The JCilk Multithreaded Language

by

I-Ting Angelina Lee

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
August 17, 2005

Certified by .... [illegible] ....
Charles E. Leiserson
Professor
Thesis Supervisor

[illegible]

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The JCilk Multithreaded Language

by

I-Ting Angelina Lee

Submitted to the Department of Electrical Engineering and Computer Science
on August 17, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

JCilk is a Java-based multithreaded programming language which extends Java to provide a dynamic threading model. Specifically, JCilk imports Cilk's fork-join primitives `spawn` and `sync` into Java to provide procedure-call semantics for concurrent subcomputations. More importantly, JCilk integrates exception handling with multithreading by defining semantics consistent with Java's existing semantics of exception handling.

JCilk's strategy of integrating multithreading with Java's exception semantics yields some surprising semantic synergies. In particular, JCilk extends Java's exception semantics to allow exceptions to be passed from a spawned method to its parent in a natural way that obviates the need for Cilk's `inlet` and `abort` constructs. This extension is "faithful" in that it obeys Java's ordinary serial semantics when executed on a single processor. When executed in parallel, however, an exception thrown by a JCilk computation signals its sibling computations to abort, yielding a clean semantics in which only a single exception from the enclosing `try` block is handled. To minimize the complexity of reasoning about aborts, JCilk signals them "semisynchronously" so that abort signals do not interrupt ordinary serial code. Because JCilk uses Java's normal exception mechanism to propagate an abort throughout a subcomputation, the programmer can handle clean-up by simply catching a thrown `CilkAbort` exception. This thesis documents in detail the designed semantics, the linguistic decisions we made, and their justifications.

This thesis also describes the structure of JCilk compiler and how it supports the exception semantics. Specifically, the JCilk compiler performs a two-stage compilation process to support the continuation mechanism required by the runtime system's work-stealing algorithm. By performing static analysis, the compiler generates code to support the "catchlet" and "finallet" mechanisms for handling exceptions.

The design of JCilk represents joint research with John S. Danaher and Charles E. Leiserson.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

I have received much help, one way or the other, along the way. There are many people whom I would like to thank, but I haven't mentioned their names here one by one, because otherwise this would be a really long list.

I'd like to thank the following people, in no particular order:

Thanks to everyone in the Supercomputing Technologies group. Since I joined the group, everyone has helped me with ideas and writing. In particular, thanks to Bradley Kuszmaul and Gideon Stupp for helpful discussions; thanks to Kunal Agrawal, Jeremy Fineman, and Jim Sukha who are always willing to help me organize my thoughts and improve my writing; thanks to Alissa Cardone, who read through my entire thesis draft to help me with my English, despite the fact that she is not in the Computer Science field and takes no personal interest in my thesis topic.

Thanks to Supertechies of olden days who created the original Cilk. JCilk wouldn't exist without their work.

Thanks to the Scott Ananian for his many helpful discussions and insightful comments. His in-depth knowledge of Java helped us get off the ground.

Thanks to the Singapore-MIT Alliance for giving me an opportunity to present my work to an audience outside of my group. In particular, thanks to Hsu Wen Jing, He Yuxiong, Chung Shin Yee, Wong Weng Fai, and Chin Wei Ngan, for their helpful comments and their genuine hospitality during our stay in Singapore.

Thanks to people who built Polyglot [40]. Without them, this project would have taken a lot more engineering time and effort.

Thanks to my families and friends for their constant support and encouragement. Special thanks to Mom and Dad. Without them calling me and nagging at me on a weekly basis, I would not have finished by mid August and managed to find time to go home before school starts again. Thanks to Geoff Volker for his encouragement and guidance during the period of time when I was having serious doubts about staying in graduate school. Thanks to Bryan Chiang for always being there for me and for believing in me. Thanks to Vered Anzenberg for listening to my whining and distracting me away from my thesis at appropriate moments so that I wasn't completely stressed out.

Thanks to John for implementing the JCilk-1 runtime system and for being a wonderful partner. It was a great pleasure working with him. Without him, this project wouldn't haven gotten this far.

Thanks most of all to my adviser, Charles Leiserson. Without his ideas and constant support, JCilk would not have gotten started. I like to mention more than that, however. Charles is a great adviser. He encouraged me when I was having doubts about my own abilities. He taught me how to struggle with my own writing and how to improve my presentation skills. He always sees the brighter side of things and is always full of energy and enthusiasm. He has been wonderful.

This work was supported in part by the Singapore-MIT Alliance and in part by NSF Grants ACI-0324974 and CNS-0305606. During the past year, I was supported in part by a Sun Microsystems Fellowship.

Contents

Chapter 1

Introduction

The proliferation of multiprocessor machines has accentuated the importance of linguistic support for writing parallel programs. Although the shared-memory multiprocessor machine is becoming a commodity, the development of parallel applications seems to be growing slowly. Most programmers prefer to write serial programs rather than parallel programs, because writing parallel programs is inherently more challenging than writing serial programs. When developing parallel software, programmers must worry about task decomposition, scheduling, communication, and synchronization. Parallel software is also more difficult to debug due to issues such as race conditions and deadlocks, which do not exist in serial software. For programmers to take advantage of the fast growth in parallel hardware, adequate linguistic support for writing parallel software is necessary.

Over the past year, together with the rest of the JCilk design team, I have created a high-level parallel programming language, JCilk, and hammered out a solid set of semantics for the language. JCilk is a Java-based multithreaded language which aims to support the development of parallel programs efficiently by automating thread management and providing simple yet powerful constructs for parallel programming. Specifically, JCilk extends the semantics of Java [20] by introducing "Cilk-like" [16,46] linguistic constructs for parallel control.

