible effects C. Equivalently, we also sometimes say that B *covers* A. We may think of this as meaning that B can be used to conservatively summarize the effect A, perhaps together with

other effects that are also covered by B. This is useful because it enables us to use a single effect specification for a piece of code such as a method or task, so long as it covers all the effects of the constituent individual operations.

The TWE model could potentially be used with various effect systems; the fundamental requirement is simply that the effect non-interference and inclusion relations can be defined and checked. To use our combination of static and dynamic checks to ensure strong safety properties, it must be possible to check these properties both at compile time and run time (although TWE's effect-based scheduling mechanism could potentially be used with run-time-only effect checks). It is acceptable if the checks for these properties may be conservative in some cases (i.e. saying one effect may interfere with another even if they do not according to the above definition, or saying one effect may not be included in another when in fact it is); this will reduce the set of programs allowed or the degree of parallelism achievable at run time, but will not compromise our safety properties.

2.2 Region-Based Effects

Although various representations of effects are possible, it is natural to represent effects on shared memory in terms of reads or writes to memory regions. A *region* is simply a set of memory locations.

Given two read or write effects on regions, we may say they are non-interfering if both are reads or if the two regions (sets of memory locations) are disjoint.

We may also specify the effect inclusion relation for such effects in terms of the set-wise inclusion of regions. That is, given two regions R and S such that $R \subseteq S$, we may conclude that $\texttt{reads}\, R \subseteq \texttt{reads}\, S$ and $\texttt{writes}\, R \subseteq \texttt{writes}\, S$. Given the above definition of non-interference, it is easy to see that we may also say $\texttt{reads}\, R \subseteq \texttt{writes}\, R$, and by transitivity $\texttt{reads}\, R \subseteq \texttt{writes}\, S$.

It is also straightforward to extend region-based effects to consist of a set of reads and writes on various regions. We may say that two such effects A and B are non-interfering if all the pairs of individual read and write operations are non-interfering according to the definition above. Similarly, we may say that A is included in B if every individual read or write effect in A is included in one of the individual read or write effects in B. (This definition is conservative in that it excludes cases where an individual effect in A could be covered only be the combination of multiple effects in B,

but it is straightforward to check and sufficient for most purposes in practical programs.)

Numerous region-based effect systems have been designed, both for parallelism and for other goals such as region-based memory management. FX [63] is a seminal work that defined a region-based effect system for concurrency. Greenhouse and Boyland [41] defined a region-based effect system for Java. Some more recent works using effect systems for parallelism, including Deterministic Parallel Java (DPJ) [22] and Legion [14], have more expressive effect systems, including features like hierarchical regions, region wildcards or aliased subregions, and support for distinguishing the regions of different array elements.

The ability to express the data partitionings needed to perform various different parallel computations is a key factor in selecting a region-based effect system. Another major factor is how burdensome it is to write the necessary region and effect annotations in a program. In the TWEJava language, we chose to adopt the hierarchical, region-based effect system originally designed for DPJ. It is one of the most highly expressive effect systems we are aware of, and while there is a burden associated with writing the annotations for it, there is work showing that many of them can be inferred automatically [84, 83].

2.3 The DPJ Region and Effect System

In TWEJava, we use the effect system originally developed for the Deterministic Parallel Java (DPJ) language [22, 24]. 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. These features are described in more detail in the below section. They combine to provide a highly expressive region system, which can express the data partitioning needed to safely enable parallelism in a wide range of programs.

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 expressed as RPLs. 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.

2.3.1 Hierarchical Regions and Effects

In the DPJ effect system, a *region* is a (not necessarily contiguous) set of memory locations described by a *Region Path List* (RPL), which is a colon-separated list of RPL *elements*. An RPL element may be a simple name (a fixed string), a class region parameter, a method region parameter, an array index expression (`[expr]`), or the predefined region name, `Root`. `Root` is implicitly prepended to RPLs if it is not explicitly specified. The programmer may use RPLs to specify what regions object fields and array elements are in, and also to provide effect specifications for tasks and methods in terms of reads and writes on RPLs.

RPLs can be both static (in the program text) and dynamic (used at run time). In DPJ, RPLs are used only statically (by the compiler), although the DPJ formalism also describes their relation to dynamic RPLs. In TWEJava, dynamic RPLs are actually computed and used at run

time; the scheduler sees only dynamic RPLs. Static RPLs may contain region parameters that are substituted with a region argument provided at object allocation time, so region parameters do not appear in dynamic RPLs. Similarly, index expression elements `[`*expr*`]` become run-time integers, `[`*i*`]`, where *expr* evaluates to *i*.

RPLs may contain the *wildcard* elements `*` or `[?]`. These are called *partially specified RPLs*, and correspond to a set of *fully specified RPLs* (those without wildcard elements). An RPL containing `*` denotes the set of all legal RPLs produced by replacing the `*` with an arbitrary sequence of zero or more RPL elements. For example, `P:*` denotes the set of all RPLs beginning with the element P. An RPL containing `[?]` denotes the set of all RPLs where the '`?`' is replaced by an integer. We define a *wildcard-free prefix* of an RPL as a prefix of RPL elements that does not include `*` or `[?]`. A *maximal wildcard-free prefix* is a wildcard-free prefix that is of maximum length.

Two effects *conflict* if at least one is a write effect and their regions (RPLs) overlap, i.e., are not disjoint. Two fully specified dynamic RPLs are *disjoint* unless they are identical. Two RPLs that may contain wildcards are disjoint if every fully specified RPL represented by the first is disjoint from every such RPL represented by the second. For example, the following pairs are all disjoint: `A` and `A:B`; `A:[i]` and `A:B`; `A:*:X` and `A:B`. The following are not disjoint: `A:*` and `A`; `A:*` and `A:B:C`; `A:*` and `A:[i]`.

In practice, our system checks if two RPLs are disjoint by comparing them element-by-element first from the left until a `*` element is encountered in either RPL, and then (if necessary) from the right, stopping if they are found to be disjoint. Note that effects on two RPLs can possibly conflict only if the maximal wildcard-free prefix of one RPL is also a prefix of the other RPL.

DPJ and TWEJava require programmers to provide effect specifications for tasks and methods that may be run concurrently with other operations, as well as region specifications for the data that they access. This imposes a burden on programmers, but in our experience this burden is manageable, and we believe it can be justified in light of the strong safety guarantees that TWEJava provides. Moreover, Vakilian et al. [84] have shown that it is possible for a tool to automatically infer most DPJ/TWEJava-style effect specifications, dramatically reducing the manual annotation burden on programmers. Tzannes et al. [83] have also demonstrated that for some programs it is

feasible to infer most or all region annotations, as well as effect annotations.

Chapter 3

The Tasks with Effects Model and TWEJava

3.1 The TWEJava language

As described in section 1.2, the core idea of the tasks with effects programming model is to associate an *effect specification* with each task in a program, and then to schedule tasks at run time on the basis of these effect specifications, ensuring that tasks with conflicting effect specifications do not run concurrently. This system of run-time scheduling to ensure task isolation can be coupled with static checks that ensure each task's effect specification conservatively covers all the potential memory side effects of the task. By doing this, a system based on tasks on effects can give the programmer strong safety guarantees, including a guarantee of data race freedom and a guarantee that portions of tasks that do not create or wait for other tasks will behave atomically.

I have implemented such a system in an extended version of Java, which I call TWEJava. In this section, I will introduce the key features of the TWEJava language, and informally describe the concepts of the TWE model that they implement. (In section 3.2, I will give a more formal treatment of the dynamic semantics of tasks with effects.) TWEJava includes almost all Java language features, but TWEJava programs should not use Java's thread-based concurrency mechanisms or lock-based synchronization, which TWEJava is designed to replace. Figure 3.1 shows the new operations supported by TWEJava, which I will describe in this section.

Figure 3.2 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 6.1). 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 3.3 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.)

3.1.1 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.

```
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 3.1: 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.

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 3.1.5.)

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

```
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 3.2: Example computation.

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.

3.1.2 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 have interfering effects 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.

To specify effects in TWEJava, we adopt the region and effect system from Deterministic

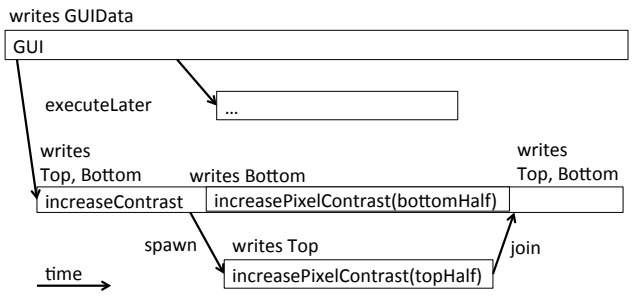


Figure 3.3: Tasks in example computation.

Parallel Java (DPJ) [22], which is described in Section 2.3. In our application of the effect system to TWEJava, every task as well as every method has effects specified by the programmer in terms of reads and writes to regions.

We also use an extension of the basic DPJ type system to support effect parameters to types (in addition to region parameters). This is based on the support for effect parameters described by Bocchino et al. [21], but it is modified to follow a more restrictive subtyping rule. Specifically, we 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 rule is helpful for supporting the `join` operation described later.)

The use of effect parameters allows us to define the abstract class `Task` as taking an effect parameter `E`. This class 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.)

3.1.3 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 3.3, 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 3.1.5.)

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 3.4.2. 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 [20], 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.

3.1.4 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.

3.1.5 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 3.2, 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.

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. We determine what effects are statically considered to be transferred on join by looking at the effect parameter of the `SpawnedTaskFuture` object being joined. In our static analysis, we treat effects as being transferred on join only if that effect parameter is fully-specified, which under

our typing rules is sufficient to ensure that it will match the actual effect of the task being joined. Dynamically, we always consider the effects of a completed child task to be transferred when it is joined. Accordingly, the static analysis may be conservative in the sense of not allowing the parent task to perform certain operations in cases where the effects are not treated as being transferred, but it is safe in the sense that any operation dynamically performed by a task is guaranteed to be covered by the dynamic effect of the task at that point.

The restriction of not treating effects as being transferred on join when a non-fully-specified effect parameter is used in the type of the `SpawnedTaskFuture` object does slightly reduce expressiveness, since in these cases shared data written by the child task cannot subsequently be accessed by the parent. However, we have not found this to a problem in programs that we have written. Moreover, this restriction could be worked around in various ways, e.g. by having the child return a new object containing the results in `final` fields, which would be accessible to the parent. It would be possible to avoid this restriction by adding new type system mechanisms, annotations, or static analyses that could be used to ensure that the effect parameter of a `SpawnedTaskFuture` object's type matches the actual effect of the task being joined; we have not done this to date in the interest of simplicity and because we have not encountered cases where it would be necessary.

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. (We can treat a method as not "giving up" any effects even in the cases discussed above where our conservative static analysis within the method does not treat effects as being transferred on join, since we know that dynamically all the effects of tasks `spawn`ed within that method will have been transferred back to it by join operations.)

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. (Intuitively, subtracting an effect means that operations whose effects conflict with that effect will no longer be permitted below that point in the program, until and unless that effect is subsequently added back.) 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 (so an operation is permitted only if its effect is covered by both of the covering effects from the different control flow paths). 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 the current covering effect at the program point before the expression, to ensure the expression's effects will be covered. In Chapter 4, I formalize this dataflow analysis and provide formal definitions of the structures used to represent covering effects (known as *compound effects*).

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 statically disallow `spawn` operations only in certain cases where we can determine by static analysis that the effects of the spawned task *definitely will not* be covered by the covering effect at run time. We allow other `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. (This differs from the way we handle the effects of all other operations in the program, where an error is reported unless our conservative static analysis can determine that their effects *definitely will* be covered.) In this case of a `spawn` where we cannot determine statically that its effects definitely will or will not be covered, 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). As mentioned above, 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.

3.2 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 3.4.

These rules are written using the K semantic framework [78], 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*

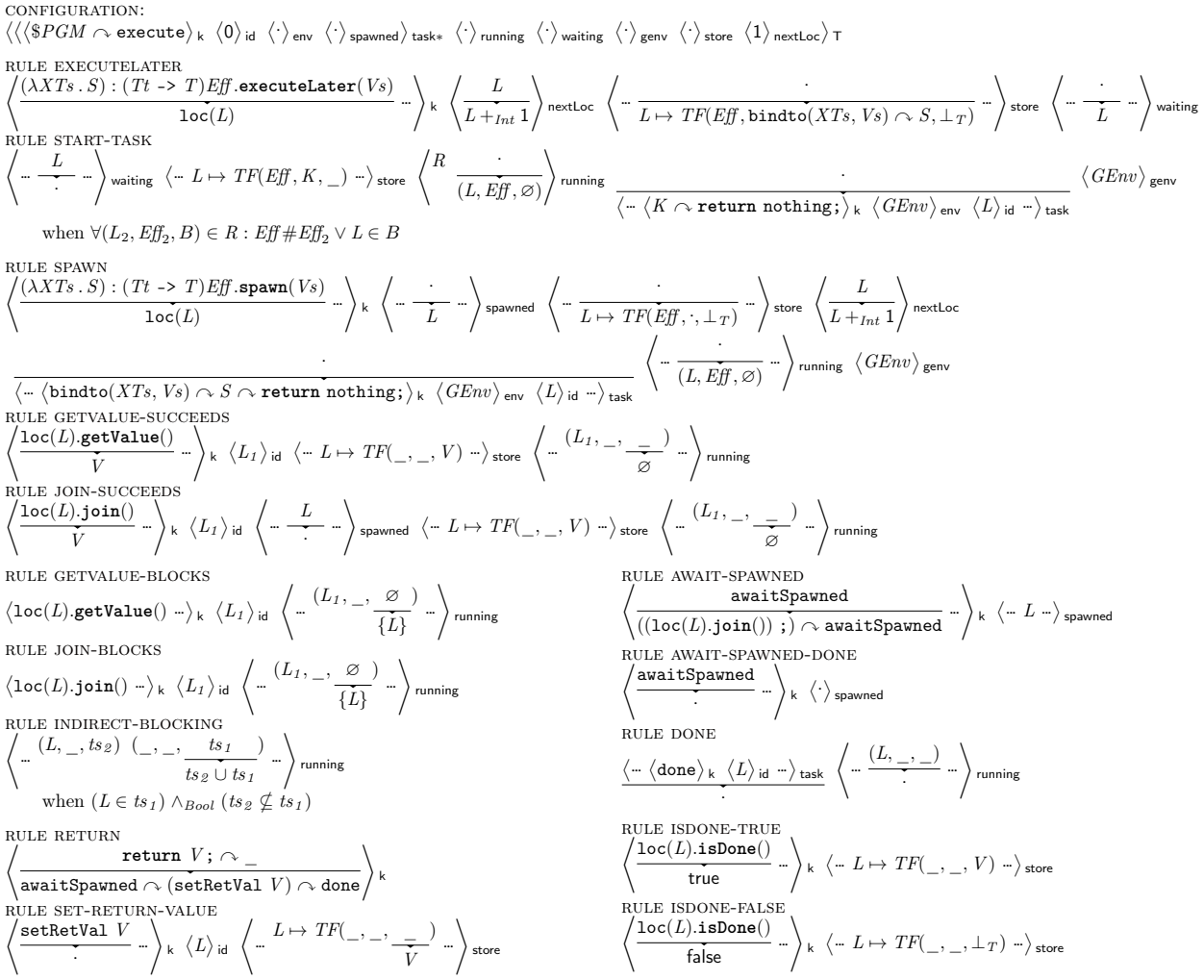


Figure 3.4: Dynamic semantics of tasks with effects.

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 [22].

3.2.1 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.

3.2.2 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.)

3.2.3 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.

3.3 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.

3.3.1 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.

3.3.2 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 3.2 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.

3.3.3 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.

3.3.4 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 3.1.4. 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 6.1).

3.3.5 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 [22] 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 3.1.5. 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 3.1.4 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.

3.4 Compiler and Runtime System

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

3.4.1 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 3.1. 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.

Our compiler is responsible for generating the code that at run time will compute dynamic RPLs based on the static RPLs appearing in the program text (specifically, dynamic RPLs must be computed for those RPLs that may be used in the effect of a task, either directly or through effect parameter instantiations). For RPLs consisting of simple region name elements, this is straightforward, but we must also deal with other features including region parameters and index expressions in RPLs. Our compiler generates code to actually pass region parameters at run time (as extra parameters to constructors or methods in the generated Java code), as well as to evaluate index expressions (which are restricted to simple side-effect-free expressions). This allows us to compute the dynamic RPLs appearing in the effects of tasks at run time; these are the RPLs that will actually be used within our run-time scheduler. This computation of dynamic RPLs is a new feature not present in DPJ; although the DPJ formalism defines dynamic RPLs, they are not actually computed in DPJ programs.

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.

3.4.2 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. 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.

Here I describe the initial naïve prototype implementation of the scheduler, which uses a queue of tasks protected by a single lock to manage the effect-checking phase of task scheduling. This implementation has since been replaced by the much more scalable tree-based implementation discussed in chapter 5, but I briefly describe it here so that the reader can understand the comparative performance characteristics of the two implementations.

When attempting to execute a task, the naïve scheduler 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.

Chapter 4

Covering Effect Analysis

Our compiler needs to statically ensure that the effect of any operation in the program is covered by the static covering effect at that point in the program. As discussed informally in section 3.1.5, this must take into account that the covering effect may be changed by `spawn` and `join` operations, and we use a data flow analysis to do that. Our data flow analysis computes the current covering effect at each point in the program. Our compiler then checks whether the effect of each individual operation is included in the current covering effect at that point. If it is not, our compiler reports an error.

4.1 Compound effects

We represent the current covering effect at each point in a method using a *compound effect*. A compound effect is a set of effects, although it is often convenient to represent it using an alternate notation that corresponds more directly to the operations that are used in our covering effect analysis. Specifically, the compound effects that appear in our analysis can be represented by the following simple grammar (where E is an effect):

$\mathcal{E} ::= E \mid \mathcal{E} + E \mid \mathcal{E} - E \mid \mathcal{E} \cap \mathcal{E}$

Each of these forms is shorthand for a certain set of effects. Apart from the standard set intersection operation $\cap$, they are defined as follows:

1. $E \equiv \{E' \in \mathcal{D} : E' \subseteq E\}$
2. $\mathcal{E} + E \equiv \{E' \in \mathcal{D} : E' \in \mathcal{E} \vee E' \subseteq E\}$
3. $\mathcal{E} - E \equiv \{E' \in \mathcal{D} : E' \in \mathcal{E} \wedge E' \# E\}$

In these definitions, $\subseteq$ is the effect inclusion relation and $\#$ is the noninterference relation for

our effect system. Note that definition 1 gives the meaning of the *compound effect* E (which is a set, and may contain multiple effects) based on the effect inclusion relation for the single *effect* E; it is not a circular definition.

We use a region-based effect system adapted from DPJ, which is described informally and explained in Chapter 2. The DPJ effect system is formalized in [24]. The concept of compound effects and the covering effect analysis described in this chapter could potentially also be applied to other effect systems that define inclusion and noninterference relations.

In these definitions, $\mathcal{D}$ is some set of effects. In the below formalism, we assume that the same choice of $\mathcal{D}$ is adopted for all the compound effects under discussion, and do not show it explicitly. In principle, $\mathcal{D}$ can be any set of effects (including the set of all effects). In practice, in our covering effects data flow analysis $\mathcal{D}$ will be a set of effects that are relevant for the analysis of a particular flow graph; we will discuss the choice of $\mathcal{D}$ for our data flow analysis in detail below.

We call a compound effect of the form $\mathcal{E} + E$ an *additive compound effect* and one of the form $\mathcal{E} - E$ a *subtractive compound effect*. It is sometimes useful to consider in particular compound effects where several operations of the form $+E$ or $-E$ may occur in sequence, e.g. $E_1 + E_2 - E_3 - E_4 + E_5$. We refer to a sequence of zero or more successive operations of the form $+E$ or $-E$ (e.g. $+E_2 - E_3 - E_4 + E_5$ in the example given) as an *additive-subtractive sequence*. Note that the $+$ and $-$ operators are left-associative.

4.1.1 Checking effect inclusion in additive and subtractive compound effects

Note that based on the above definitions, we can evaluate whether an effect E' is in $\mathcal{E} + E$ using a sequential procedure:

1. If $E' \subseteq E$, then stop. We know that $E' \in \mathcal{E} + E$.
2. Otherwise, the result is the same as for checking if $E' \in \mathcal{E}$.

Similarly, we can check whether $E' \in \mathcal{E} - E$ as follows:

1. If $\neg E' \# E$, then stop. We know that $E' \notin \mathcal{E} - E$.
2. Otherwise, the result is the same as for checking if $E' \in \mathcal{E}$.

```
1  def effectIn(E, 𝓔 t):
2      for each operation o in t, from right to left do
3          if o is of the form +E′ && E ⊆ E′ then
4              return true
5          if o is of the form −E′ && ¬E#E′ then
6              return false
7      if E ∈ 𝓔 then
8          return true
9      else
10         return false
```

Figure 4.1: Procedure to determine if an effect E is in a compound effect $\mathcal{E}\,t$, where t is an additive-subtractive sequence

In either case, it is only necessary to proceed to evaluating whether $E' \in \mathcal{E}$ if the condition checked in step 1 of the respective procedures (based on E) is not true.

Based on these observations, we can define a straightforward iterative procedure for evaluating whether an effect E is in a compound effect of the form $\mathcal{E}\,t$, where t is an additive-subtractive sequence. This procedure is shown in Figure 4.1. It will be useful when analyzing our data flow analysis framework (described below), because the transfer functions in that framework yield compound effects of this form.

4.1.2 Semilattice for compound effects

We can define a semilattice over the domain of compound effects, with $\cap$ as the meet operator. The corresponding partial order is that given by $\subseteq$. The top element $\top$ of the semilattice is a compound effect consisting of the effect that covers all possible effects. For our effect system, this is `writes *`. The bottom element $\bot$ is the compound effect consisting of the effect `pure`, which covers no read or write operations and is a subeffect of every possible effect.

4.2 Data flow analysis framework for covering effects

We now define a data flow analysis framework which can be used to compute the covering effect at each point in a procedure. It is a forward analysis over the semilattice of compound effects with the meet operator $\cap$, as defined above.

The family F of transfer functions include the following, where E may be any effect:

- $f_{id} : \mathcal{E} \to \mathcal{E}$
- $f_E : \mathcal{E} \to E$
- $f_{+E} : \mathcal{E} \to \mathcal{E} + E$
- $f_{-E} : \mathcal{E} \to \mathcal{E} - E$
- the composition $f \circ g$, for any two functions $f, g \in F$.

The function f_{-E} can be used when a subtask with static effect E is `spawn`ed, reflecting that the effect E is being transferred to the spawned subtask, and is therefore subtracted from the covering effect in the parent task. Conversely, the function f_{+E} is used when a `join` operation appears; it transfers the effect E of a completed subtask back to the parent task. (As mentioned in Section 3.1.5, the effect to transfer is determined by the effect parameter in the static type of the `SpawnedTaskFuture` object being joined; effect transfer is performed only if that effect parameter is fully-specified.) The identity function f_{id} is used for blocks that do not contain operations that would change the current covering effect, and the constant function f_E is used to initialize the current covering effect to the declared effect of a task or method at the entry node to that task or method's control flow graph.

4.2.1 Properties of the data flow analysis framework

We will now show that the data flow analysis framework for covering effect analysis is monotone, distributive, and rapid.

Lemma 1. *Every function $f \in F$ is of the form (i) $\mathcal{E} \to \mathcal{E}\, t$ or (ii) $\mathcal{E} \to E\, t$, where t is an additive-subtractive sequence.*

Proof. This can be seen by straightforward induction. The property obviously holds for the base cases f_{id}, f_E, f_{+E}, and f_{-E}.

For the composition $f \circ g$, assume the property holds for f and g. If f is of form (ii) then it is a constant function independent of its input, so we simply have $f \circ g = f$ and property holds by assumption. If f is of form (i) then we have f of the form $\mathcal{E} \to \mathcal{E}\, t$ and g of the form $\mathcal{E} \to \mathcal{E}\, t'$ or

$\mathcal{E} \to E\,t'$. So $f \circ g$ is of the form $\mathcal{E} \to \mathcal{E}\,t'\,t$ (if g is of form (i)) or $\mathcal{E} \to E\,t'\,t$ (if g is of form (ii)). In either case, it is clear that concatenation of sequences t' and t is itself a sequence of the desired form, so the property holds. □

We now prove a key lemma that helps us to reason about the relationship between multiple compound effects of the form $\mathcal{E}\,t$, where $\mathcal{E}$ may differ but an additive-subtractive sequence t is the same in all the effects. Informally, we may see it as showing that effects fall into two categories:

1. There are some effects E where deciding whether E is in $\mathcal{E}\,t$ depends only on t (and so gives the same result for all $\mathcal{E}_n\,t$ sharing a common t).
2. For all other effects E, $E \in \mathcal{E}\,t$ iff $E \in \mathcal{E}$.

We will use this lemma to help prove that our framework is distributive and rapid. (Note that $\wedge$ in the below lemma and the remainder of our formalism is the logical and operation, and $\vee$ is logical or; the meet operator in our framework is $\cap$ and is always shown as such.)

Lemma 2. *Given any effect E and any compound effects $\mathcal{E}_1\,t, ..., \mathcal{E}_n\,t$, where t is an additive-subtractive sequence that is the same in all the compound effects, at least one of the following is true:*

1. $\bigwedge_{i=1}^{n} E \in \mathcal{E}_i\,t$
2. $\bigwedge_{i=1}^{n} E \notin \mathcal{E}_i\,t$
3. $\bigwedge_{i=1}^{n}((E \in \mathcal{E}_i\,t \wedge E \in \mathcal{E}_i) \vee (E \notin \mathcal{E}_i\,t \wedge E \notin \mathcal{E}_i))$

Proof. To see this, first observe that we can determine whether an effect E is in $\mathcal{E}_i\,t$ using the sequential procedure shown in Figure 4.1. Note that the operations in the initial **for** loop (lines 2–6) involve only E and t, not $\mathcal{E}_i$. Accordingly, whether the procedure will return a value from within that loop (i.e. from line 4 or line 6), and if so what value, depends only on E and t. So, for any effect E, either *effectIn*$(E, \mathcal{E}_i\,t)$ will return the same value from within the loop for all values of i, or it will not return a value from inside the loop for any value of i. The former case means that either $E \in \mathcal{E}_i\,t$ for all values of i or $E \notin \mathcal{E}_i\,t$ for all values of i, so condition 1 or condition 2 will hold.

In the latter case, the procedure will proceed to check whether E is in $\mathcal{E}_i$. If it is, the procedure will return from line 8, indicating that E is also in $\mathcal{E}_i\, t$; if not, it will return from line 10, indicating the reverse. So we will have either $E \in \mathcal{E}_i\, t$ and $E \in \mathcal{E}_i$, or $E \notin \mathcal{E}_i\, t$ and $E \notin \mathcal{E}_i$. This result, as applied to $\mathcal{E}_1\, t, ..., \mathcal{E}_n\, t$ (since, as mentioned, the procedure will proceed to this latter phase for all $\mathcal{E}_i\, t$ if it does for any of them), is condition 3. So we may conclude that at least one of the three conditions will apply in every case. □

Theorem 1 (Distributivity)**.** *For any function $f \in F$ and any two compound effects $\mathcal{E}_1$ and $\mathcal{E}_2$, $f(\mathcal{E}_1 \cap \mathcal{E}_2) = f(\mathcal{E}_1) \cap f(\mathcal{E}_2)$.*

Proof. Lemma 1 gives the two possible forms of a function $f \in F$. If f is of form (ii) ($\mathcal{E} \to E\, t$, where t is an additive-subtractive sequence) then the desired property follows immediately because f is a constant function, independent of its inputs. So the remaining case to be considered below is if f is of form (i) ($\mathcal{E} \to \mathcal{E}\, t$, where t is an additive-subtractive sequence).

Since $f(\mathcal{E}) = \mathcal{E}\, t$, the equality we wish to prove is $(\mathcal{E}_1 \cap \mathcal{E}_2)\, t = \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$. By Lemma 2, we know that for any effect E, at least one of the following holds (we break the third case from Lemma 2 into two possibilities, depending on which clause of its disjunction applies for $(\mathcal{E}_1 \cap \mathcal{E}_2)\, t$):

1. $E \in (\mathcal{E}_1 \cap \mathcal{E}_2)\, t \wedge E \in \mathcal{E}_1\, t \wedge E \in \mathcal{E}_2\, t$
2. $E \notin (\mathcal{E}_1 \cap \mathcal{E}_2)\, t \wedge E \notin \mathcal{E}_1\, t \wedge E \notin \mathcal{E}_2\, t$
3. $E \in (\mathcal{E}_1 \cap \mathcal{E}_2)\, t \wedge E \in \mathcal{E}_1 \cap \mathcal{E}_2 \wedge$
   $((E \in \mathcal{E}_1\, t \wedge E \in \mathcal{E}_1) \vee (E \notin \mathcal{E}_1\, t \wedge E \notin \mathcal{E}_1)) \wedge$
   $((E \in \mathcal{E}_2\, t \wedge E \in \mathcal{E}_2) \vee (E \notin \mathcal{E}_2\, t \wedge E \notin \mathcal{E}_2))$
4. $E \notin (\mathcal{E}_1 \cap \mathcal{E}_2)\, t \wedge E \notin \mathcal{E}_1 \cap \mathcal{E}_2 \wedge$
   $((E \in \mathcal{E}_1\, t \wedge E \in \mathcal{E}_1) \vee (E \notin \mathcal{E}_1\, t \wedge E \notin \mathcal{E}_1)) \wedge$
   $((E \in \mathcal{E}_2\, t \wedge E \in \mathcal{E}_2) \vee (E \notin \mathcal{E}_2\, t \wedge E \notin \mathcal{E}_2))$

In case 1, $E \in \mathcal{E}_1\, t \wedge E \in \mathcal{E}_2\, t \implies E \in \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$, so we have both $E \in (\mathcal{E}_1 \cap \mathcal{E}_2)\, t$ and $E \in \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$. Similarly, in case 2, $E \notin \mathcal{E}_1\, t \wedge E \notin \mathcal{E}_2\, t \implies E \notin \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$, so we have both $E \notin (\mathcal{E}_1 \cap \mathcal{E}_2)\, t$ and $E \notin \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$.

In case 3, we know $E \in \mathcal{E}_1 \cap \mathcal{E}_2$, so $E \in \mathcal{E}_1$ and $E \in \mathcal{E}_2$. Based on the latter two clauses of case

3, we can therefore conclude that $E \in \mathcal{E}_1\, t$ and $E \in \mathcal{E}_2\, t$. Therefore, we will have $E \in \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$ and also (immediately from case 3) $E \in (\mathcal{E}_1 \cap \mathcal{E}_2)\, t$.

In case 4, we know $E \notin \mathcal{E}_1 \cap \mathcal{E}_2$, which implies $E \notin \mathcal{E}_1 \vee E \notin \mathcal{E}_2$. Based on the latter two clauses of case 4, this in turn implies $E \notin \mathcal{E}_1\, t \vee E \notin \mathcal{E}_2\, t$, and therefore $E \notin \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$. We also have $E \notin (\mathcal{E}_1 \cap \mathcal{E}_2)\, t$ immediately from case 4.

So in each possible case we have either $E \in (\mathcal{E}_1 \cap \mathcal{E}_2)\, t$ and also $E \in \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$, or else $E \notin (\mathcal{E}_1 \cap \mathcal{E}_2)\, t$ and also $E \notin \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$. Since we know that at least one of these four cases will apply for every possible effect E, this is sufficient to conclude that $(\mathcal{E}_1 \cap \mathcal{E}_2)\, t = \mathcal{E}_1\, t \cap \mathcal{E}_2\, t$. This concludes the proof that $f(\mathcal{E}_1 \cap \mathcal{E}_2) = f(\mathcal{E}_1) \cap f(\mathcal{E}_2)$ for all $f \in F$, i.e. all functions $f \in F$ are distributive. □

Theorem 1 proves that the data flow analysis framework we have defined is a *distributive framework*. From this distributivity property the weaker corollary of *monotonicity* also immediately follows:

Corollary 1 (Monotonicity)**.** *For any function $f \in F$ and any two compound effects $\mathcal{E}_1$ and $\mathcal{E}_2$, $f(\mathcal{E}_1 \cap \mathcal{E}_2) \subseteq f(\mathcal{E}_1) \cap f(\mathcal{E}_2)$, or equivalently $\mathcal{E}_1 \subseteq \mathcal{E}_2 \iff f(\mathcal{E}_1) \subseteq f(\mathcal{E}_2)$.*

Theorem 2 (Rapidity)**.** *For any $f \in F$ and any compound effect $\mathcal{E}$, $f(\mathcal{E}) \supseteq \mathcal{E} \cap f(\top)$.*

Proof. If f is of form (ii) ($\mathcal{E} \to E\, t$, where t is an additive-subtractive sequence), it is a constant function independent of its input, and therefore the desired property follows easily. So the remaining case is if f is a function of form (i) ($\mathcal{E} \to \mathcal{E}\, t$, where t is an additive-subtractive sequence).

With $f(\mathcal{E}) = \mathcal{E}\, t$, the desired property becomes $\mathcal{E}\, t \supseteq \mathcal{E} \cap \top\, t$ (where $\top$ is the compound effect `writes *`, as noted in Section 4.1.2). Applying Lemma 2 to $\mathcal{E}\, t$ and $\top\, t$, we know at least one of the following must hold for any effect E:

1. $E \in \mathcal{E}\, t \wedge E \in \top\, t$
2. $E \notin \mathcal{E}\, t \wedge E \notin \top\, t$
3. $((E \in \mathcal{E}\, t \wedge E \in \mathcal{E}) \vee (E \notin \mathcal{E}\, t \wedge E \notin \mathcal{E})) \wedge ((E \in \top\, t \wedge E \in \top) \vee (E \notin \top\, t \wedge E \notin \top))$

Moreover, we may simplify case 3 as follows, since we know $E \in \top$:

$$((E \in \mathcal{E}\, t \wedge E \in \mathcal{E}) \vee (E \notin \mathcal{E}\, t \wedge E \notin \mathcal{E})) \wedge E \in \top\, t \wedge E \in \top$$

This in turn can be split into two possible cases as follows:

3(a). $E \in \mathcal{E}\,t \wedge E \in \mathcal{E} \wedge E \in \top\,t \wedge E \in \top$

3(b). $E \notin \mathcal{E}\,t \wedge E \notin \mathcal{E} \wedge E \in \top\,t \wedge E \in \top$

If case 3(a) applies, then case 1 also applies. So we may conclude that one of cases 1, 2, and 3(b) (which are mutually contradictory, so only one can apply) will hold for any E.

If case 1 applies, we have $E \in \mathcal{E}\,t$.

If case 2 applies, we have $E \notin \mathcal{E}\,t$ but also $E \notin \top\,t \implies E \notin \mathcal{E} \cap \top\,t$.

If case 3(b) applies, we have $E \notin \mathcal{E}\,t$ but also $E \notin \mathcal{E} \implies E \notin \mathcal{E} \cap \top\,t$.

So the only case where E may be in $\mathcal{E} \cap \top\,t$ is case 1, in which case we also know $E \in \mathcal{E}\,t$. Therefore we may conclude that for any E, if $E \in \mathcal{E} \cap \top\,t$, then $E \in \mathcal{E}\,t$, and so $\mathcal{E}\,t \supseteq \mathcal{E} \cap \top\,t$, or equivalently $f(\mathcal{E}) \supseteq \mathcal{E} \cap f(\top)$. □

This suffices to prove that our monotone framework is *rapid* [55, 65]. This has implications for the speed of convergence of the iterative data flow algorithm, which will be discussed below.

4.3 Iterative data flow analysis for covering effects

Since we have now established that our data flow analysis framework for covering effects is distributive (and monotone), we can define an iterative algorithm to compute covering effects and discuss its properties. Given the control flow graph for a task or method in our system, we can compute the covering effect for each operation in it using the algorithm in Figure 4.2. (This is a straightforward adaptation of the general algorithm for an iterative data flow analysis, as given by Aho et al. [7]. Note that we assume the flow graph has an empty entry block called ENTRY.)

To use this algorithm, we should restrict the domain $\mathcal{D}$ of effects to be considered to the effects of individual operations actually appearing in the flow graph being analyzed. These include basic read and write effects from accesses to shared memory locations, as well as the statically declared effects of methods that are called. This approach restricts $\mathcal{D}$ to a finite set, while enabling all of the effects actually of interest for effect checking to be analyzed. By restricting $\mathcal{D}$ to a finite set, we ensure that the semilattice over compound effects has a finite size and thus a finite depth, which

```
1 OUT[ENTRY] = declared effects of task or method
2 for each basic block B other than ENTRY do
3 |  OUT[B] = writes *
4 while any OUT is changed do
5 |  for each basic block B other than ENTRY do
6 |  |  IN[B] = ⋂_{P a predecessor of B} OUT[P]
7 |  |  OUT[B] = f_B(IN[B])
```

Figure 4.2: Iterative algorithm for computing covering effects

(along with our framework's monotonicity) serves to ensure the algorithm will converge. Moreover, this makes it possible to represent compound effects with data structures like bit vectors, so it becomes straightforward to determine when one has been changed. The algorithm in Figure 4.1 can be used to incrementally compute the changes in compound effects when a function $f_B \in F$ is applied.

Because our data flow framework is monotone, we know that the solution found will be the maximum fixedpoint (MFP) for our data flow equations [56]. Moreover, because our framework is distributive, this will in fact be equal to the meet-over-paths solution [59], i.e. for each block B other than ENTRY, the final computed value of IN[B] is equal to $\bigcap_{\textit{all paths P from ENTRY to B}} f_P(v_{ENTRY})$, where v_{ENTRY} is the initial value of the covering effect at the entry block (set equal to the declared effects of the task or method) and f_P is the full transfer function for the path P (a composition of the transfer functions of the individual blocks on the path). This is a safe solution, in the sense that it will necessarily be equal to or a subset of the compound effect that could be computed dynamically based on whatever path is actually taken in an execution of the program.

As mentioned above, our framework is rapid (and has a $\top$ element), which gives a bound on how quickly the iterative algorithm will converge if it processes blocks in depth-first order (reverse postorder) [55, 65]. Specifically, the algorithm will halt after at most $d + 2$ iterations, where d is the depth of the flow graph. In fact, the final values of the covering effects will have been computed after at most $d + 1$ iterations, since the final iteration just serves to confirm they have converged. (The depth d could in general depend on the depth-first spanning tree chosen for the CFG, but in fact all CFGs for our language are reducible, so the depth of a given CFG is a single value, which is no greater than the loop nesting depth that could be computed based on the code structure.)

4.4 Structure-based data flow analysis algorithm

The iterative algorithm above could be used to compute the covering effects, and we believe it would perform reasonably, but in our actual implementation we use a different, structure-based algorithm. We chose to use this algorithm because the `javac` codebase on which our compiler is based provided an existing infrastructure for structure-based data flow analyses that we could leverage to more easily implement our new covering effects analysis. Note that since we are using a Java-based language, our programs can include only structured code, so the CFG is always reducible, and loops within it can arise only from explicit looping statements in the code.

Our analysis essentially proceeds over the structure of the AST in program order, maintaining the data flow set (in this case, the covering effect) and updating it as appropriate based on each operation encountered. Upon encountering control flow branching constructs other than loops (e.g. `if` or `switch` statements), the analysis computes covering effects for each side of the branch, and at the subsequent control flow merge point, it applies the meet operation ($\cap$) to merge the covering effects computed in the various branches.

When the analysis encounters a loop statement, it computes covering effects for the body of the loop in one pass, and if the covering effect at the end of the loop differs from the one at the beginning, it performs another pass to recompute the covering effects within the loop, taking the intersection of the covering effect just before the loop and the one computed at the end of the loop in the initial pass as the covering effect at the start of the loop body. (Note that this applies at each level of loop nesting, so the covering effects for a loop body nested n levels deep may be recomputed up to 2^n times in total, although this maximum will apply only when an operation that changes the covering effect appears within that loop.)

We informally argue that the traversal of any AST subtree in our analysis yields a result corresponding to the meet over all paths within the CFG subgraph corresponding to that AST subtree. This is obvious for straight-line code. It is also fairly easy to see for branching constructs other than loops. In these cases, there will be paths corresponding to each possible side of the branch, so we do the computation for each side, and then apply the meet operation ($\cap$) at the point where they merge.

The more interestesting case is of a loop. The rapidity property informally means that a single traversal of a loop will fully summarize the contributions of that loop to the data flow values being computed, and those contributions are independent of the initial value at the beginning of the loop [65]. So the initial pass over a loop will fully summarize its additions and subtractions to the covering effects. If there are any changes, then a second iteration is needed simply to recompute the covering effect at loop entry to take into account the contributions of the loop body, and then propagate those contributions through the loop (reflecting paths where the loop runs two or more iterations). But additional iterations would produce no further changes, so they are unnecessary.

Based on the above property, we can conclude that the full AST traversal for a task or method produces the the meet-over-paths solution, the same as the iterative algorithm discussed above. We chose this algorithm rather than the iterative algorithm chiefly for ease of implementation in the codebase we were using, but we also observe that it may offer performance benefits over the iterative algorithm, at least in some cases. In particular, with this algorithm, it is not necessary to compute concretely what individual effects are included in the compound effects produced during the computation. Instead, compound effects can be represented in an abstract form corresponding directly to the simple grammar given in Section 4.1, and in fact that is what our implementation does.

This approach has the advantage of avoiding the need to do numerous effect inclusion and noninterference checks in the course of computing covering effects, which may well make it faster. However, it does have the disadvantage that it is not easy in general to determine if two compound effects in the abstract form are equal. To address this, we use some simple heuristics, but in some cases we may not be able to determine that two compound effects are equal in cases where they actually are. In these cases, we conservatively treat them as potentially unequal, which may lead our algorithm to do a second iteration around a loop in cases where it is actually unnecessary. This is harmless for correctness, although obviously there is a performance cost. (Another advantage of keeping the covering effects we compute in the abstract form is that they can be printed in error messages about operations with uncovered effects. Since the additions and subtractions in the covering effect correspond to operations in the code, they are fairly easy to understand.)

In general, our algorithm may be faster than the iterative algorithm for the reasons mentioned above, but on the other hand it has the potential to perform more iterations for inner loops. As mentioned, this is in the worst case exponential in the loop nesting depth. In practice, our algorithm performs acceptably on all the codes we have used it on, and we believe it will do so in the vast majority of cases. We expect `spawn` and `join` operations would rarely be written within very deeply nested loops, so the exponential scaling would not pose a major problem in practice. We believe both the iterative algorithm and the structure-based algorithm discussed here are practical approaches to implementing covering effects analysis. It could also be fruitful to explore other possible approaches such as region-based analysis, which might combine some of the advantages of the two approaches, and potentially be faster than either.

Chapter 5

Scalable Scheduling for Tasks with Hierarchical Effects

5.1 Introduction

The Tasks With Effects (TWE) programming model described in Chapter 3 offers strong parallel safety guarantees, including data race freedom, strong isolation, and determinism where desired, while providing the expressiveness to support a wide variety of parallel and concurrent programs. TWE is more flexible than any other system we are aware of with similarly strong guarantees, because it can support nearly arbitrary unstructured as well as structured parallelism, using a combination of static and dynamic techniques to enforce its guarantees. These properties make TWE a highly attractive programming model, with the potential to program a wide range of parallel programs while avoiding the numerous forms of concurrency errors that are possible in traditional parallel programming models that give few safety guarantees.

These advantages, however, come at the cost of potentially high run-time overheads in the task scheduler. In particular, TWE achieves its safety guarantees by using a *side-effect-aware run-time task scheduler*, which schedules tasks so that no two tasks with conflicting side-effects can run concurrently (these terms are made precise in Section 5.2). In TWE (following DPJ) the effect specifications of tasks are declared in the source program, using a rich system of hierarchical effect specifications based on memory regions. As in DPJ, the declared effects of a task are guaranteed to be sound using modular static checking. The static checking and the run-time invariant of conflict-freedom enforced by the TWE scheduler together give rise to the strong determinism-by-default guarantees, and they do so in a far wider range of programs than DPJ.

To realize the benefits of this model, however, we require an efficient, scalable task scheduler that not only dispatches tasks but also checks run-time conflicts between potentially hundreds of

concurrent tasks. The initial TWEJava implementation [48] used a naïve scheduler design with a single task queue protected by a global lock, which predictably provided poor scalability in many cases.

In this chapter, I present a new, scalable algorithm for enforcing mutual exclusion between concurrent operations, and describe how we use it to develop a scalable task scheduler for TWEJava. For maximum scalability, the scheduler takes advantage of the special hierarchical structure of TWE effect specifications to eliminate or minimize the need for run-time effect checks as much as possible. The scheduler supports the full generality of TWE effect specifications, efficiently supports programs that have many fine-grain tasks, and is designed not to introduce deadlock. The scheduler uses a tree structure corresponding to the hierarchical structure of memory regions in TWE to track effects. The scheduler is designed to have two key properties that ensure scalability. First, it is necessary to compare an effect at one node only with other effects at the same node, its ancestor nodes, and (in some cases) its descendants. We know that it will not conflict with effects at other nodes, so we do not need to explicitly compare them. This feature can greatly reduce the number of effect comparisons needed when running a set of tasks with largely independent effects. Second, the algorithm is designed to minimize contention within the scheduler itself by generally locking only individual tree nodes during scheduling operations, permitting operations on other parts of the tree to proceed concurrently. The combination of these features allows the scheduler to remain scalable even for codes that use quite fine-grain tasks.

We focus here on the region and effect system of TWEJava, but we believe our scheduler design could also be adapted to work with other hierarchical region systems such as those in Legion [14] and Joe [32]. These systems all support a similar basic hierarchical structure of regions, as well as some mechanism that allows broader, potentially overlapping region specifications. Those latter mechanisms could be handled similarly to our handling of wildcards in TWEJava (described below).

This chapter makes includes following contributions:

- We define an efficient, scalable algorithm for enforcing mutual exclusion of concurrent operations based on hierarchical effects, and describe its application to scheduling TWEJava tasks. Our algorithm has two key properties that ensure scalability, even for fine-grain tasks: effect

conflicts must be checked with only ancestor and perhaps descendant nodes in the scheduler tree; and fine-grain locking of scheduler tree nodes is used to allow scheduling operations to proceed concurrently. These properties enable it to provide scalability even with unstructured concurrency in which the effects of any two tasks in the system could potentially conflict.

- We prove that our algorithm enforces the task isolation property (mutual exclusion of tasks with conflicting effects), which yields TWE's major safety guarantees.

- We describe some optimizations over our base implementation, including the use of read-write locking to avoid contention at the root node of the tree and a mechanism to skip certain unnecessary effect conflict checks.

- We evaluate our scheduler on six TWEJava programs, showing that it gives significant speedups on all of them, often comparable to versions of the programs with low-level synchronization and parallelism constructs that provide no safety guarantees.

5.2 Background

TWEJava uses the hierarchical, region-based effect system adapted from DPJ, which is described in Chapter 2. In our scheduler design, we rely on the hierarchical structure of this effect system (described in detail in Section 2.3.1), with Region Path Lists used to specify the memory regions operated on by effects. We make use of that hierarchical structure in our scheduler design, and we must account for other expressive features of the effect system, such as the use of wildcards.

5.2.1 TWE Model and TWEJava Language

Figure 5.1 shows an example TWEJava program that we use to explain the basic features of the TWEJava language. A *task* is the basic unit of execution managed by the scheduler. Two basic operations are used to create and wait for tasks. `executeLater` creates an asynchronous task and returns a handle for it. The caller can continue executing immediately. `getValue` waits for a task to complete and returns the value returned by that task, if any. The example uses `executeLater` on line 22 to create multiple parallel tasks and `getValue` on line 23 to wait for them.

```
1  region TF;  // Declaring global region name TF
2  class WorkTask extends Task<Void, Void, reads Root> {
3    int ii;
4    WorkTask(int _ii) { ii = _ii; }
5    public Void run(Void _) {
6      int index = f(ii); accumulate(index, ii); return null;
7    }
8  }
9  void accumulate(final int clusterIdx, final int ptIdx) reads Root{
10   new Task<Void, Void, reads Root writes [clusterIdx]>()  {
11     public Void run(Void _) {
12       // centers[clusterIdx][k] is in region Root:[clusterIdx]
13       for (int k = 0; k < nattributes; k++)
14         centers[clusterIdx][k] += attribute[ptIdx].getFeature(k);
15       return null;
16     }
17   }.executeLater().getValue();
18 }
19 void work (final float[][] clusters) reads Root writes TF {
20   TaskFuture[]<TF> tf = new TaskFuture[npoints]<TF>;
21   for (int i = 0; i < npoints; i++)
22     tf[i] = new WorkTask(i).executeLater();
23   for (int i = npoints-1; i >= 0; i--) tf[i].getValue();
24 }
```

Figure 5.1: Running example: KMeans clustering. `f(ii)` computes the cluster index into which features of `attributes[ii]` are accumulated. Multiple attributes may go into the same cluster, requiring `accumulate()` to be atomic. This is achieved by making its body a task.

Every task specifies input and output argument types and an *effect summary*, which must describe a superset of the memory effects of executing an instance of the task, excluding any effects of asynchronous tasks it may in turn create using `executeLater`. The TWEJava compiler statically checks that all effect summaries are sound using a modular, conservative flow analysis. The example in Figure 5.1 declares two kinds of tasks, `WorkTask` (with effect summary `reads Root`) and the anonymous class on lines 10–17 (with effect summary `reads Root writes [clusterIdx]`).

A run-time scheduler uses the information about declared effects to ensure that two tasks are allowed to run concurrently only if their effects do not conflict. One subtlety arises if task A calls `getValue` to wait for task B, but B has an effect that conflicts with A, preventing it from running until A completes. To avoid a resulting deadlock, the scheduler does *effect transfer*: it transfers "ownership" of the conflicting effects from A to B, so that B may execute.

TWEJava also includes `spawn` and `join` operations, which are similar to `executeLater` and `getValue`, but with stricter semantics enforced using static checking. Spawned tasks can be run immediately without using the effect-based scheduler described in this work, because their effects

are conceptually 'transferred' from an already-running parent task, and thus must be non-conflicting with the effects of all other concurrently running tasks. These transferred effects must be accounted for when checking effect conflicts (as shown in Figure 5.8), but for other purposes the scheduler can ignore spawned child tasks.

5.2.2 The Naïve TWEJava Scheduler

The original prototype of TWEJava [48] uses a scheduler with a single queue of tasks protected by a single global lock. It enables a task for execution only once it is safe to do so based on its effects, then hands it off to the Java `ForkJoinPool` framework for execution. When attempting to execute a task, the scheduler generally works by scanning from the 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.

The use of a single task queue of undifferentiated tasks causes high overhead because a task's effects are compared with those of all tasks ahead of it, taking no advantage of information that could allow some comparisons to be skipped. The use of a single global lock means that there is no concurrency in scheduling operations: only a single thread can insert, compare, or launch tasks at a time. The new scheduler proposed in this work takes advantage of the hierarchical structure of effect summaries to provide a much more scalable solution.

5.2.3 Safety Guarantees Provided by TWEJava

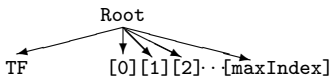
TWEJava gives strong safety guarantees to all conforming programs: (i) data race freedom; (ii) atomicity for a task or portion thereof that does not create or wait for any other tasks; (iii) deterministic, sequential-equivalent semantics for declared portions of an execution that follow certain restrictions; and (iv) safe composition of deterministic and non-deterministic parallel computations, without compromising any of the specified security guarantees. See [48] for a detailed discussion of these properties and how the TWE model guarantees them.

The new scheduler we develop preserves all these guarantees. Section 5.4 provides a proof that our scheduler ensures the fundamental property of task isolation (mutual exclusion of tasks

with conflicting effects), which, in combination with the soundness of task and effect annotations statically enforced by our compiler, is the foundation from which all the above properties are derived.

5.3 Scheduling Tasks with Effects

In this work, we describe a new approach for scheduling tasks with effects which relies on the hierarchical structure of the RPLs that we use to specify effects. In particular, we observe that the dynamic regions can be represented by a tree we call the *RPL tree*, where each node corresponds to some wildcard-free RPL. The tree is rooted at the RPL `Root`, and its nodes are labeled by RPL elements. Each node in the tree represents the RPL identified by the node labels on the path from the root. An RPL containing a wildcard corresponds to a set of nodes, e.g., `Root:A:*` corresponds to the set of all nodes in the subtree rooted at the node for `Root:A`. The RPL tree for the example is simple (we assume for convenience that all cluster indices in the range 0 ... *maxIndex* occur at run-time):



Since each effect in our system is on an RPL, the scheduler can maintain a *scheduling tree* corresponding to the RPL tree, and insert each effect at a node corresponding to a wildcard-free prefix of its RPL.

A key insight is that all effects potentially conflicting with a given effect will be at the same node of the tree as that effect, or, in the case of RPLs with wildcards, at an ancestor or descendant of that node. Effects in sibling subtrees cannot conflict with it, so they need not be explicitly checked when looking for possible effect conflicts. This can greatly reduce the number of effects that need to be explicitly checked against each other. For example, Figure 5.2 shows the scheduling tree at a hypothetical point when a number of tasks have been inserted in it. Before a_4 (an *accumulate* task instance) can start executing, the scheduler only needs to compare the effect ${W^a}_4$ against the effect ${W^a}_1$ and the effects in the parent node `Root`, but not against the effects ${W^k}_1$ or ${W^a}_{\{2,3,5,6\}}$ in sibling nodes.

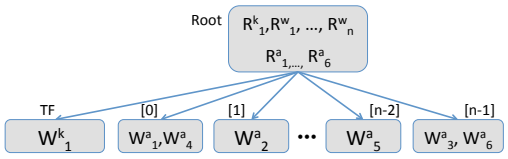


Figure 5.2: Scheduler tree during KMeans execution. R^X_i represents a read effect for the i^{th} instance of task X, $X = k$ for the task executing *work()*, $X = w$ for a *WorkTask*, $X = a$ for an *accumulate* task. Similarly, W^X_i is a write effect. Each effect is shown in the node for its RPL, e.g., the effect `writes Root:[1]` of accumulate task a_2 is in the node for `Root:[1]`.

Moreover, by using locks on individual tree nodes, we can allow scheduling operations on distinct subtrees to proceed concurrently, which is essential to avoid the scheduler becoming a bottleneck in programs that use many small tasks.

In our scheduler, we attempt to schedule a task by inserting each of its effects from the root of the tree, checking each effect for conflicts at each node and if there are none moving down to the next node in the tree as determined by its RPL. If this process reaches the node corresponding to the maximal wildcard-free prefix of the RPL and finds no conflicts there, then, if the effect contains a wildcard, we also check if the effect may conflict with any effects recorded at nodes below it in the tree. If there are still no conflicts found, then we can conclude that no other effects conflict with the new effect, so we may *enable* that effect. Once all the effects of a task are enabled, that task can be run. If, on the other hand, an effect conflict is found at some point, then we record this and do not enable the effect. In this case, we will re-check the effect once the task with the conflicting effect finishes or blocks in a way that may allow the disabled effect to be enabled.

When a `getValue` operation is performed, we re-check the effects of the task that was blocked on to see if it can now be run (possibly due to effect transfer, described in Section 5.2.1), and also mark the task as prioritized, which prompts special handling to ensure that it will be run if it is possible to do so.

5.3.1 Data structures for tree-based task scheduling

Figure 5.3 shows the main data structures our scheduler uses.

The `Node` class represents a node in the scheduler tree, which we use to hold effects. It contains

```
1  class Node {
2    Set<Effect> effects;
3    Map<RPLElement, Node> children;
4  }
5  class Effect {
6    final boolean write;
7    final RPLElement[] rpl;
8    final TaskFuture task;
9    volatile Node node;
10   boolean enabled;
11   Set<Effect> waiters;
12 }
13 class TaskFuture {
14   final Set<Effect> effects;
15   volatile enum {WAITING, PRIORITIZED, ENABLED, DONE} status;
16   volatile boolean rechecking;
17   volatile TaskFuture blocker;
18   ...
19 }
```

Figure 5.3: Data structures used in tree scheduler

a set of effects held at the node, plus a map defining the edges to its child nodes, each identified by an RPL element. `Node`s may be locked and unlocked in our algorithm.

The `Effect` class represents one effect of a task. Its `rpl` field gives the RPL that the effect is on (represented as an array of RPL elements, omitting the initial `Root` element). `write` is true if it is a write effect, and `task` indicates what task it is an effect of. The other fields of `Effect` are updated during the scheduling process. `node` gives the node currently holding this effect, and `enabled` indicates if the effect is enabled. The `waiters` set contains any other effects that are waiting and not currently enabled because they conflict with this effect.

The `TaskFuture` class represents a runnable instance of a task. It has a set of effects, plus three fields updated by the scheduler. `status` indicates if the task is waiting to run, still waiting but prioritized because another task is blocked on this one, enabled to run, or done. It also has a flag set when the effects of this task are being rechecked (explained below), and a reference to the task that this task is blocked on, if any.

We distinguish between *waiting* and *blocked* tasks. An *effect is waiting* if it is not enabled because it conflicts with another effect. A *task is waiting* if it is not enabled because one of its effects is waiting. An executing task is *blocked* if it performs a *getValue* or *join* operation on a task that has not yet completed.

5.3.2 Tree-based task scheduling algorithm

Here we present our task scheduling algorithm. In this section, we will refer to a version of the example shown in Figure 5.1 that has been modified to illustrate more of the features of our scheduler. In this modified example, the method *work()* (run by a task we also call *work*, with the effect `writes TF`) creates and later blocks on another task *scribble* (with the effect `writes Root:*`), as follows:

```
1 void work (final float[][] clusters) writes TF {
2   TaskFuture[]<TF> tf = new TaskFuture[npoints]<TF>;
3   //ScribbleTask has effect writes Root:*
4   TaskFuture scribble = new ScribbleTask().executeLater();
5   for (int i = 0; i < npoints; i++)
6     tf[i] = new WorkTask(i).executeLater();
7   for (int i = npoints-1; i >= 0; i--) tf[i].getValue();
8   scribble.getValue();
9 }
```

Inserting effects

Figure 5.4 shows the algorithm for inserting a set of effects in a node of the tree. Conceptually, an *insert(root, E, 0)* operation can be used to try to schedule a new task with effect set E created by an `executeLater` operation. *insert* will then be invoked recursively to move effects toward the node corresponding to their maximal wildcard-free prefix. (Our actual implementation handles the root node specially for improved performance. See Section 7.5.)

The node n is required to be locked on entry to *insert*. This permits a hand-over-hand locking discipline: as the *insert* operation at one node identifies effects that should be inserted at its child nodes, it locks those nodes. The parent node can then be unlocked before processing begins on the child nodes.

The *insert* algorithm loops over the effects e in the set E and checks whether the node n corresponds to the maximal wildcard-free prefix of their RPLs. If it does, then the effect will definitely be placed at node n (using the *addEffect* helper operation shown in Figure 5.5). In this case, we first check if e conflicts with any other enabled effects at n, and then (if it does not) if it conflicts with any enabled effects at nodes below n. If not, we enable e.

If n does *not* correspond to the maximal wildcard-free prefix of e's RPL, we only check if e conflicts with other effects *at* node n. If so, it will be added at that node, but otherwise it should

```
1  def insert(n : Node, E : Set<Effect>, depth : int):
       // n must be locked on entry
2      effectsBelow ← ∅          // map: Node→Set<Effect>
3      for e in E do
4          if e.rpl.length = depth || isWildcard(e.rpl[depth]) then
5              addEffect(n, e)
6              conflictsHere ← checkAt(n, e, false)
7              if ¬conflictsHere then
8                  conflictsBelow ← checkBelow(n, e, n, false)
9                  if ¬conflictsBelow then
10                     enable(e)
11         else
12             conflictsHere ← checkAt(n, e, false)
13             if conflictsHere then
14                 addEffect(n, e)
15             else
16                 nextElt ← e.rpl[depth]
17                 n′ ← n.children.getOrCreate(nextElt)
18                 childEffects ← effectsBelow.getOrCreate(n′)
19                 effectsBelow.put(n′, childEffects ∪ {e})
20     for (n′,E′) in effectsBelow do
21         lock n′
22     unlock n
23     for (n′,E′) in effectsBelow do
24         insert(n′, E′, depth+1)
```

Figure 5.4: Algorithm to insert a set of effects at a tree node

be inserted at a child node of n. In that case, we look up the child node using `getOrCreate()` (which atomically allocates a new node if it doesn't exist yet), and record a mapping from that node to e.

After all effects have been processed, we lock all child nodes in which effects need to be inserted and then release the lock on n. Finally, we recursively call *insert* on each of these nodes with the corresponding set of effects. Through this process, all the effects will ultimately be inserted at a node.

Checking for conflicts

The *insert* algorithm uses the *checkAt* and *checkBelow* operations shown in Figures 5.6 and 5.7 to check whether an effect conflicts with existing effects at a node or in the subtrees below it, respectively.

```
1 def addEffect(n : Node, e : Effect):
    // n must be locked on entry
2   n.effects ← n.effects ∪ {e}
3   e.node ← n
4 def removeEffect(n : Node, e : Effect):
    // n must be locked on entry
5   n.effects ← n.effects \ {e}
```

Figure 5.5: Adding and removing effects at a node

```
1 def checkAt(n : Node, e : Effect, prio : bool) : bool:
    // n must be locked on entry
2   for e′ in n.effects do
3     if e′.enabled && conflicts(e′, e) then
4       if prio && tryDisable(e′) then
5         e.waiters ← e.waiters ∪ {e′}
6       else
7         e′.waiters ← e′.waiters ∪ {e}
8         return true;
9   return false;
```

Figure 5.6: Checking for conflicts between a new effect and existing effects at a node

checkAt tests if the effect e conflicts with any enabled effect at node n. It takes a flag indicating if e should be prioritized over other effects that have been enabled but whose tasks have not yet been enabled. (A task and its effects are prioritized if it is still waiting to execute but a running task is blocked on it.) In *checkAt*, we loop over all the enabled effects e' at node n and check if e conflicts with them. If it does, then either e is marked as waiting on e' or vice versa. The latter case occurs only if e is prioritized and we are able to disable e' (i.e. mark it as no longer enabled). This will be possible only if e''s task has not yet been enabled. *checkAt* returns *true* if a conflict was found and was not resolved by disabling e'.

In our modified example, the effect of *scribble* would be inserted at node `Root` when *scribble* is created. It would *not* be enabled, however, because it conflicts with the effect of the executing task *work*. *scribble* will be enabled only once *work* blocks on it on line 8. *checkAt* will return false for all the *WorkTask* and *accumulate* tasks even though *scribble* is in the Root node because *scribble* is not yet enabled.

The *checkBelow* operation checks whether effect e conflicts with any effects in the subtrees below node n. It also takes a prioritization flag, and also uses the node n_e, which gives the node

```
1  def checkBelow(n : Node, e : Effect, n_e : Node, prio : bool) : bool:
       // n and n_e must be locked on entry
2      if ¬hasWildcard(e.rpl) then
3          return false;
4      for n' in n.children do
5          lock n'
6          conflictFound ← false
7          for e' in n'.effects do
8              if conflicts(e', e) then
9                  if ¬ e'.enabled || (prio && tryDisable(e')) then
10                     e.waiters ← e.waiters ∪ {e'}
11                     removeEffect(n', e')
12                     addEffect(n_e, e')
13                 else
14                     e'.waiters ← e'.waiters ∪ {e}
15                     conflictFound ← true
16                     break
17         if ¬conflictFound then
18             conflictFound ← checkBelow(n', e, n_e, prio)
19         unlock n'
20         if conflictFound then
21             return true
22     return false
```

Figure 5.7: Checking for conflicts between a new effect and existing effects below a node

containing e. This will be the node corresponding to the maximal wildcard-free prefix of e's RPL. Because *checkBelow* is called only to check subtrees below n_e, we can immediately rule out the possibility of any conflicts if e does not contain any wildcards, because in that case its RPL will be disjoint from any other RPL having a longer wildcard-free prefix.

If e's RPL contains a wildcard, *checkBelow* recurses over the subtrees below n, checking for any conflicting effects through a procedure similar to *checkAt*. For example, *scribble*'s effect `writes Root:*` will find a conflict in node `Root:TF` with the effect `writes TF` of the *work* task. *checkBelow* must check even effects that are not enabled, because if conflicts are found among those effects, they must be moved up the tree to n_e. This is needed so that when subsequently rechecking those effects starting from their containing node, the conflicting effect e will be encountered.

checkAt and *checkBelow* call *conflicts*, shown in Figure 5.8, to check whether effects conflict. Two effects normally conflict unless they are on disjoint RPLs or are both read effects, but *conflicts* also implements the TWE effect transfer mechanism: two effects are treated as non-conflicting if

```
1  def conflicts(e′ : Effect, e : Effect) : bool:
2      if e.task = e′.task then
3          return false
4      if (¬e.write && ¬e′.write) || e.rpl # e′.rpl then
5          return false
6      if blockedOn(e′.task, e.task) then
7          for t in spawnedChildTasks(e′.task) do
8              for e″ in t.effects do
9                  if conflicts(e″, e) then
10                     return true
11         return false
12     return true
```

Figure 5.8: Checking whether two effects conflict

```
1  def blockedOn(t′ : TaskFuture, t : TaskFuture) : bool:
2      blocker ← t′.blocker
3      while blocker ≠ null do
4          if blocker = t then
5              return true
6          blocker ← blocker.blocker
7      return false
```

Figure 5.9: Checking whether task t' is blocked on task t

one of their tasks is blocked on the other directly or indirectly. The *blockedOn* helper operation in Figure 5.9 is used to check for this. A slight complication is that if the blocked task has previously transferred some of its effects to a child task created using *spawn*, we must check for conflicts with the child task's effects too. We formally define when effects conflict in Section 5.4.

Enabling and disabling effects

The *enable* method, shown in Figure 5.10, is called to enable an effect. If all the effects of its task are now enabled, then it also enables the task and submits it to a lower-level, non-effect-aware task framework for execution. For clarity, we show the check of whether all the task's effects are enabled as occurring in an atomic block, so that there is no possibility of some of the effects being disabled while the check is being done. Our actual implementation maintains a count of the disabled effects of each task using a Java `AtomicInteger`, and the atomic blocks in *enable* and *tryDisable* are implemented using atomic operations on it, so no run-time support for general atomic blocks is required.

```
1  def enable(e : Effect):
       // e.node must be locked on entry
2      e.enabled ← true
3      atomic
4          allEnabled ← ∀ e′ ∈ e.task.effects, e′.enabled = true
5      if allEnabled then
6          e.task.status ← ENABLED
7          submit e.task to low-level task scheduler for execution
8  def tryDisable(e : Effect) : bool:
       // e.node must be locked on entry
9      atomic
10         canDisable ← ∃ e′ ∈ e.task.effects s.t. e′.enabled = false
11         canDisable ← canDisable && ¬ e.task.rechecking
12         if canDisable then
13             e.enabled ← false
14     return canDisable
```

Figure 5.10: Enabling and disabling effects

tryDisable tries to disable an effect, potentially allowing a conflicting prioritized effect to be enabled instead. This is possible only if not all of the task's effects have been enabled (and thus it has not yet been submitted to the low-level task scheduler). Also, we cannot disable effects of a task that is being rechecked, a process we will describe later. In our implementation the rechecking flag is represented by a special range of negative values for the task's disabled-effects counter, permitting the atomic block in *tryDisable* to be implemented in terms of operations on that single `AtomicInteger`.

Waiting for tasks and rechecking tasks and effects

Figure 5.11 shows the *await* procedure used when awaiting the completion of a task t with `getValue` or `join`. It first checks if t is already done and returns if so. Otherwise, if t is still in the WAITING state, then it is updated to PRIORITIZED. t is then recorded as the *blocker* for the current task t', meaning that t' will not resume until t has finished. In our modified example, the *work* task calls *await* on *scribble*, marking *scribble* as prioritized and recording it as the blocker for *work*.

await then iterates over the chain of blockers starting from t and including any tasks it is transitively blocked on, until a task that has not yet been enabled is encountered. At that point, it will call the *recheckTask* operation to see if that task can now be enabled. (In our example,

```
1  def await(t : TaskFuture):
2      if t.status = DONE then
3          return;
4      atomic
5          if t.status = WAITING then
6              t.status ← PRIORITIZED
7      t′ ← getCurrentTask()
8      t′.blocker ← t
9      t_bl ← t
10     while t_bl ≠ null do
11         if t_bl.status < ENABLED then
12             recheckTask(t_bl)
13         t_bl ← t_bl.blocker
14     Call low-level task scheduler to await completion of t
15     t′.blocker ← null
```

Figure 5.11: Waiting until a task is done

recheckTask is called on *scribble*.) This is important because the fact that t' is now blocked on a task t_{bl} may allow one or more of t_{bl}'s effects to be enabled thanks to effect transfer from t', potentially allowing it to run when earlier it could not. (In fact, this will finally allow *scribble* to run.) After the loop, a low-level task scheduler operation is used to await the completion of t.

Figure 5.12 shows *recheckTask* and *recheckEffect*, which are called when a task or effect could not previously be enabled but now possibly can be. In addition to being called when a task is blocked on, *recheckTask* can also be called when a conflicting task has ended. (We can also prioritize and recheck an arbitrary task in the very rare case that there are waiting tasks remaining but no tasks currently running. This serves to ensure that the scheduler does not give rise to deadlocks.)

recheckTask first takes a global lock, so only one task at a time can undergo rechecking. This avoids a case where multiple tasks with conflicting effects could each have their effects rechecked, but in different orders, possibly resulting in no task having all its effects enabled. The *rechecking* flag for the task t is then set to true, which prevents its effects from being disabled, again preventing cases that could result in no progress being made. *recheckTask* then loops over t's effects, locking their containing nodes using the *lockContainingNode* operation in Figure 5.13 and then calling *recheckEffect* on them. In the modified example, when *recheckTask* is called for *scribble*, it calls *recheckEffect* for the effect `writes Root:*`.

recheckEffect re-checks a single effect that could not previously be enabled. It is broadly similar

```
1  def recheckTask(t : TaskFuture):
2      lock recheckLock
3      t.rechecking ← true
4      for e in t.effects do
5          n ← lockContainingNode(e)
6          if ¬ e.enabled then
7              recheckEffect(e, true)
8              if t.status = ENABLED then
9                  break
10         else
11             unlock n
12     t.rechecking ← false
13     unlock recheckLock
14 def recheckEffect(e : Effect, prio : bool):
       // e.node must be locked on entry
15     n ← e.node
16     conflictsHere ← checkAt(n, e, prio)
17     if ¬conflictsHere then
18         d ← depthOf(n)
19         if e.rpl.length = d || isWildcard(e.rpl[d]) then
20             conflictsBelow ← checkBelow(n, e, n, prio)
21             if ¬conflictsBelow then
22                 enable(e)
23         else
24             removeEffect(n, e)
25             n′ ← n.children.getOrCreate(e.rpl[d])
26             lock n′
27             addEffect(n′, e)
28             unlock n
29             goto 15
30     unlock n
```

Figure 5.12: Rechecking tasks and effects to see if they can now be enabled

in operation to *insert* except that it starts from the node currently containing the effect e and may move e down toward lower nodes if e has no conflicts at its current node and is not at the node corresponding to the maximal wildcard-free prefix of its RPL. *recheckEffect* also takes a priority flag indicating if another task is blocked on *e.task*, in which case other tasks' effects may be disabled in favor of e.

In the modified example, `writes Root:*` is at the node corresponding to the maximal wildcard-free prefix of its RPL and cannot be moved down. After *work()* calls *getValue()* on all *WorkTask* tasks, all those tasks and their child accumulate tasks will have completed, leaving `writes TF` in node `Root:TF` as the only enabled effect. When *work* blocks on *scribble*, *conflicts* will ignore the

```
1  def lockContainingNode(e : Effect) : Node:
2      n ← e.node
3      if n = null then
4          goto 2
5      lock n
6      if e.node ≠ n then
7          unlock n
8          goto 2
9      return n
```

Figure 5.13: Locking an effect's containing node

```
1  def taskDone(t : taskFuture):
2      t.status ← DONE
3      for e in t.effects do
4          n ← lockContainingNode(e)
5          removeEffect(e.node, e)
6          unlock n
7          for e′ in e.waiters do
8              n ← lockContainingNode(e′)
9              if ¬e′.enabled then
10                 prio ← e′.task.status = PRIORITIZED
11                 recheckEffect(e′, prio)
12                 if prio && e′.task.status = PRIORITIZED then
13                     recheckTask(e′.task)
14             else
15                 unlock n
```

Figure 5.14: Actions to perform when a task completes

conflict between the effects of *scribble* and *work*. Therefore, *checkAt* and *checkBelow* will find no conflicts, and so *recheckEffect* will enable the effect of *scribble*. This will enable the task itself, allowing it to execute.

Task completion

Figure 5.14 shows the steps done after a task t completes. It sets its status to *DONE* and removes its effects from the tree. It also rechecks the effects that conflicted with effects of t. As an optimization, it first tries to recheck only the conflicting effect. If that does not allow the task to be enabled, then *recheckTask* is called to recheck all the effects of the task. This may be necessary in order to enable the task if some of its effects have been disabled.

5.4 Task Isolation

In this section, we will prove that our scheduling algorithm ensures task isolation, i.e. that tasks with conflicting effects will not run concurrently. We first formally define some terms.

Definition 1. *An effect e is* enabled *if $e.enabled = true$ and $e.task.status \neq DONE$.*

We regard the effects of completed tasks as no longer enabled, since their task is no longer running (and will not resume running), even though their *enabled* field is still set.

Definition 2. *A task t is* enabled *if it has been submitted to the low-level scheduler for execution and $t.status \neq DONE$.*

A task will not be run until it is submitted to the low-level task scheduler and is done executing by the time *taskDone* sets *t.status* to *DONE*, so it may only execute while it is enabled.

Definition 3. *Two effects e_1 and e_2 are* non-conflicting, *written $e_1 \# e_2$, if at least one of the following holds:*

1. *e_1 and e_2 are effects of the same task*
2. *e_1 and e_2 are both read effects*
3. *e_1 and e_2 are on disjoint RPLs*
4. *e_1's task is blocked on e_2 and each effect of a spawned subtask of e_1's task is non-conflicting with e_2, or vice versa.*

Otherwise, we say e_1 and e_2 conflict.

We now prove two lemmas about effects' positions in the tree.

Lemma 3. *If an effect e is contained at node n, e has the wildcard-free RPL prefix corresponding to n.*

Proof. This follows from the structure of the *insert*, *recheckEffect*, and *checkBelow* operations (the only ones which may add an effect at a node). *insert* starts from the root node and may subsequently invoke itself recursively on child nodes corresponding to successive non-wildcard elements in the RPLs of effects passed to it. This ensures that when *insert* is called with a given effect in the effect set E, it is always with a node n that corresponds to a wildcard-free prefix of the RPL of that effect. *recheckEffect* also similarly moves effects down the tree to deeper nodes corresponding to successive non-wildcard elements of their RPLs, ensuring that this property holds. *checkBelow* may move an effect only to an ancestor node of its current node, which preserves this property. □

Lemma 4. *If an effect e is enabled, it is contained at the node n corresponding to the maximal wildcard-free prefix of its RPL.*

Proof. This is the case because before any effect is enabled it must satisfy the condition in line 4 of *insert* or line 19 of *recheckEffect*. This condition states that the node depth is either equal to the full length of the RPL, or the next RPL element is a wildcard element. By Lemma 1, the wildcard-free prefix corresponding to n is a prefix of the RPL of e, and by the condition above, this prefix must be maximal. Therefore, this condition will hold if and only if the node into which the effect is about to be inserted (in the *insert* case) or is already inserted (in the *recheckEffect* case) corresponds to the maximal wildcard-free prefix of its RPL. Because that node is locked when the condition is checked and remains so until after the effect is enabled, no other concurrent operation may remove the effect from that node in the interim.

The only operations that could subsequently remove the effect from that node are *recheckEffect*, *checkBelow*, and *taskDone*. But *recheckEffect* is only called for effects that are not enabled (and may not be enabled by a separate concurrent operation because their node is locked). *checkBelow* will not move effects while they are enabled. (If an effect is enabled when first checked in *checkBelow*, it may not be moved unless a *tryDisable* operation on it succeeds. As with *recheckEffect*, an effect found to be disabled in *checkBelow* may not be enabled by a separate concurrent operation because its node is locked.) *taskDone* will remove an effect from a node only once its task is done, at which point the effect is no longer enabled. Thus, the claim is satisfied in all cases. □

We now prove two key invariants maintained by our scheduler, which allow us to conclude that certain effects are non-conflicting based on their relative positions in the tree.

Lemma 5. *If an effect e at node n is enabled, then for all effects e' contained in the subtrees rooted at the child nodes of n, $e \# e'$.*

Proof. This property holds when e is initially enabled, because this is done only after the *checkBelow* operation concludes that there are no conflicts between e and any effects in the subtrees below the one containing e (in line 8 of *insert* or line 20 of *recheckEffect*).

Because the *insert*, *recheckEffect*, and *checkBelow* operations use hand-over-hand locking, any two *insert* or *recheckEffect* operations that access one or more of the same nodes (including through recursive invocation of *insert* or *checkBelow*) will access all such nodes in the same order, so we may think of those two operations as ordered with respect to each other even though the actual timing of the operations may overlap. We can extend this to a total order by arbitrarily ordering operations that access disjoint sets of nodes.

The effect e may be enabled only after a *checkBelow* operation (within the same *insert* or *recheckEffect* operation that enables it) returns *false*. This indicates that there are no conflicts between e and any effects in the subtrees below n. This is trivial if e has no wildcards, but otherwise *checkBelow* will recurse over the subtrees below n and return *false* only if it has concluded they contain no effects that conflict with e. (In some cases *checkBelow* will find conflicting effects and move them up the tree to node n: those effects are no longer in the subtrees below n by the time *checkBelow* returns.) Thus, e may have no conflicts with effects in those subtrees that were placed there before the operation that enabled e in terms of the ordering discussed above.

Now we must consider operations that occur after e is enabled in terms of that ordering and that may add new effects in the subtrees below n. These are in two categories. The first is *recheckEffect* operations that start at nodes below n: these will not add new effects to those subtrees. The second is *insert* or *recheckEffect* operations that start at n or an ancestor of n but eventually add effects in subtrees below n. Before doing so, any such operation would perform a *checkAt* operation at n, checking for conflicts with enabled effects at n. Thus, they will be checked against e (which must remain at n as long as it is continuously enabled, by Lemma 4). If they conflict with e they will

be placed at n rather than at a node below it, so the desired property will be preserved. □

We may also restate this lemma as the following corollary:

Corollary 2. *If an effect e is at node n, then for all enabled effects e' at ancestor nodes of n, $e \# e'$.*

Lemma 6. *If an effect e at node n is enabled, then for all other enabled effects e' contained at the same node as e, $e \# e'$.*

Proof. This property holds when e is initially marked enabled by an *insert* or *recheckEffect* operation, because that is done only after a *checkAt* operation verifies that e does not conflict with any other enabled effect at n. Since n is locked from before the start of the *checkAt* operation until after e is enabled, no other effects at n may be enabled in the interim. The property remains true as long as e is enabled, because any other effect at n could only be enabled after a *checkAt* operation concludes that it does not conflict with e. □

Now we are ready to show that all enabled effects are mutually non-conflicting, and therefore task isolation is maintained.

Lemma 7. *If an effect e at node n is enabled, then for all other enabled effects e', $e \# e'$.*

Proof. By Lemmas 6 and 5, and Corollary 2, respectively, e has no conflicts with other enabled effects (a) at node n, (b) at nodes below n, and (c) at ancestors of n.

What remain are enabled effects at nodes other than the ancestors of n or the subtree rooted at n. By Lemma 2, any such effect e' will be contained at the node n' corresponding to the maximal wildcard-free prefix of its RPL. By the tree structure of RPLs, that node corresponds to a wildcard-free RPL prefix p' that is different from the prefix p corresponding to n, with neither being a prefix of the other. So we can identify some position at which p and p' have differing non-wildcard RPL elements. Under the rules described in Section 5.2 [48], this means that any two RPLs having p and p' as prefixes are disjoint from each other. From this we can conclude that for any effect e' at such a node, $e \# e'$. □

Lemma 8. *If task t is enabled, then all effects of t are enabled.*

Proof. *enable* and *tryDisable* ensure this. *enable* submits a task to the low-level scheduler only once all its effects are enabled. Once all its effects are enabled, they cannot be disabled in *tryDisable*, so they will remain enabled until t is done. □

Theorem 3 (Task isolation)**.** *If tasks t_1 and t_1 are concurrently enabled, then for all pairs of effects e_1 of t_1 and e_2 of t_2, $e_1 \# e_2$.*

Proof. This follows from Lemmas 8 and 7. By Lemma 8, all the effects of t_1 and t_2 are enabled. Therefore, by Lemma 7, $e_1 \# e_2$ for all pairs of effects e_1 of t_1 and e_2 of t_2. □

5.5 Implementation for TWEJava

We implemented our tree-based scheduler for the TWEJava language. Once tasks are enabled in the scheduler, we normally submit them to Java's `ForkJoinPool`, which executes tasks using work-stealing. However, where possible our scheduler will run tasks in the same thread that blocks on them. Our implementation includes several additional optimizations beyond the basic algorithm described in Section 5.3.

5.5.1 Optimizing critical sections

We support an operation called `execute`, which is semantically equivalent to an `executeLater` immediately followed by a `getValue`. This is useful for tasks that act as a critical section within a larger task: the scheduler treats the `execute`d task as prioritized when it is first inserted in the tree, and tries to run it in the same thread as the task that calls `execute`.

5.5.2 Avoiding contention at the root node

We handle *insert* operations at the root node specially to improve scalability. The basic *insert* algorithm would require taking an exclusive lock on the root node every time a task is scheduled, which can become a bottleneck. Instead, we use a read-write lock for the root node, allowing

concurrent readers or a single writer. The write lock needs to be acquired only if an effect is being inserted whose maximal wilcard-free RPL prefix is `Root`, or if an effect with a wildcard is already present at the root node. Otherwise, only the read lock is taken. Adding new child nodes of the root node only needs the read lock, because we use a concurrent hash map operation to atomically implement *getOrCreate*.

We ensure that locks for child nodes of the root node are acquired in a consistent order and use the ordinary *insert* operation for all other nodes in the tree. This ensures that any two *insert* or *recheckEffect* operations still access all nodes in the same order, except that read-only accesses to the root node may be in a different order (which has no observable effects).

This approach could potentially be applied at some or all other nodes as well, but we have not done this because the read-write locking scheme involves higher time and space overheads, so we believe it would only be beneficial at nodes with significant contention. As a simple heuristic, we expect the root node to be more highly contended than most other nodes, since all effects must be inserted starting from the root. Moreover, it is often possible to restructure the effects used in a program so that most contention occurs at the root node (e.g. by using RPLs that differ in the first position after `Root`).

5.5.3 Avoiding unnecessary effect conflict checks

While our scheduler design as presented in Section 5.3 avoids many unnecessary effect conflict checks because of the tree structure it uses, it still examines effects for potential conflicts in some cases where simple logic could show that those checks are unnecessary. In particular, these include the following situations:

- Two read effects may not interfere with each other, so when the effect e being checked is a read effect, *checkAt* and *checkBelow* need not check against other read effects.
- Effects e' that are contained at the node n but are not enabled will not be identified as conflicting in *checkAt* (Figure 5.6).
- The `checkAt` call on line 12 of *insert* (Figure 5.4) needs to check only against effects at n

that are enabled (as mentioned above) *and* have a 'tail' of one or more RPL elements beyond the wildcard-free RPL prefix corresponding to n. This is the case because the effect e being checked has at least one more non-wildcard element beyond that prefix, so it cannot conflict with effects whose RPL consists just of that prefix.

In our implementation, we avoid all these unnecessary checks by storing the effects at a node in six separate sets, as follows:

1. Enabled read effects with a tail
2. Enabled read effects with no tail
3. Enabled write effects with a tail
4. Enabled write effects with no tail
5. Disabled read effects
6. Disabled write effects

Whenever conflict checks are done, our implementation will check only against the sets that could potentially contain a conflicting effect, completely avoiding checks against those that are known to be non-conflicting based on the conditions given above.

5.5.4 Interoperation with Java atomics

The TWE model is designed to preclude the need for lower-level concurrency mechanisms, such as Java's `synchronized`, which lack TWE's strong guarantees. Java's atomic classes, however, can be used safely in TWE, and we exploit this in our TSP benchmark (described later) for improved performance. Those classes encapsulate a value which may only be accessed using the class's atomic operations. We may think of each such value as being in its own unique region, distinct from the regions in the RPL tree. Each operation on it is semantically equivalent to using `execute` to run a task with a read or write effect on that region alone. Thus, these operations can be used while preserving the safety guarantees of the TWE model.

Chapter 6

Evaluation

I have carried out evaluations of the tasks with effects model and the TWEJava language by writing or porting several concurrent programs in TWEJava and evaluating their performance. One key result is this evaluation is to demonstrate that TWEJava and the tasks with effects model can express a variety of concurrent programming styles used in real-world applications. Another result is to show that TWEJava can achieve good performance with substantial parallel speedups, often comparable to versions of the programs written using other concurrency and synchronization constructs that do not give strong safety guarantees.

I evaluate the performance of both my initial naïve scheduler and the current, much more scalable tree-based scheduler below. The naïve scheduler could give substantial speedups for some programs with relatively coarse-grain parallelism, but was limited in its scalability and its ability to efficiently support fine-grain parallelism. The tree-based scheduler, which is the focus of my ongoing work, offers much better scalability for programs with fine-grain parallelism, as demonstrated in the results below.

6.1 Expressiveness

In our initial evaluation of the TWE model's expressiveness, we ported four existing concurrent programs to TWEJava and wrote one new application in it. The first two programs are interactive applications which use a combination of event-driven concurrency and structured parallelism for performance.

FourWins: The first ported program is an interactive Connect Four game implementation called FourWins, which was ported from an original code that used JCoBox [79], an actor-like con-

current 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.

ImageEdit: 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 [30]), 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 [22], 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 the STAMP benchmarks [69]); and a Monte Carlo financial simulation, originally from the Java Grande parallel benchmarks [26]. 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.

Barnes-Hut: 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.

Monte Carlo: 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

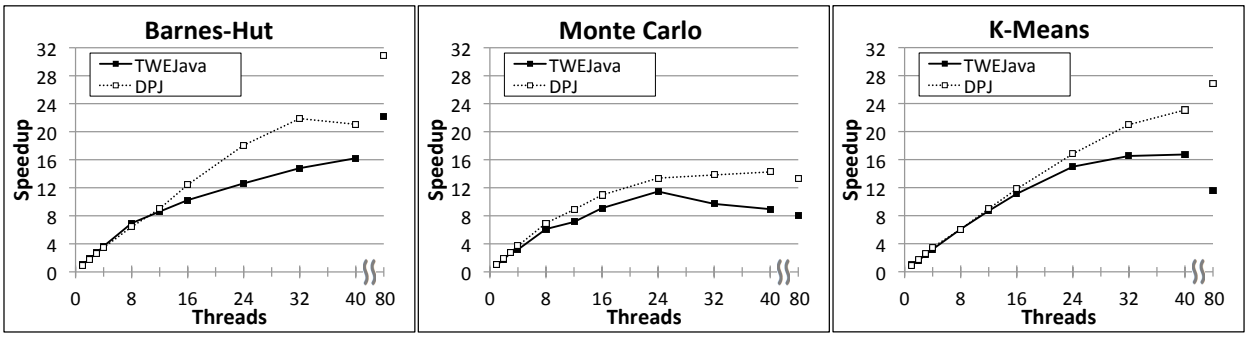


Figure 6.1: Parallel speedups of benchmarks ported from DPJ, showing performance of TWEJava and DPJ versions. These speedups are for the parallel portion of each code and are relative to the DPJ code compiled and run in sequential mode, in which the DPJ parallelism constructs are erased by the compiler, creating a sequential program with no run-time overheads related to parallelism constructs.

insert manual locking operations. As with Barnes-Hut, we create one task per thread in the parallel loop.

K-Means: 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 with initial naïve scheduler

In this section, we evaluate the TWEJava system with the initial naïve scheduler design described in Section 3.4.2, which uses only a single queue protected by a single lock to manage all scheduling

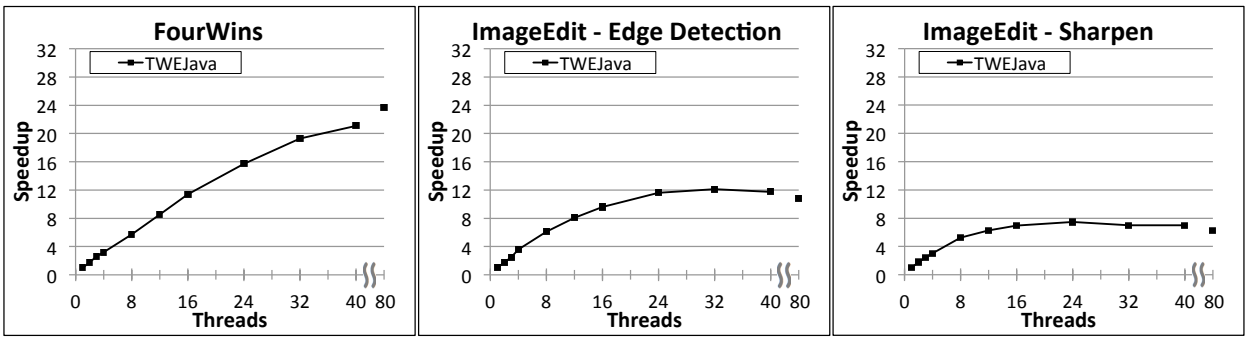


Figure 6.2: Speedups for the FourWins AI computation and two filters in the ImageEdit application. We did not have pure sequential versions of these programs available for comparison, so we give speedups relative to the TWEJava codes run using one worker thread and configured so that the major potentially-parallel computations in the codes each run as a single task, thereby minimizing task-related overheads.

operations. 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 6.1 and 6.2 report the speedups achieved in the parallel portion of each code, relative to the sequential running time. For ImageEdit, we report speedups for both the edge detection filter and the sharpening filter. For the three DPJ codes, we could use the DPJ sequential versions with no parallelism overhead as a sequential baseline, because the DPJ compiler erases all parallelism constructs and has no run-time overhead associated with them. For FourWins and ImageEdit, no such equivalent sequential version exists, so we ran the TWEJava version using a single thread and with configuration parameters that minimize the number of tasks used for the main parallel computation phase of each benchmark (which should reduce per-task overheads relative to the default configuration used for the parallel runs of each benchmark). The performance results use an average over 11 runs for each configuration. 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 [22].

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 initial naïve scheduler implementation is that it 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 single-queue scheduler 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 this evaluation showed that our initial single-queue scheduler could provide significant speedup in some cases, it also could become a bottleneck for many computations, particularly those using many fine-grained tasks. This motivated us to develop the more scalable tree-based scheduler; we evaluate its performance in the next section.

6.3 Performance with tree-based scheduler

We evaluated the performance and scalability of our tree scheduler using six benchmarks. For three of the benchmarks, we compared against versions that used native Java concurrency mechanisms that give no safety guarantees. We also compared our tree scheduler against the previous single-queue TWEJava scheduler, which is known to have fairly poor scalability on benchmarks with fine-grained parallelism due to its design. Providing a scheduler with better performance characteristics is important for making the TWE model usable in practice.

The k-means clustering algorithm (originally adapted from the STAMP benchmarks) groups a set of points into K clusters. The computation consists of a parallel loop with a reduction step. We used an input of 50,000 points, and ran it with K=25000, 5000, and 1000. Short reduction tasks are used to update data related to each cluster during the computation. The lower the value of K, the more objects are in the same cluster, resulting in increased contention. (This is the same k-means code evaluated with our single-queue scheduler above, but in that case we only considered K=25000.)

SSCA2 (also adapted from STAMP) uses parallel tasks to add nodes to a large, directed multi-graph. It uses many short transaction-like tasks to protect accesses to adjacency arrays.

TSP computes a minimum-weight Hamiltonian cycle in a weighted graph using a recursively

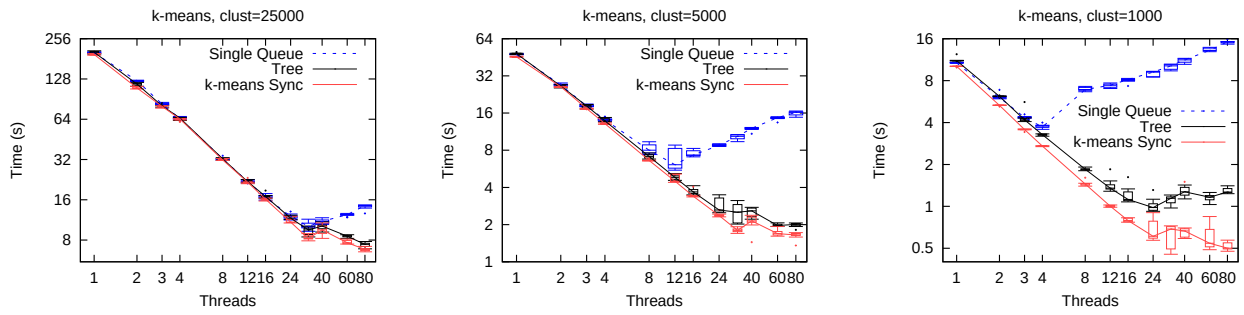


Figure 6.3: Performance scaling of the k-means benchmark. We compare the TWEJava version run with our tree scheduler and with the existing single-queue TWE scheduler, and also compare against a version using Java `synchronized` blocks instead of TWE tasks for fine-grained reductions (labeled 'k-means Sync'). We show results for K=25000, K=5000, and K=1000. Dots indicate outliers. Boxes and whiskers show quartiles (min, 25%, median, 75%, max).

parallel algorithm. Each time a solution is found, we atomically update a globally shared best tour, which is used to prune the search. We used Java's `AtomicLong` to read and update this value, avoiding the overhead of using TWE tasks for such a fine-grained operation. (As explained in Section 7.5, TWE programs may safely use Java's atomics.) We also used a parallelism cut-off, whereby after a predefined recursive depth we switch to a sequential version of the algorithm in order to avoid excessive scheduling overheads.

The other three benchmarks are Barnes-Hut, Monte Carlo, and FourWins. Barnes-Hut measures the force computation in an n-body simulation. Monte Carlo (originally from the Java Grande benchmarks) performs a financial simulation. FourWins measures the AI computation for a Connect Four game. These benchmarks were previously used to evaluate TWEJava's existing single-queue scheduler. The single-queue scheduler was not a major performance bottleneck for these benchmarks because they use smaller numbers of parallel tasks than the above ones, but we want to ensure their performance remains similar with our tree scheduler.

We ran our measurements on a machine with four Intel Xeon E7-4860 processors (with a total of 40 cores and 80 hardware threads) and 128GB of memory. It runs 64-bit Scientific Linux 6 with kernel version 2.6.32-504 and 64-bit Oracle JDK 7u40. Figures 6.3 and 6.4 plots the benchmark running times. We used $\log_2$ scales on both axes to easily see if performance scales linearly with thread count. We performed 11 runs for each configuration; the lines show median running times,

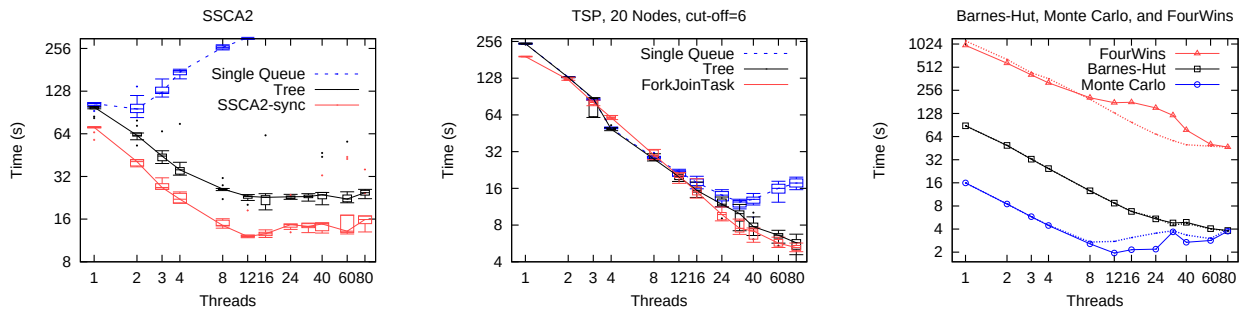


Figure 6.4: Performance scaling of the other benchmarks. We compare the TWE version of SSCA using our tree scheduler with a version using Java `synchronized` blocks instead of TWE tasks for fine-grained reductions. We compare TSP with a ForkJoinTask version. Both are also compared against the TWEJava version run with the existing single-queue scheduler, which provides inferior scalability. We show results for FourWins, Barnes-Hut, and Monte Carlo using our tree scheduler (solid lines) and the single-queue scheduler (dotted lines).

and box-plots show their variability.

For k-means and SSCA2, we compared against versions of our TWE codes using Java `synchronized` blocks instead of TWE tasks for the fine-grained critical sections. (These versions, labeled as 'sync', used the tree scheduler for the small number of remaining TWE tasks. There are too few tasks for the scheduler to have a significant performance impact, so these versions act as proxies for pure Java versions using `synchronized`.) For TSP, we also compared against a version using Java's `ForkJoinTask` framework instead of TWE. Java `synchronized` blocks and `ForkJoinTask` are lower-level mechanisms that have been carefully tuned for performance by the JDK developers, and do not provide parallel safety guarantees like ours, leaving the user responsible for any necessary checks. The comparison against these versions of the codes therefore serves to compare the performance of the TWE programming model as implemented with our tree scheduler against a simpler programming model offering no safety guarantees.

In k-means, the tree scheduler scaled well for all K values tested, and was generally only marginally slower than the `synchronized` version. There was a somewhat larger performance gap in the K=1000 case with high thread counts, but the tree scheduler still showed substantial scalability. With the single-queue scheduler, for comparison, speedups degraded severely as we reduced K (increasing contention).

For SSCA2, the tree scheduler is about 40% slower than the Java `synchronized` version, though

the scalability is very similar in both versions. The improvement compared to the single-queue scheduler is dramatic: the tree scheduler gives useful speedups where the single-queue scheduler slows down substantially.

For TSP, the tree scheduler scales similarly to the ForkJoinTask version, with only marginally slower performance at high thread counts. The single-queue scheduler is comparable until about 12 threads, but the tree scheduler outperforms it at higher thread counts, and the single-queue scheduler's performance degrades above 32 threads while the tree scheduler continues scaling.

The remaining three benchmarks all showed significant performance scaling, generally similar to what they achieved with the previous single-queue scheduler. On FourWins, our tree scheduler provides inferior scaling to the single-queue scheduler on 12–40 threads, but very similar scaling at higher and lower thread counts. Monte Carlo slows down beyond 12 threads, but the performance is similar using both schedulers, with the tree scheduler sometimes a bit faster. Barnes-Hut performs very similarly with both schedulers.

In summary, these results demonstrate that our tree scheduler achieves good speedups, even for codes that use fairly fine-grain tasks. In many cases, the performance of TWE codes is competitive with codes that provide no safety guarantees. Most importantly, the tree scheduler enables users to achieve strong safety guarantees with only a small or no sacrifice of performance in many cases, which is essential for a safe parallel language to be widely accepted.

6.4 Summary

My evaluation has shown that the TWE model as implemented in TWEJava is able to express a wide variety of programs, including ones that more restrictive models cannot express, and deliver good performance and scalability for them. The major outcomes of this evaluation are as follows:

- The TWE model, and in particular TWEJava, can be used to express a wide variety of concurrent and parallel programs, including event-driven concurrent programs that cannot be expressed by structured parallelism models like DPJ's. It gives strong safety guarantees including task isolation and data race freedom for all these programs.

- TWEJava can express programs that combine structured concurrency for parallelism with other forms of concurrency, expressing all the concurrency in the program using a single unified model and giving strong safety guarantees for the program as a whole. In many cases where the parallel computation or a portion of it is deterministic, TWEJava can give a static guarantee of determinism for that part, while allowing it to be combined with other parts of the program that may not be deterministic.

- The TWEJava system can deliver substantial parallel speedups for a variety of programs. In many cases performance is similar to DPJ codes or codes using unsafe, lower-level parallelism mechanisms, which do not have any runtime overheads from effect-based task scheduling. Even the naive single-queue scheduler can deliver substantial speedups on a variety of codes, although its scalability is limited, particularly in codes using many fine-grain tasks.

- Codes using fine-grain tasks can achieve good performance and scalability when run with TWEJava's tree-based task scheduler. In many cases, the performance of TWE codes is competitive with codes that use lower-level parallelism mechanisms without any safety guarantees.

Chapter 7

Enabling Dynamic Effects Without Sacrificing Safety

7.1 Introduction

The TWE model as described in the previous chapters of my thesis offers very strong parallel safety guarantees while also being able to express a wide range of parallel and concurrent programs. It can guarantee task isolation, data race freedom, and atomicity for all programs, and determinism for certain programs or portions of programs where the programmer desires it. The basic TWE model as already described is more expressive that any other parallel programming model we are aware of that provides such strong safety guarantees.

Nonetheless, the basic TWE model still has a limitation in its expressivity. It requires the programmer to write a static specification of the effects of a task, which can be statically verified to cover all its possible run-time effects. This is possible for many programs, but it is difficult or impossible in cases where the total set of data operated on by a task at run time is determined by the dynamic data itself. For example, an operation on a node in an irregular graph may affect that node and all its neighbors. That set of neighbors, or even the number of them, is not generally known statically. (It may be possible to give a highly conservative static summary of the effects of such operations, e.g. saying that they may access any data in a large data structure, but this would force all such operations to be serialized, which is contrary to our desire for parallelism.)

Some algorithms use even more complicated dynamic criteria to determine the set of data to be operated on. For example, the Delaunay mesh refinement algorithm [31, 60] works by 'refining' certain triangles in a mesh. When it does such a refinement step, it computes a 'cavity' consisting of the 'bad' triangle in need of refinement and other nearby triangles meeting certain criteria, which will all be replaced by new triangles in the refinement process. This cavity computation is an

iterative process that may add new triangles to the cavity based on the ones already identified, so the only way to determine and specify the full cavity is to do this dynamic computation. Such an algorithm cannot be supported in the existing TWE model in a way that allows for parallelism, because the effects of the refinement task cannot be specified statically.

Some other programming models can handle algorithms like this, in which the effects of a certain operation are only known as it is executing. However, none of these models give the strong safety guarantees that TWE does. Blelloch et al. [19] propose a mechanism called 'deterministic reservations' which can support algorithms with dynamic effects (although it is limited to those that can be stated as parallel loops) and can be shown to provide determinism if used correctly, but it requires custom code to be written for the reservations in each parallel computation. This code may be fairly complicated and rely on algorithm-specific properties, and the system provides no automatic mechanism for checking that it is correct. Galois [60, 66, 75] also can support algorithms with dynamic effects, and provides guarantees such as data race freedom if used correctly (e.g. only accessing shared data through certain APIs, and correctly specifying if accesses are commutative), but it provides no static or dynamic mechanism for enforcing these correctness requirements.

In this work, we present an extension to the TWE model to handle dynamic effects, allowing tasks to add certain effects to themselves while they are executing. This extension permits the TWE model to support many parallel computations where the effects of a task are only known dynamically. To do this, we extend the TWEJava language to allow object references to be used as memory region specifications. We then extend the effect system to allow the effects of a task to be statically specified in terms of accesses to a set of regions specified by object references, but without statically defining what those object references will be. Instead, we allow references to be added to these *dynamic reference sets* at run time, as the task executes.

We still need to ensure that the run-time effects of two tasks cannot conflict. We augment the TWE system's existing combination of static and dynamic checks with additional checks to handle these dynamic reference sets. As in the original TWE system, we statically check that the actual memory effects of each operation in the program will be covered by the *covering effects*, i.e. the effects of the task that the operation is running in, as defined by the combination of static and

dynamic effect specifications. To support dynamic effects in this analysis, we add a new dataflow analysis that will identify the set of variables known to be in each dynamic reference set at each point in the program. Based on this, we can employ a conservative static check to ensure that the effects of all operations in the program will be covered, even when using dynamic effects.

We also introduced new run-time mechanisms to manage dynamic reference sets, and detect and handle conflicts between the dynamic effects of tasks. In particular, when a task attempts to dynamically add an effect, our run-time system must be able to detect if that effect conflicts with an effect of any other running task, and prevent the two tasks from running concurrently if so. We chose to do this using a run-time mechanism that tracks the set of references added to all potentially conflicting dynamic reference sets. When a task attempts to add a reference to a dynamic reference set, our runtime system will attempt to add it to the corresponding set, atomically checking whether it is already present. If our system detects no possible conflict with another reference already in the set, then the new reference is added and execution of the task may proceed. On the other hand, if a possible conflict is detected, the task will be aborted and retried later, once the existing conflicting task is done.

We currently choose to only support tasks that add effects during an initial phase in which their access to shared data is read-only, and do not add effects once they begin performing writes. These correspond to the cautious operators described by Pingali et al [75]. This restriction means it is trivial to simply stop execution of an aborted task and re-run it later, since it cannot have modified any shared state. While our approach could be extended to non-cautious tasks, this would require support for rolling back the writes done by an aborted task, which would add overhead to our runtime system.

This work makes the following contributions:

- We define an extension to the TWE model to handle dynamically-specified effects, and implement this extension for the TWEJava language.

- We formalize the dynamic semantics of our extended TWE model, and prove that they preserve the key safety property of task isolation.

- We define a static dataflow analysis that allows us to statically check the soundness of effect specifications even in the presence of dynamic effects.
- We describe our runtime system for supporting dynamic effects in TWEJava, which is built on top of the tree scheduler described in Chapter 5.
- We evaluate our system on the Spanning Forest and Maximal Independent Set graph algorithms, showing that it can express parallel computations in which the effects of tasks can only be specified dynamically. Our evaluation shows that our system can provide self-relative parallel speedups for these algorithms, but with our current run-time system they perform worse than sequential versions of the algorithms. This is due to the overheads of the run-time system, which we are working to reduce in ongoing and future work. We also show results from a synthetic benchmark that suggests that even our current run-time system could provide speedups for algorithms in which the overheads of our run-time operations are amortized by larger amounts of computation per dynamic effect added.

7.2 Dynamic Effects

The existing TWE model and TWEJava language can handle tasks whose effects can be expressed statically. Effects may be specified in terms of run-time effect parameters, but the values of those parameters must be fully specified before the task starts running. Therefore, they cannot support tasks whose effects are based on run-time data structures and become known only as the tasks execute. These include various important algorithms, such as computations that act on a neighborhood in an irregular graph. (Note that the existing TWEJava language cannot express these sorts of algorithms while providing parallelism, because there is no general mechanism for saying that some piece of code may run with the effects of the current task plus some additional ones. Thus, even if one were to start a new task with `executeLater` whenever a dynamic computation encounters a new piece of data that it needs to access, there would be no general way to express that that new task can access both the new data and also whatever data the parent task had access to. It might be possible to support something like this using a different effect system, but I believe

it would be difficult to do so while using only static effect checking. Moreover, starting a new task every time a new piece of data that needs to be accessed is encountered would likely impose excessive run-time overheads in many cases.)

In this work, we seek to remove the TWE model's restriction to static effect specifications. To do this, we extend the TWE model and the TWEJava language to support tasks that can add certain effects dynamically as they are executing. To prevent the dynamic checks required by this mechanism from becoming overly complex, we do not allow adding completely arbitrary effects at run time. Instead, we require each task to statically specify a 'template' for the effects that will be added at run time, and we allow only effects matching that template to be added as the task executes. Specifically, the 'template' will specify a read or write effect on an RPL in which one element is a *dynamic reference set*, and the task may then dynamically add object references to that set.

7.2.1 References as regions

To support the use of dynamic reference sets, we first need to allow object references to be used in region specifications. To do this, we introduce a new type of RPL element called an *object reference RPL element*. This RPL element consists of an object reference variable name that is in scope wherever the RPL containing it occurs. To simplify our static analysis, we permit only final local variables and the special reference `this` to be used as static object reference RPL elements, so the corresponding reference may not change in the scope where the RPL is in use.

Object reference RPL elements may be used, for example, to place the fields of an object in the region `this`, as shown in the `Node` class in Figure 7.1. This means that each instance of the `Node` class will have its fields in a distinct region corresponding to that specific `Node` object. Like other RPLs, RPLs containing object reference RPL elements can also be passed as region parameters, as is done when declaring the `neighbors` array on line 2 of Figure 7.1. (That declaration has the effect of placing the cells for each `Node` object's neighbors array in that `Node` object's `this` region.) By passing such an RPL as an RPL parameter, fields of other objects may be placed in the region named after a given object. For example, a data structure internally consisting of several objects

may have all its data placed within the region named by the root object of that data structure. In general, object reference RPL elements can often be used in place of the region names used in existing TWE code.

DPJ also had a feature that permitted using object references as RPL elements in certain cases, but it worked differently from the mechanism described here, which was designed specifically for TWEJava. In particular, DPJ allowed a reference only as the first element in an RPL, and it was considered to be nested under the region given by the first region parameter of the reference's type. This mechanism was provided in DPJ to support a feature known as *subarrays*, but these restrictions and special semantics would not be appropriate for the way we wish to use references as regions in TWEJava. In our approach, references work more similarly to named RPL elements, with the main difference being simply that their disjointness is determined based on the run-time values of the references.

7.2.2 Dynamic reference sets

By using references as regions, we can support dynamic effects in which the only elements that can be added dynamically are object references. This is desirable, because it makes the checks we will need to perform when effects are added dynamically considerably simpler and faster than they would be if we allowed the full range of TWE effects to be added while a task is running. Essentially, it reduces the dynamic check we will need to do to testing whether an object reference is already in a set and (atomically) adding it if it is not.

We allow the effect specifications for our tasks to contain a new kind of element in their RPLs, which we call a *dynamic reference set* element. These will play the role of the template effects discussed above. Like other effects in TWE, a dynamic reference set effect can be specified as either a read or write effect. In either case, it does not have a fixed static definition. Instead, the RPL that the effect operates on is specified as a template in which one element corresponds to an initially-empty set of object references that can be added to as the task runs. These special RPL elements are specified with the syntax $\ldots name$. After an element is added to the set at run time, the task may subsequently access data in the region specified by substituting that reference into

```
1  static class Node {
2    Node[]<this> neighbors in this;
3    boolean inMIS in this = false;
4    ...
5  }
6
7  class Check extends Msg1Arg<Node,
8  reads ...rdNodes writes ...wrNodes> {
9    public void process(Node v) {
10     rdNodes.addAccess(v);
11     for (Node u : v.neighbors) {
12       rdNodes.addAccess(u);
13       if (u.inMIS) return;
14     }
15     wrNodes.addAccess(v);
16     // Writes used below here
17     // No more effect additions allowed
18     v.inMIS = true;
19   }
20 }
```

Figure 7.1: Task for computing a maximal independent set of a graph, using tasks with dynamic effects. This task will be invoked once for each node in the graph.

the corresponding position in the RPL.

Figure 7.1 shows an example of using our system to compute a maximal independent set of a graph. The task shown can be run in parallel on all the nodes of the graph to compute the MIS. The task for each node is shown as having effects `reads ...rdNodes writes ...wrNodes`. This specifies that it will have dynamic effects on two sets, a set of object reference regions that may be read or written (`wrNodes`), and a separate set of those that can only be read (`rdNodes`). These extend the existing syntax of read and write effects in TWEJava, with the `...`*name* syntax declaring them as effects on dynamic reference sets.

7.2.3 Adding elements to dynamic reference sets

In a task whose statically-declared effects contain dynamic reference sets, references may be added to those sets dynamically to enable new effects as the task executes. To do this, the programmer uses the *addAccess* operation, with syntax *name*`.addAccess(`$reference$`)` to add a new object reference

to the set. If the task successfully continues executing beyond the *addAccess* operation, then it was successful, and in the remainder of the task it may perform operations that have effects covered by substituting the object reference *reference* for the dynamic reference set *name* in the effect containing it.

In our example, the `Check` task performs *addAccess* operations on lines 10, 12, and 15. The first will permit read access to the node `v` which is being processed. This allows `v.neighbors`, which is in the region `v`, to be read in the following line. The second will permit read accesses to a node `u` which is a neighbor of the node `v`. This is taken advantage of in the following line to read `u.inMIS`, which is in the region `u`. Since in each of these cases the *addAccess* operation is a control-flow dominator of the subsequent read, our compiler can statically verify that the effect of that read is covered by the effects of the task at the point where it occurs. Similarly, the write to `v.inMIS` in line 18 can be verified to be safe based on the *addAccess* operation on line 15. We discuss the full details of the static analysis used for these checks in Section 7.2.6.

7.2.4 Aborting and retrying tasks

An *addAccess* operation may fail if another running task already has a conflicting dynamic effect. In this case, it is generally necessary to abort the task performing the operation and retry it later. Our runtime system will do this, waiting until the conflicting task has completed before retrying the aborted task. (In some cases it would be possible to simply pause the task, but in general that could lead to deadlock; we do not currently implement this.)

To simplify this abort and retry process and reduce overheads, we adopt the rule that a task must consist of an initial phase in which it performs no writes to shared state (and launches no other tasks) but may perform *addAccess* operations, and a later phase in which it may perform writes but its effects remain fixed. Since a task may be aborted only during the first phase (when it attempts an *addAccess* operation), this rule means it may simply be stopped and later retried; it is not necessary to roll back any writes it may have performed when aborting it. This is a significant restriction, but many useful computations are naturally expressed in this form, which corresponds to the cautious operators discussed by Pingali et al [75]. Our system could be extended to remove

this restriction, but it would come at the cost of higher overheads to maintain data to support rollback. Note that the example in Figure 7.1 follows this restriction, with the only write to shared data on line 18 following the last *addAccess* operation on line 15.

7.2.5 Integration with the TWE region and effect system

For simplicity, the example in Figure 7.1 shows a case in which the only effects of a task are dynamic, and the dynamic reference set element is the only element of the RPLs used for those effects. In reality, however, the support for dynamic effects in this work fully integrates with the TWE system's existing support for highly-expressive region and effect specifications, including effects on regions that form a hierarchical structure and may contain wildcards. The only restrictions we impose are that only one dynamic reference set element may appear in a single effect specification, and it must appear to the left of any wildcard elements in its RPL. (These restrictions simplify our implementation, as discussed in Section 7.5).

7.2.6 Dataflow analysis

Like in the original TWE system, we rely on a static analysis to determine whether the *current covering effect* applicable to any operation in the program will in fact cover the effects of that operation. If it cannot be statically determined to do so, the compiler will flag a compile-time error. With the addition of dynamic effects, this analysis becomes more complex, because the covering effect can change at run time as elements are added to dynamic reference sets. However, we still rely on a conservative static analysis of covering effects to determine this.

To support dynamic effects in this analysis, we augment it with an intraprocedural dataflow analysis which will compute for each dynamic reference set S a set of variables $inSet_S$ whose values are definitely known to be in that dynamic reference set at each point in the program.

At the point of an *addAccess* operation S`.addAccess(`v`)`, our dataflow analysis will add the variable v to $inSet_S$. A variable is also added to $inSet_S$ if it is assigned from a variable that is already in the set; it is removed from $inSet_S$ if assigned from any other expression. At control flow join points, $inSet_S$ is computed as the intersection of the $inSet_S$ sets on the incoming edges.

These rules ensure that $inSet_S$ will always be a conservative approximation of the set of variables whose run-time reference values will be in the dynamic reference set S at each program point. Thus, if $v \in inSet_S$ at the point of an operation whose effect would be covered by the effect on S if v were substituted for S in that effect, then our static effect analysis may safely conclude that effect is covered, and will not flag it as an error.

This dataflow analysis (conceptually conducted separately for each dynamic reference set S, although it is possible to analyze them together in the actual implementation) is a forwards analysis with $\cap$ as the meet operator. Like many other dataflow analyses, it has transfer functions of the general form $f_B(x) = gen_B \cup (x - kill_B)$. It is fairly straightforward to see that this is a monotone dataflow framework, and as such could be solved by an iterative algorithm. In practice, however, we solve it with a structure-based algorithm similar to the one discussed in Section 4.4; as in that case, we chose this algorithm because of its ease of implementation within the compiler framework we are using.

7.2.7 Asserting that a reference is in a set

The dataflow analysis described above can be used in many cases to determine that a variable's value would be in a dynamic reference set S, but in some cases the conservative analysis may not be able to determine this, even though it will actually always be the case. Most importantly, since the dataflow analysis described above is purely intraprocedural, it cannot be used to check that a variable is in a dynamic reference set if its reference value was added to that set in another method.

To address this shortcoming, we add operation *assertContains*, which can be used by the programmer to assert that the value of a specified reference value is in fact in a certain dynamic reference set. This assertion will be discharged at run time by verifying that the actual run-time value of the variable is in fact in the set. If it is not, it will throw an exception.

Our dataflow analysis is augmented so that following an *assertContains* operation S`.assertContains(`v`)`, the variable v will be added to $inSet(S)$, allowing it to then check that subsequent operations whose effects involve v may be covered by the dynamic effect on S.

This mechanism provides a way to handle and check cases where our basic dataflow analysis

alone cannot verify that such effects are safe, e.g. because of complex control or data flow, or because effects on the reference occur in a different method from the one that adds it to the set. The use of these assertions involves a run-time overhead, but the cost of each check is fairly small, and it is usually sufficient to use only one assertion for a variable v in each major computation phase or method where effects on it are used.

7.3 Dynamic Semantics and Safety Properties

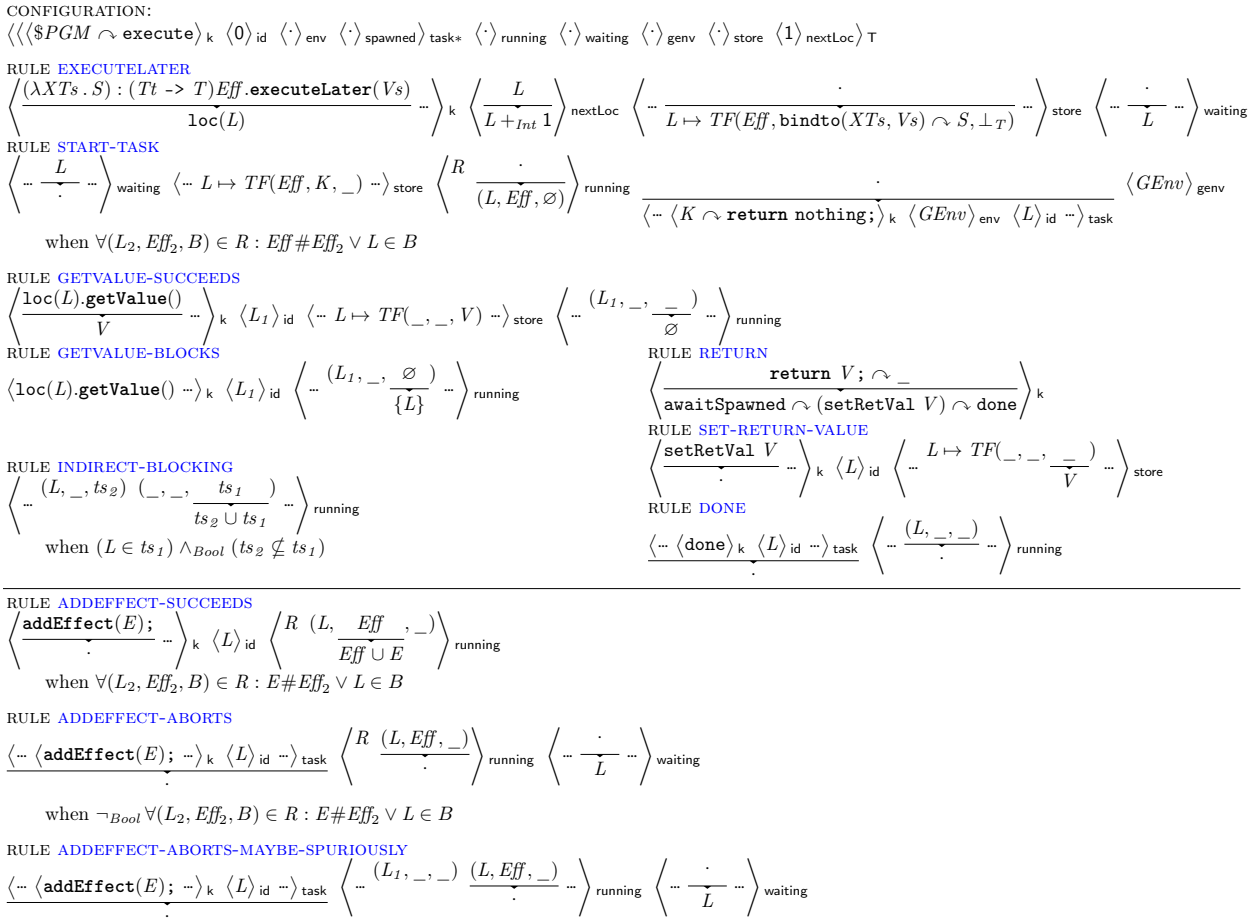


Figure 7.2: Dynamic semantics for key TWE operations. For simplicity, we omit rules related to the `spawn` and `join` operations for statically-checkable safe parallelism, which are largely orthogonal to our work on dynamic effects. The rules above the horizontal line are taken without modification from the original dynamic semantics of TWE (without dynamic effects) as given in Chapter 3. The rules below the line are added in this work to support dynamic effects.

In this section, we present the dynamic semantics of tasks with effects, including our extensions

to support dynamic effects, and discuss how these semantics guarantee the key safety properties of tasks with effects.

7.3.1 Dynamic semantics

Figure 7.2 shows the dynamic semantics, which are written using the K semantic framework [78]. These semantics are based on the original semantics for tasks with effects presented in Chapter 3, but are extended with additional rules to support our addition of dynamic effects. For simplicity, we omit this rules related to TWE's `spawn` and `join` mechanisms for statically-checkable safe parallelism, which are largely orthogonal to the dynamic effects mechanism presented in this work (although the two can be safely used together in one program). We also omit all non-TWE-related rules (e.g. for arithmetic operations).

These rules are based on rewriting logic. Each rule may be applied when it can match the configuration elements shown at the top of it, and when it is applied any elements with a horizontal line under them will be replaced by what is below the line. The top of the figure shows the format for the configuration that is operated on, which consists of several nested cells. This includes the *task* cell (with several subcells), which captures the state of a running task, as well as the *running* cell which contains a set of tuples corresponding to running tasks and containing the information necessary to check for effect conflicts between tasks. Each of these tuples consists of a unique location value that identifies a task, as well as its current effect set and the set of tasks that it is blocked on (if any). The *waiting* cell contains a set of location values for tasks that are waiting to begin executing; they may or may not have effects that conflict with another running task. For full details about the remaining configuration cells, see Section 3.2.

The new rules we added to support dynamic effects are those below the horizontal line. They all relate to the `addEffect` operation, which is used to dynamically add a new effect to a running task. While our actual language design and implementation discussed in Section 7.2 allow only effects following certain templates to be added at run time, the semantics presented here are more general and can support the dynamic addition of arbitrary effects.

The ADDEFFECT-SUCCEEDS rule covers the case where no conflict with another running task

is found when doing an `addEffect` operation, and thus that operation succeeds. In this case, the new effect E is added to the task L's entry in *running*, and the task continues to execute. This condition on when this rule may execute ensures it can be run only if the task L with the new effect added will in fact be non-conflicting with all other running tasks.

The ADDEFFECT-ABORTS covers aborting a task L when it tries to do an `addEffect` operation that would cause an effect conflict between L and some other running task. In this case, the current execution of task L is aborted (deleting the *task* cell for it), and L is returned to the set of waiting tasks, where it may be retried later. This rule does not show any rollback of state updated by L, because we assume that `addEffect` operations are performed only in an initial read-only portion of a task. This property can be statically checked, ensuring that tasks do not violate it.

The ADDEFFECT-ABORTS-MAYBE-SPURIOUSLY rule also covers aborting a task, but unlike the ADDEFFECT-ABORTS rule, it may also abort a task trying to do an `addEffect` operation even if that operation would not actually create a conflict with any other task. This reflects the semantics of some possible implementations of dynamic effects which may spuriously detect a possible conflict when there really is none, and abort based on that. (Our actual implementation, as discussed in Section 7.5, may do this because of simplifications in how it represents dynamic effects and checks for conflicts between them.) This rule does require that there be at least one other running task in addition to the task that will be aborted; this ensures that we do not abort the only running task, which could prevent the system from making progress.

7.3.2 Safety properties

Our semantics guarantee the core safety properties of TWE, including task isolation, data race freedom and strong atomicity for portions of tasks that do not perform TWE operations. These properties all stem from the fundamental property of task isolation, i.e. that two concurrently executing tasks may not have conflicting effects. In the following section, we state this property formally and prove that our semantics guarantee it.

7.4 Proof of Task Isolation

In this section, we prove that the dynamic semantics of tasks with effects given in this chapter (Figure 7.2), including our additions to support dynamic effects, guarantee the fundamental property of task isolation, i.e. that two conflicting tasks may not be executing concurrently. (We define this property formally below.)

7.4.1 Effect system properties

We require the effect system to obey the following simple property, that the union of two effect sets is disjoint from any effect set that is disjoint with both of the constituent sets. Our actual effect system obeys this property, as do many others.

Property 1. $E_1 \# E_3 \wedge E_2 \# E_3 \implies E_1 \cup E_2 \# E_3$.

Note that # is used to denote that two effect sets are non-interfering.

7.4.2 Proof of Task Isolation

We first define what it means for two tasks to be non-conflicting. In general, this is the case if their effects are non-conflicting, but it can also be the case because of *effect transfer* in cases where one task is blocked on the other.

Definition 4 (Non-conflicting tasks)**.** *We say two tasks denoted by tuples* (L_1, E_1, B_1) *and* (L_2, E_2, B_2) *are* non-conflicting*, denoted* $(L_1, E_1, B_1) \# (L_2, E_2, B_2)$*, iff* $E_1 \# E_2 \vee L_1 \in B_2 \vee L_2 \in B_1$.

For simplicity, we sometimes simply write $L_1 \# L_2$, meaning that the tuples in *running* with location values L_1 and L_2 are non-conflicting. Our rules ensure that at any point *running* may contain at most one tuple with any given location value L, so this is unambiguous.

We now prove that the TWE semantics guarantee task isolation, that is, that no two concurrently executing tasks may be conflicting. Specifically, we show that each operation in the dynamic semantics preserves the following invariant:

Invariant 1 (Task isolation)**.** $\forall(L_1, E_1, B_1), (L_2, E_2, B_2) \in \textit{running}$, *where* $L_1 \neq L_2$ *:* $(L_1, E_1, B_1)\#(L_2, E_2, B_2)$.

The task isolation invariant trivially holds at the beginning of a program, because only one main task is initially running. We will proceed to show that all the operations in the TWE system maintain this invariant.

Lemma 9. *Rules that do not change running preserve Invariant 1.*

Proof. This is clearly true, because Invariant 1 concerns only the contents of *running*, which will not be changed by such operations. □

This covers the EXECUTELATER, RETURN, and SET-RETURN-VALUE rules in Figure 7.2, as well as the rules for all non-TWE-related operations in the program (e.g. arithmetic operations), which are omitted from Figure 7.2.

Lemma 10. *The* GETVALUE-BLOCKS *and* INDIRECT-BLOCKING *rules preserve Invariant 1.*

Proof. These rules only expand the set of tasks that some task is recorded as being blocked on, and make no other changes to *running*. If Invariant 1 holds before the application of one of these rules, it will still hold afterward, because the only change possibly relevant to the invariant would be to expand B_1 or B_2, which will not invalidate the invariant when it was previously true. □

Lemma 11. *The* DONE, ADDEFFECT-ABORTS, *and* ADDEFFECT-ABORTS-MAYBE-SPURIOUSLY *rules preserve Invariant 1.*

Proof. These rules remove the tuple for a task from *running*, but make no other changes to it. If the invariant held for every pair of task tuples in *running* prior to applying one of these rules, it will still hold afterward. □

Lemma 12. *The* START-TASK *rule preserves Invariant 1.*

Proof. This is the case because of the condition on when the START-TASK rule can execute. Specifically, it can only execute (adding a new tuple $(L, \mathit{Eff}, \varnothing)$ to *running*) if $\forall(L_2, \mathit{Eff}_2, B) \in R$: $\mathit{Eff}\#\mathit{Eff}_2 \vee L \in B$, where R is the value of *running* immediately before the rule executes. This

condition ensures that the newly added task tuple will be non-conflicting with any existing task tuple in *running*. If Invariant 1 held prior to the application of the START-TASK rule, this will ensure that it still holds afterward. □

Lemma 13. *The* ADDEFFECT-SUCCEEDS *rule preserves Invariant 1.*

Proof. This rule also preserves the invariant because of the condition on when it executes. Specifically, it changes the effect specified in *running* for an existing task L from Eff to $\mathit{Eff} \cup E$, with the required condition that $\forall(L_2, \mathit{Eff}_2, B) \in R : E \# \mathit{Eff}_2 \vee L \in B$, where R consists of the existing contents of *running* with the tuple for L removed. That is, for all $(L_2, \mathit{Eff}_2, B) \in$ *running* where $L_2 \neq L$, either $L \in B$ or $E \# \mathit{Eff}_2$. For values of L_2 where the former case applies, we will have $L \# L_2$ regardless of what the effects of L are set to. For values of L_2 where only the latter case applies, we know that $E \# \mathit{Eff}_2$, but in order to have $L \# L_2$ we will need to show that $\mathit{Eff} \cup E \# \mathit{Eff}_2$. By assumption, Invariant 1 held prior to the application of the ADDEFFECT-SUCCEEDS rule, and since in this case we know $L \notin B$, we must have $\mathit{Eff} \# \mathit{Eff}_2$. By Property 1, we therefore must have $\mathit{Eff} \cup E \# \mathit{Eff}_2$, and so $(L, \mathit{Eff} \cup E, -) \# L_2$. Thus, $(L, \mathit{Eff} \cup E, -)$ is non-conflicting with the task tuples for all other tasks L_2 in *running*. Since we know those other task tuples are all mutually non-conflicting based on the assumption that Invariant 1 holds initially, we may conclude that after the application of the ADDEFFECT-SUCCEEDS rule the invariant will continue to hold. □

Theorem 4. *Invariant 1 holds throughout the execution of a TWE program according to the semantics in Figure 7.2.*

Proof. The invariant holds initially at the start of the program because only the initial main task is running, and therefore Invariant 1 is trivially satisfied. By Lemmas 9–13, each of the rules shown in Figure 7.2 preserves the invariant, as do other rules (not shown) for non-TWE-related operations. Thus, any TWE program containing only these operations will preserve the invariant. □

While we do not show them here because they are orthogonal to our new rules for dynamic effects, the semantic rules related to TWE's `spawn` and `join` operations for statically checkable safe parallelism can be formulated to preserve the task isolation invariant as well, so in fact it is preserved in all TWE programs.

7.5 Runtime Implementation

To support our dynamic effect system, we must augment the TWE runtime system with new functionality to handle dynamic effects, while preserving TWE's fundamental guarantee that tasks with conflicting effects will not be allowed to execute concurrently. The first type of new functionality is to ensure that dynamic effects will not conflict with the statically-specified effects of other tasks. This can be done as a straightforward extension of the scalable, tree-based scheduling algorithm already used for TWEJava [49].

The second, more significant type of new functionality is to actually support the dynamic addition of effects to tasks. The runtime system must record those effects, check whether they conflict with other dynamic effects, and abort and later retry tasks if a conflict is found. We take advantage of TWE's hierarchical effect system in combination with the restrictions that we place on dynamic effects to minimize the cost of these run-time conflict checks.

In particular, it is possible to implement them such that the runtime conflict check needs to consist only of checking if the reference to be added is in a certain set data structure, held at a certain node of the RPL tree, and either adding it if it was not already present or aborting the task if it was. Several variations of this basic approach are possible, generally trading off speed and perhaps storage space against the ability to accurately detect that dynamic effects are non-conflicting in as many cases as possible. For example, one simple approach would simply maintain one set of references, adding to it whenever a dynamic effect is added and conservatively detecting a conflict whenever the reference to be added duplicates one already in the set. This is straightforward and fairly quick, but it may detect a possible conflict in cases where there really isn't one, either because both effects are reads or because the two effects could be determined to be disjoint based on the portions of their RPLs beyond the dynamic reference set element.

A slightly more refined approach is to distinguish reads and writes but still not take into account the RPL tail beyond the dynamic reference set element. This is what our implementation currently does, and this approach is described in detail below. Another possibility is to use a probabilistic data structure such as a Bloom filter, which uses a fixed amount of space but may detect a conflict when there really isn't one because of its probabilistic approach to representing a set. Although

we only describe our current method in detail below, we have designed our runtime system so that different conflict checking methods can be 'plugged in' in to. We believe that experimenting with data structures and approaches for the runtime conflict checks can be a fruitful area of research, and different choices might be best for different types of programs.

7.5.1 Integration with the tree-based TWE scheduler

The first major area of functionality we added to our run-time system involves integrating with the existing tree-based scheduler used for effect-based scheduling of TWEJava tasks [49]. This scheduler uses a tree data structure that corresponds to the hierarchical structure of RPLs. Each node in the tree is labeled by a non-wildcard RPL element, and the path from the root of the tree to that node corresponds to a wildcard-free RPL prefix. All effects which begin with that prefix will be checked against the other effects of running tasks at that node, and the effects of executing tasks will be held at the node corresponding to the maximal wildcard-free prefix of their RPLs. (Some effects containing wildcards are also checked against other effects in other nodes below the node they are held at.)

Dynamic effect templates can be straightforwardly integrated into the tree-based scheduler. They are like other effects except that one of their RPL elements is a dynamic reference set element. We conservatively treat this as conflicting with all non-dynamic object reference RPL elements (and with wildcards that could cover such an element). While this may spuriously prevent parallel execution in cases where it would actually be safe, this approach allows us to ensure that dynamic effects will not conflict with non-dynamic effects without introducing major additional complexity in our scheduler, and without introducing extra overheads each time a reference is added to a dynamic reference set.

Similar to non-dynamic effects, our scheduler will move dynamic effect templates down until they reach the node corresponding to the RPL prefix preceding the dynamic reference set element (which is required to be wildcard-free). Also like other effects, dynamic effect templates will be *enabled* once they are found not to conflict with any other non-dynamic effects of running tasks, and a task will be enabled only once all its effects and dynamic effect templates are enabled.

The key new feature that we added to the tree-based scheduler to support dynamic effects is that at each node it maintains a data structure to manage the dynamic effects at that node. By the time a task with dynamic effects has been enabled, its dynamic effect templates will all have been placed at the nodes in the scheduler tree corresponding to the RPL prefix preceding the dynamic reference set element in their RPLs.

Note that *a conflict between the dynamic effects of two concurrently-executing tasks can arise only between two effects whose templates have the same RPL prefix and are therefore at the same node.* This is the case because two dynamic effects with different prefixes either can be statically proven to be non-conflicting based on their prefixes, without regard to the actual dynamic component, or else will be conservatively treated as always conflicting without examining their dynamic component. The latter case can arise only when at least one of the effects has a wildcard after the dynamic component in its RPL. In this case, since the wildcard is statically present in its effect template, we apply the conservative analysis described above and treat the two effect templates as always conflicting, so the corresponding tasks will not be permitted to run concurrently.

Because of the above property, it will suffice when checking dynamic effects to do so only using a data structure associated with the scheduling tree node where the dynamic effect template resides.

7.5.2 Recording and checking individual dynamic effects

When a task performs an *addEffect* operation, it must attempt to add the specified object reference to the specified dynamic effect set, and check whether it conflicts with any other dynamic effects of concurrently-executing tasks. As discussed above, it is only necessary to do this check among dynamic effects whose templates are located at the same node of the scheduler tree, so we can use a data structure associated with an individual node of the scheduler tree to manage it. We say that those dynamic effects are 'at' the corresponding scheduler node.

As discussed above, several approaches are possible, but we currently use one where dynamic effects at a given scheduler node are distinguished based on whether they are reads or writes: multiple read effects using the same reference at the same node can be allowed, but only one write effect for each reference is possible, and it will exclude any simultaneous reads. As mentioned, for

simplicity and speed, we do not take into account the 'tails' of effects, i.e. the portions beyond the dynamic reference element. In general, this will result in some effects being detected as conflicting when they need not be, but in the algorithms we have examined so far, it is possible (and in fact natural) to structure them so that this is not the case, e.g. by not using any additional RPL elements beyond the dynamic reference element.

To support differentiating read and write effects, we maintain separate maps from object references to tasks, one for reads and one for writes. The map for reads is a multi-map, in which one reference can map to multiple tasks.

When a task attempts to add a new reference to a dynamic reference set associated with a write effect, we first perform an atomic *putIfAbsent* operation to attempt to add it to the write map. If this fails, it indicates that another task has a conflicting write effect. We can check if that task is done, in which case we can remove its entry in the map and retry; otherwise, the task adding the new effect must be aborted. If the addition of a new mapping to the write map succeeds, we then look up the mapping for the corresponding task in the read mapping and see if there are any currently-executing tasks with conflicting read effects. (Once again, we may encounter mappings for completed tasks, which can be discarded.) If there are, we must abort the task adding the new effect.

When adding a new reference to a dynamic reference set associated with a read effect, we perform a similar procedure, but in this case we first add its mapping to the read set. We will not identify any conflicts at this stage, since it is legal to have multiple read effects using the same dynamic reference. After inserting the reference in the read set, we proceed to check if there is an entry in the write set for the same reference, and if so whether it is associated with a still-running task. If so, we must abort the task adding the new effect; otherwise, it may proceed.

This procedure requires synchronization at the level of individual entries in the mappings, but operations involving different object references can proceed concurrently. This relatively fine-grain synchronization, in combination with our ordering of operations for writes and reads, ensures that any conflict with an effect of another concurrently-running task will be detected. Our approach to recording dynamic effects and checking for conflicts does impose overheads which must be amortized

in order to achieve reasonable performance. As discussed in the evaluation section, the benchmarks we currently have do not perform sufficient computation to amortize these overheads, so they perform poorly with our system. We believe our system could perform better on some other programs, but in general its overheads are a serious limitation. We are currently exploring lower-overhead approaches for performing these checks, which we discuss briefly later.

7.6 Evaluation

We performed an initial evaluation of our implementation using programs that implement two basic graph algorithms, Maximal Independent Set (MIS) and Spanning Forest (SF). In both cases, the basic parallel algorithm we adopted for computing these properties is the same one used in the versions of these programs in the PBBS benchmarks [19].

7.6.1 Expressiveness

We were able to express both MIS and Spanning Forest using our system, structuring each of them using tasks with dynamic effects only on the portion of the graph that the individual task needs to access.

For MIS, we use a straightforward adjacency graph representation with a separate object for each node, and maintains a flag for each node indicating whether it has been assigned to the MIS (initially false for all nodes.) Our algorithm involves a parallel loop over all the nodes of a graph. For each node v, it checks whether any of its neighbors are already assigned to the MIS, and if not it flags that node as belonging to the MIS. The step for each node must behave atomically; it would be an error if any of the neighbor nodes were placed in the MIS after they were checked when considering v but before the flag for v is set. The step for each node could conceptually be a task, but in our actual implementation we coarsen this to operate on several nodes in each task in order to reduce overheads.

This algorithmic structure could not be specified in the original version of TWE without dynamic effects, because both the number and the identities of the neighbor nodes are only known at run time. Because of this, it would not be possible to statically specify the effects of the task

operating on each node, unless the data for all the nodes were simply placed in a single region (which would destroy the opportunity for parallelism.)

Our extension to support dynamic effects lets us express this algorithm, with parallelism. As the task actually runs, it can access the run-time data structures specifying the neighbors of a node, and add a read effect for each neighboring node, which will enable it to read that node's in-MIS flag. Similarly, it adds a write effect for v before writing its in-MIS flag. This ensures that there can be no data race on the in-MIS flags, and so the algorithm works properly.

For Spanning Forest, we use an edge graph representation in which both nodes and edges in the graph are represented by edges. Each node has a unique index number (assigned when constructing the graph) as well as an ancestor reference (to a lower-numbered node in the same connected component, initially null). Each node has a flag indicating if it is in the spanning forest. Our algorithm involves a parallel loop over the edges of a graph. For each edge e, we examine the two nodes u and v incident on that edge, and follow the chain of ancestor links to find the lowest-numbered node in the same connected component. If they are not the same (indicating that u and v are not already in the same connected component), then we flag e as being in the spanning forest and update the ancestor link of the lowest-numbered node in one of the components to point to the other one. Once this process has been done for all the edges, the spanning forest will have been constructed.

To run this algorithm in parallel, we must provide synchronization for the ancestor references accessed during the operation for each node; the ancestor references it accesses may not be updated by a concurrent operation while it is in process. As with MIS, our implementation coarsens the tasks, processing several edges together in one task, but conceptually the operation for each edge could be its own task. The task will first do a read-only traversal of the ancestor links of u and v, always adding a read effect for each node before reading its ancestor link. Once it has determined what link needs to be updated (if any), it will add a write effect for that node. In the subsequent read-write portion of the task, it will actually perform that update.

The data access pattern of Spanning Forest tasks is even more complicated than that of MIS, and it also could not be specified by static effects in the original TWE system (while preserving

parallelism). Our support for dynamic effects, however, allows us to support this algorithm, because as the task runs and discovers what data it needs to access based on the run-time graph data structures, it can add appropriate effects dynamically before doing the corresponding memory accesses. Thus, our system enables this parallel algorithm to be expressed, while the original TWE system or others with only static effect specifications would not.

7.6.2 Performance

We performed an initial evaluation of the performance of these algorithms using our current runtime implementation. They were evaluated on a 40-core (80-thread) system using four Intel Xeon E7-4860 processors, using Oracle JDK 8u74. Figures 7.3 and 7.4 show the performance of these two benchmarks in our tests.

Both benchmarks show at least some level of self-relative scalability. MIS shows a maximum self-relative speedup of 6.95x on 60 threads, although scaling is poor beyond 8 threads. (Due to a known interaction between the code and our current runtime system, one thread is not effectively utilized in multi-threaded configurations, explaining the lack of scaling from one to two threads.) Spanning Forest has poorer self-relative speedups, showing very little improvement beyond three threads and having a maximum self-relative speedup of about 2x.

Unfortunately, even the fastest times we observed with both of these codes are significantly slower than the performance of an alternative sequential implementation. Accordingly, with our current implementation, our system does not represent a viable approach to obtaining parallel performance improvements for these codes.

There are several types of performance overhead that impact these codes. A major one, and probably the most fundamental one for our system, is simply that MIS and SF tasks do very little computational work *other than* traversing the portion of the graph accessed in the task. Since we need to do an additional run-time check for each node accessed (which in our current implementation involves operations on two hash map data structures, and some auxiliary operations), the cost of these run-time checks can easily become a dominant performance factor. While the individual checks are not terribly expensive, they still consume more time than anything else the tasks are

doing.

Another factor contributing to performance overheads is the cost of setting up tasks, involving creating the task object and scheduling it using our tree scheduler. We try to ameliorate this by coarsening, with one TWE-level task aggregating the basic operations on a number of nodes or edges, but it is still a factor in performance overheads. Another factor, which is exacerbated by coarsening, is that tasks may be aborted, causing work to be lost and subsequently redone. There are generally few aborts for MIS, but there are more for SF. This relates to the fact that the nodes accessed in SF are not generally highly localized, and particularly in the middle and later stages of the algorithm, a number of nodes may share the same ancestors, leading to contention and resultant aborts.

Another factor in performance is the data representation. Because our system for dynamic effects is based on object references, it is necessary to have a Java object for each node. But without that requirement, it would be possible to represent the graph data using arrays and refer to nodes by their integer index numbers in the arrays, which could avoid many of the overheads of creating and managing Java objects. This is what the sequential versions of the computations do.

As a preliminary experiment to illuminate how our system might perform on other programs that do significant computation beyond simply traversing a graph, we also tried creating a modified version of MIS that performs extra synthetic work in addition to the actual MIS computation. (When computing whether each node is in the MIS, it also does 10,000 steps of a loop whose body just performs an integer multiplication and subtraction.) The results for this benchmark are shown in Figure 7.5. The difference between the times in this figure and corresponding times in Figure 7.3 can be seen as reflecting the extra work performed. This synthetic benchmark gives much better self-relative speedups, up to 17x on 80 threads. While this is a synthetic benchmark, we do believe it reflects that even the current version of our system can perform better, and maybe offer useful speedups, on programs that do more computation to amortize the costs of our checks.

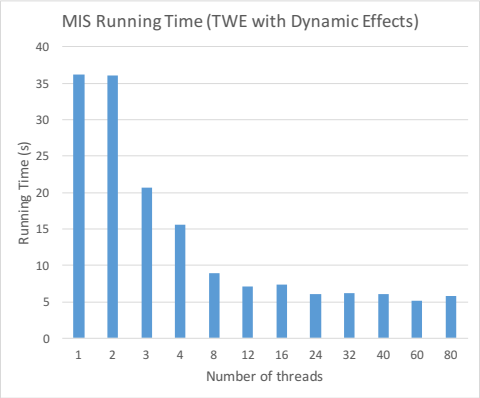


Figure 7.3: MIS benchmark performance, using a random 10,000,000-node graph

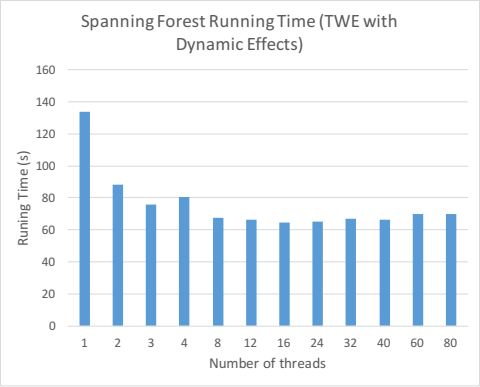


Figure 7.4: Spanning Forest benchmark performance, using a random 1,000,000-node graph

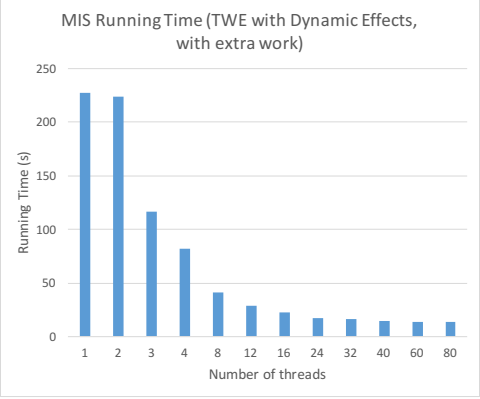


Figure 7.5: Performance of MIS benchmark, modified to do extra work in each step (10,000 steps of a loop whose body just performs an integer multiplication and subtraction) in addition to the actual MIS computation.

7.7 Ongoing and Future Work

Improving the performance of the dynamic effects system is a subject of ongoing work. I have several ideas about how to do this. One is to use very low-overhead data structures and computations for the run-time dynamic effect checks. This could involve using a field of each object to track if it is being used as a reference for a dynamic effect, or using alternative data structures such as Bloom filters (with a small and fast hash function, e.g. the native one provided by Java). Another possibility is to allow the use of integers in dynamic effects in addition to or instead of object references, which could permit more efficient array-based graph representations to be used.

Another fruitful direction for future work would be to automatically aggregate small tasks specified by the programmer into larger tasks to be processed by our run-time system. This could make programs easier to write while also reducing task creation overheads. In addition, it would be possible to abort only the current programmer-specified task, rather than the full aggregated task, when an effect conflict is detected.

I believe that by pursuing these directions, it will be possible to produce a system that can support programs using dynamic effects while providing useful parallel speedups.

Chapter 8

Related Work

8.1 Structured parallel programming models

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. Several parallel programming systems have been developed that are more structured and easier to reason about than traditional threads. These include OpenMP [71], Cilk [20], Threading Building Blocks (TBB) [52] (except for the "lower-level" task interfaces), and Java's `ForkJoinTask` [72]. OpenMP is commonly used for computations based on parallel loops, but recent versions also support task-based parallelism. Cilk uses structured parallelism based on nested fork and join operations. TBB offers both parallel loops and task-based mechanisms. `ForkJoinTask` is focused on nested fork-join computations, although it also supports other task invocation and dependence structures. Similar mechanisms are also employed in commercial systems such as Microsoft's Task Parallel Library [68] and Apple's Grand Central Dispatch [10].

These systems provide a somewhat higher-level structure for parallel computations than directly using threads and low-level synchronization operations, and that can certainly make it easier to program using them. However, they still do not provide any checked 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. Systems that limit programs to use a particular parallelism structure (e.g. fork-join) can simplify this reasoning, but in doing so, they make it impossible to 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.

8.2 Programming models with parallel safety guarantees

Many parallel and concurrent programming systems provide various correctness guarantees but have weaker expressive power than TWEJava. Several of these systems, including Jade [77], Prometheus [8], OoOJava [54], Dynamic Out-of-Order Java (DOJ) [36], Regent [80], Pāñini[61], and Ke et al.'s system for parallelization with dependence hints [58], 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 [73] 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, DOJ, SMPSs, SvS [18], Legion [14, 82], and Aida [62], 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. (We compare effect-based run-time scheduling systems to ours in Section 8.4; here we focus on the expressiveness and parallel safety properties of these languages.)

DPJ [22, 23], Legion and SvS can express deterministic and 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. Also, Legion can provide strong correctness guarantees only for programs that essentially have sequential-equivalent semantics, except that certain operations may be commutative. This is a more restrictive programming model that TWEJava or even DPJ. (Legion also provides modes which are more flexible but provide few guarantees about access to shared data, so the programmer is responsible for managing it correctly, as in traditional multi-threaded programs.) Legion has a hierarchical region system with some valuable features, including allowing more dynamic assignment of data to regions, and supporting explicit program management of locality via region maps. Legion's run-time scheduling approach is discussed below, in Section 8.4.

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.

JOE [32] and MOJO [29] define sophisticated ownership-based effect systems with hierarchical regions. These can be used to provide deterministic semantics for object-oriented programs with hierarchical data structures, although these works do not define a full parallel programming model. They are designed for use in static type checking, in which case the use of purely static checks would impose expressiveness limitations similar to DPJ. Also, these systems have limited expressiveness for parallelism over arrays, which is crucial for many parallel algorithms.

CoreDet [16], Kendo [70], Grace [17], and DMP [33] 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.

Aviram et al. [13], Burckhardt et al. [27], and Hower and Hill [51] propose models for deterministic parallelism in which concurrent threads of execution essentially have private copies of shared data which may be updated independently, but these are merged deterministically at join points. These models are restricted to fork-join computations, and cannot support the more general parallelism patterns that TWE can. Moreover, the data duplication and copying that they require may involve significant overheads.

A number of systems provide safety guarantees such as data race freedom, but do not address stronger properties such as determinism. Several papers [25, 2, 53, 9] provide language mechanisms for enforcing a locking discipline in nondeterministic programs, to prevent data races and deadlocks. These systems generally support an expressive range of programs but they do not aim to guarantee stronger, e.g., deterministic, semantics. RCCJava [38] can ensure data race freedom, but it does not provide structured concurrency constructs or guarantee other safety properties such as determinism. SharC [9] 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.

8.3 Actors

One significant style of concurrent programming is *actors* [6]. 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 [79, 57, 67] 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. The TWE model, 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.

8.4 Systems using effect-related run-time task scheduling

Several systems use some type of effects for run-time task scheduling. Many are limited to some form of structured parallelism, and most do not support a rich effect system like ours. Jade [77], Pāñini [61], and Dynamic Out-of-Order Java [36] all express effects of tasks directly in terms of accesses to memory objects or locations. Concurrent Collections [28], CellSS [15], SMPSs [73], StarPU [12], and the system of Vandierendonck et al. [85] use a task dataflow model and express the dependencies of tasks in terms of memory objects, or with explicit task-to-task dependencies. SMPSs [73] allows parallelization of sequential codes by having the programmer specify tasks with input and output parameters (variables or array slices) and scheduling based on the dependencies between them. Jade [77] schedules tasks based on their effects, but its effect specifications are less expressive than ours. They are based on shared objects and do not include any hierarchical structure or wildcards. Pāñini [61] dynamically schedules tasks based on effects in terms of reads or writes on fields of classes.

Dynamic Out-of-Order Java [36] uses a combination of static and dynamic pointer analysis of heap data to allow parallelization of programs with sequential-equivalent semantics. Synchronization via Scheduling [18] uses a static task graph and analyzes effects based on a hash of accessed memory object signatures, which may give rise to false conflicts. In Prometheus [8], programmers put each task in a *serialization set*, and a scheduler serializes tasks in the same set. Aida [62] detects conflicts by analyzing dynamically-observed effects at the level of shared objects. It implements 'delegated isolation', where when two tasks conflict, one is rolled back and 'delegated' to be performed by the other.

None of the systems mentioned above support a rich effect system like ours. They generally

express effects in terms of accessed objects or simple identifiers, with no notion of hierarchy or wildcards. Our effect system, adapted from DPJ, offers a major advantage over them in its ability to express parallel access patterns on hierarchical or modular data structures. This rich effect system, in which various effect specifications can refer to different but overlapping sets of memory locations, creates a challenge for the scheduler in identifying when effects conflict. Our work on TWEJava, and in particular its scheduler, seeks to solve this problem while maintaining good performance.

Legion [14, 82] is similar to our system in that it does use effects on hierarchical regions for run-time task scheduling, but it supports only structured parallelism and is designed so that only the effects of child tasks of the same parent need to be compared against each other. Because of this, Legion does not need to employ a global data structure keeping track of effects, instead using a separate tree structure for scheduling the child tasks of each parent task. Legion does not describe a way to access these individual tree data structures in parallel. Unlike Legion, we define a scheduling algorithm that efficiently and scalably supports the less restrained TWE programming model, in which the effects of any two tasks might conflict. Our scheduling algorithm uses a single global tree structure and defines synchronization patterns that permit safe parallel access to that tree, which is critical for scalable scheduling. Unlike Legion, the TWE model also supports explicitly waiting for a task, and to support this while not introducing the possibility of deadlock, we define methods for prioritizing a task and rechecking its effects, which are not present in Legion. We believe our scheduling algorithm could be adapted to work with Legion's region system if it were used in an implementation of the TWE model.

Myrmics [64] also schedules tasks based on hierarchical region-based effects, but it too uses a structured parallelism model that is much more constrained than TWE. It is designed for a custom many-core architecture and supports parallelism in scheduling operations only by partitioning the region tree between several dedicated scheduler cores. This parallelization strategy is not appropriate for the general-purpose multi-core systems we target.

Functional programming systems like Sisal [37] and I-structures [11] support efficient parallel execution of functional programs, via run-time schedulers. These systems do not have to control

concurrent access to mutable data in an imperative programming model, which is the core problem addressed by our use of effects.

8.5 Multi-granularity locking in databases

My work on the tree-based scheduler is also related to other work that uses hierarchical structures for synchronization, in particular multiple-granularity locking [40] in databases. There is a significant body of work in this area, aiming to define efficient locking schemes that allow concurrent access where possible for data in databases. This is a different problem domain from my work on scheduling tasks with effects, and differences in the locking approach include that we always proceed from the root of the tree rather than upwards, and do not use intention locks. In future work, some of the techniques from this body of work could potentially be applied to our tree-based scheduling approach, e.g. supporting dynamic adjustment of the granularity at which the tree is locked [39].

8.6 Run-time checks used for speculative parallelization

Systems which use some form of speculative parallelism are another important class of related work. They are particularly related to my system for supporting dynamic effects, which itself uses speculative parallelism with run-time checks, but they are also related to the basic TWE system in that they reflect alternative approaches for providing some of the safety properties that it does.

Transactional memory (TM) systems [42, 43, 44, 34, 3] are one major category of this work. TM systems use hardware and/or software mechanisms to monitor the memory accesses of atomic blocks as they are executing, and detect if two atomic blocks try to perform conflicting accesses. If this happens, one of the blocks will be aborted and rolled back, and retried later. Our system to support dynamic effects is similar in that it also involves speculative parallelism and detects conflicts between tasks based on their memory effects. However, we rely on programmer-specified and statically-checked effect annotations, which allow us to summarize a potentially large number of memory accesses using a single effect. This contrasts with TM systems that typically need checks at the level of individual memory accesses. Because of this, software-based TM systems often have

high overheads; hardware TM systems can have lower overheads but are generally limited in the size of transactions they support.

TM systems enforce atomicity between atomic blocks, but they do not provide a programming model that can give stronger safety properties. The TWE model guarantees task isolation, data-race freedom, and a form of atomicity for all operations in a program. TM provides only atomicity, and to avoid exorbitantly high overheads many TM systems only guarantee isolation between atomic blocks and not with respect to other operations (weak isolation). Moreover, the TWE model provides a mechanism for guaranteed-deterministic computations, which can be used alongside nondeterministic computations within one program; TM systems do not provide this.

Non-deterministic algorithms in DPJ also use atomic blocks implemented via an STM system [23]. This mechanism allows non-deterministic computations to be expressed in DPJ, but as noted in Chapter 1, DPJ is limited to structured fork-join computations and cannot express other forms of concurrency. Due to overheads of the STM, this mechanism has fairly poor absolute performance.

Another class of related work involves systems that can speculatively parallelize programs or run parallel programs while providing a guarantee of determinism, using checks based on memory accesses to detect conflicts. Behavior oriented parallelization [35] speculatively parallelizes programs and uses checks based on the pages of memory accessed by parallel tasks. Grace [17] takes a similar approach to guarantee determinism for fork-join parallel programs. CoreDet [16] does the same for general threaded programs by either tracking ownership for memory chunks or buffering stores for later commit. In general, these systems require monitoring of memory accesses similar to that used by TM systems, which can cause significant overheads.

Galois [60] and Aida [62] also use speculative parallelism. They detect conflicts at the level of shared objects and must be able to roll back operations on a conflict. Neither has an equivalent to the hierarchical structure and wildcards in our effects, and both are limited to structured, fork-join concurrency. Galois provides guarantees such as data race freedom if used correctly (e.g. only accessing shared data through certain framework methods, and correctly specifying if accesses are commutative), but it provides no static or dynamic mechanism for enforcing these correctness

requirements. Aida implements 'delegated isolation', where when two tasks conflict, one is rolled back and 'delegated' to be performed by the other. It detects conflicts by analyzing dynamically-observed effects at the level of shared objects, making it easier than in our system to detect when the effects of two tasks conflict.

Blelloch et al. [19] use a system of deterministic reservations in which an operation within a parallel computation can be specified as a combination of a *reserve* step that is responsible for checking for conflicts and a *commit* step that performs updates if no conflicts are found. This is similar to our mechanism for dynamic effects in that it does checks for conflicts at run time and uses speculative parallelization. If used correctly, this system can provide a strong safety guarantee (determinism). However, this system requires the programmer to write custom code for the reservations in each parallel computation (which may be fairly complicated and rely on algorithm-specific properties), and it provides no automatic mechanism for checking that this code is correct. Also, the deterministic reservation mechanism is used only for parallel loops, so is less flexible than the general task-based parallelism of TWE.

Chapter 9

Conclusion and Future Work

In this thesis, I have presented the Tasks With Effects (TWE) model for parallel programming and its implementation in the TWEJava language. The TWE model can provide very strong safety guarantees while still having the expressiveness to support a wide range of parallel and concurrent programs. These safety guarantees include task isolation, data race freedom, atomicity, and (optionally) determinism. No other programming model I am aware of combines such strong parallel safety guarantees with the level of expressiveness provided by TWE.

To make the TWE model practical, it is necessary to produce an actual implementation of it and show that it can provide good performance. I have done this through the TWEJava language and the effect-based scheduler design described in this thesis. I have shown that TWEJava can both express a wide variety of parallel programs and give good performance on most of the programs I have benchmarked, including some with fine-grained parallelism.

To enhance the expressiveness of the TWE model and TWEJava to an even larger class of programs, I have also defined extensions to the TWE model that enable it to support programs where the effects of a task can only be determined dynamically while the task is executing. I have produced an initial implementation of this extension for TWEJava, which does provide the desired expressivity benefit, although it currently comes at the cost of significant performance overheads. I believe that those overheads can be substantially reduced, making it practical to use the TWE model for many programs that require the use of dynamic effects. This will be one fruitful direction for future work.

I believe the TWE model also presents several broader directions for possible future research. One of these is to implement it with different effect systems. These could provide different trade-offs between the burden of writing effect annotations, the expressiveness needed to express the

parallelism patterns of various programs, and the run-time overheads of effect checks. Another way of reducing the annotation burden would be to infer some or all of the annotations; some work has been done in this area for our current effect system [84, 83], but opportunities for more extensive annotation inference still remain.

Perhaps even more broadly, I believe it could be fruitful to extend the Tasks With Effects model beyond shared-memory multicore systems. While this has been the focus of my work to date, the basic concepts of tasks with effects could also be applied to other systems, potentially including large-scale HPC systems, systems with various kinds of compute units (CPUs, GPUs, etc.), and other distributed systems. I believe the safety benefits of the TWE model would be valuable in all these types of systems, and applying it effectively in all these domains could provide numerous opportunities for future research.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, Jan. 2011.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, Mar. 2006.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 26–37. ACM, 2006.

[4] 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.

[5] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. on Comp. Systs.*, 22(1):94–136, 2004.

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

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, 2nd Edition.* Addison Wesley, 2007.

[8] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 85–96. ACM, 2009.

[9] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 149–158. ACM, 2008.

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

[11] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, Oct. 1989.

[12] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de recherche RR-7240, INRIA, Mar. 2010.

[13] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-enforced Deterministic Parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 193–206. USENIX Association, 2010.

[14] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11. IEEE Computer Society Press, 2012.

[15] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06. ACM, 2006.

[16] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64. ACM, 2010.

[17] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96. ACM, 2009.

[18] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via Scheduling: Techniques for Efficiently Managing Shared State. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 640–652. ACM, 2011.

[19] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally Deterministic Parallel Algorithms Can Be Fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 181–192. ACM, 2012.

[20] 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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216. ACM, 1995.

[21] R. L. Bocchino and V. S. Adve. Types, Regions, and Effects for Safe Programming with Object-oriented Parallel Frameworks. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 306–332. Springer-Verlag, 2011.

[22] R. L. Bocchino, Jr., 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 *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116. ACM, 2009.

[23] R. L. Bocchino, Jr., 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 *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 535–548. ACM, 2011.

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

[25] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230. ACM, 2002.

[26] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Methodology for Benchmarking Java Grande Applications. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 81–88. ACM, 1999.

[27] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707. ACM, 2010.

[28] M. Burke, K. Knobe, R. Newton, and V. Sarkar. Concurrent collections programming model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 364–371. Springer US, 2011.

[29] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple Ownership. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 441–460. ACM, 2007.

[30] J. Canny. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 8(6):679–698, June 1986.

[31] L. P. Chew. Guaranteed-quality Mesh Generation for Curved Surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, pages 274–280. ACM, 1993.

[32] D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 292–310. ACM, 2002.

[33] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 85–96. ACM, 2009.

[34] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208. Springer-Verlag, 2006.

[35] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 223–234. ACM, 2007.

[36] Y. h. Eom, S. Yang, J. C. Jenista, and B. Demsky. DOJ: Dynamically Parallelizing Object-oriented Programs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 85–96. ACM, 2012.

[37] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A Report on the Sisal Language Project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

[38] C. Flanagan and S. N. Freund. Type-based Race Detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232. ACM, 2000.

[39] G. Graefe. Hierarchical locking in B-tree indexes. In *Business, Technologie und Web*, BTW '07, 2007.

[40] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of Locks in a Shared Data Base. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 428–451. ACM, 1975.

[41] A. Greenhouse and J. Boyland. An Object-Oriented Effects System. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 205–229. Springer-Verlag, 1999.

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

[43] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60. ACM, 2005.

[44] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 14–25. ACM, 2006.

[45] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson. A Production-quality C* Compiler for Hypercube Multicomputers. pages 73–82, 1991.

[46] S. Heumann and V. Adve. Disciplined Concurrent Programming Using Tasks with Effects. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 7–7. USENIX Association, 2012.

[47] S. Heumann and V. Adve. Tasks with Effects: A Model for Disciplined Concurrent Programming. In *Third Workshop on Determinism and Correctness in Parallel Programming*, WoDet 3, 2012.

[48] S. T. Heumann, V. S. Adve, and S. Wang. The Tasks with Effects Model for Safe Concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 239–250. ACM, 2013.

[49] S. T. Heumann, A. Tzannes, and V. S. Adve. Scalable task scheduling and synchronization using hierarchical effects. In *2015 International Conference on Parallel Architecture and Compilation*, PACT '15, pages 125–137. IEEE, 2015.

[50] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice Univ., Houston, TX, 1993.

[51] D. R. Hower and M. D. Hill. Hobbes: CVS for Shared Memory. In *Workshop on Determinism and Correctness in Parallel Programming*, WoDet 2, 2011.

[52] Intel. Intel Threading Building Blocks Reference Manual. `https://software.intel.com/en-us/tbb-reference-manual`.

[53] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A Programming Model for Concurrent Object-oriented Programs. *ACM Trans. Program. Lang. Syst.*, 31(1):1:1–1:48, Dec. 2008.

[54] J. C. Jenista, Y. h. Eom, and B. C. Demsky. OoOJava: Software Out-of-order Execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 57–68. ACM, 2011.

[55] J. B. Kam and J. D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *J. ACM*, 23(1):158–171, Jan. 1976.

[56] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[57] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[58] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe Parallel Programming Using Dynamic Dependence Hints. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 243–258. ACM, 2011.

[59] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206. ACM, 1973.

[60] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222. ACM, 2007.

[61] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit Invocation Meets Safe, Implicit Concurrency. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 63–72. ACM, 2010.

[62] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated Isolation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 885–902. ACM, 2011.

[63] J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57. ACM, 1988.

[64] S. Lyberis. *Myrmics: A Scalable Runtime System for Global Address Spaces.* PhD thesis, University of Crete, 2013.

[65] T. J. Marlowe and B. G. Ryder. Properties of Data Flow Frameworks: A Unified Model. *Acta Inf.*, 28(2):121–163, Dec. 1990.

[66] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven Optimizations for Amorphous Data-parallel Programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 3–14. ACM, 2010.

[67] Microsoft. Axum Programmer's Guide. `http://download.microsoft.com/download/B/D/5/BD51FFB2-C777-43B0-AC24-BDE3C88E231F/Axum%20Programmers%20Guide.pdf`.

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

[69] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization*, IISWC '08, pages 35–46. IEEE, 2008.

[70] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108. ACM, 2009.

[71] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.5. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`, 2015.

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

[73] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing*, pages 142–151. IEEE, 2008.

[74] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308. ACM, 1996.

[75] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25. ACM, 2011.

[76] J. H. Reppy. *Concurrent Programming in ML*. Cambridge Univ. Press, 1999.

[77] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, May 1998.

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

[79] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 275–299. Springer-Verlag, 2010.

[80] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: A High-productivity Programming Language for HPC with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 81:1–81:12. ACM, 2015.

[81] Thinking Machines Corp. CM Fortran reference manual, version 1.0. Technical report, Thinking Machines Corp., Cambridge, Massachusetts, February 1991.

[82] S. Treichler, M. Bauer, and A. Aiken. Language Support for Dynamic, Hierarchical Data Partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 495–514. ACM, 2013.

[83] A. Tzannes, S. T. Heumann, L. Eloussi, M. Vakilian, V. S. Adve, and M. Han. Region and Effect Inference for Safe Parallelism. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 512–523. IEEE Computer Society, 2015.

[84] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. Inferring Method Effect Summaries for Nested Heap Regions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 421–432. IEEE Computer Society, 2009.

[85] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A Unified Scheduler for Recursive and Task Dataflow Parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 1–11. IEEE Computer Society, 2011.