The original Cilk language provides a ***dynamic threading model*** that supports call-return semantics in a C language [28] context. A dynamic threading model means that the execution of a procedure can migrate between processors. The binding between the procedure and the processor executing the procedure is not determined at compile time and does may change between activations of the procedure. In Cilk, when a parent procedure "spawns" a child procedure, the parent can continue to execute in parallel with the spawned child. The processor executing the parent before the spawn may be different from the one executing it after the spawn. When the child returns to the parent, the system handles the synchronization required for the child to update the parent's field. The Cilk system also includes a provably good scheduler, guaranteeing that programs take full advantage of the processors available at runtime. Cilk provides a set of simple parallel programming constructs to interface with the dynamic threading model. These constructs are supplied as the language primitives, and no lengthy protocol is involved to use them. Cilk's dynamic threading model and its call-return semantics are particularly appropriate for coding algorithms which use a divide-and-conquer strategy. The programmer does not need to worry about tuning the base task granularity, because the parallelism among task computations is determined at runtime. As a C language extension, however, Cilk inherently lacks support for modern language features such as object orientation, automated memory management, and exception handling.

Java, on the other hand, provides these desirable features but supports a totally different threading model, a ***static threading model***. That means the execution of a procedure cannot migrate between virtual processors. The binding of the procedure to a virtual processor is determined at compile time and is persistent across activations of the procedure. Java's static threading model is suited for applications with persistent concurrent tasks, such as displaying multiple independent animations and doing background computation while waiting for user input. It is not suited for applications that solve problems using the divide-and-conquer strategy, however. Specifically, the divide-and-conquer strategy involves recursively splitting tasks into subtasks, solving them in parallel, and composing their results. With static threading, the programmer must tune the base task granularity depending on the platform where the program executes. If the granularity is too small, the cost of constructing

Property	*Static threading model*	*Dynamic threading model*
Binding	determined at compile time	determined at runtime
Binding life span	unchanged through activations of a procedure	may change through activations of a procedure
Semantics	object-based semantics	call-return semantics
Creation	protocol	declarative (“`spawn`”)
Applications	persistent concurrent tasks	divide-and-conquer algorithms
Thread relations	independent	parent-child relations

Figure 1-1: The differences between static threading model and dynamic threading model. The binding in the first column refers to the binding between a procedure and the processor executing the procedure.

and managing the virtual processors can be greater than the computation time spent on the actual task. In addition, the threading support is provided as a library (part of Java API) but not as a language primitive. The threading model does not support the passing of exceptions or return values from one thread back to its “parent thread”.[1] Although this threading model is designed to be general purpose, it represents a tradeoff between expressiveness and convenience. In particular, a protocol is required in order to use the threading model.

The static threading model and the dynamic threading model are very different, and they are each suited for different types of applications. Figure 1-1 outlines the differences between these two models.

Since both Java and Cilk have their own pros and cons, the JCilk design team attempted to create a new language that combines the desirable features of both. In particular, we wanted to explore how the dynamic threading model behaves in the context of Java, which contains a rich set of modern language features that are not supported by Cilk. In order to achieve this goal, we ported Cilk's dynamic threading model and its parallel-programming primitives into Java, integrating them with Java's modern language features. When we brought the linguistic primitives from Cilk into the context of Java, we were faced with the task of ensuring that the new constructs would interact nicely with Java's existing constructs. Java's exception mechanism,

[1]This statement is true for the threading support in Java 1.4 and earlier versions. Java 1.5 Tiger provides a mechanism, however, that allows a thread to return a value or throw an exception.

which has no analogue in C or Cilk, turns out to be the language feature most directly impacted by the new Cilk-like primitives. Surprisingly, the interaction is synergistic, not antagonistic.

The remainder of this chapter serves as a basis for understanding JCilk's semantics. Before we dive deep into JCilk's semantics, which includes how exception handling can be integrated with Cilk's parallel-programming primitives, it is important to understand the language features supported by Java and Cilk that are impacted by this integration. Section 1.1 gives an overview of Java's exception-handling mechanism and its threading model. Section 1.2 gives some background on Cilk's parallel-programming constructs and its dynamic threading model. Section 1.3 overviews the contributions of JCilk. Section 1.4 concludes the chapter by outlining the structure of this thesis.

As a fusion of Java and Cilk, JCilk contains other modern language features that Java offers besides an exception mechanism. Since these features are not the main focus of the thesis, I do not discuss them here. Interested readers may refer to [20].

1.1 Java

Java [20] provides several significant language features not offered in C [28], which are specifically designed to make programming easier. This section reviews Java's exception handling mechanism and threading model, because they are the language features most directly impacted by the new Cilk primitives.

Exception handling

One of the most powerful and elegant programming constructs that Java provides is its exception handling mechanism [20, Ch. 11]. Unlike in C, there are two different ways to transfer control out of a method in Java. One way is to do an ordinary return to a method's caller. Another way is to ***throw*** an exception, which terminates the execution at the point of throw and skips the rest of the method body. In general, an exception either is triggered by the Java virtual machine when an unexpected error occurs (such as the divide-by-zero error) or is explicitly thrown using Java's keyword `throw` by the executing method to indicate the occurrence of an abnormal situation. The exception, if not ***caught*** within the same method using Java's `try` and `catch` constructs, is propagated upward to the caller. Once an exception occurs, the throwing method terminates, and the control is transferred nonlocally from the point where the exception is thrown to a point where the program specifies the exception is caught. All methods between the ***throw point*** and the ***catch point*** within the execution stack are terminated due to the exception, as specified by Java's termination model [9].

Java provides two types of exceptions: ***checked exceptions*** and ***unchecked exceptions*** [20, Sec. 11.2].[2] The proper use of checked exceptions is verified by any standard Java compiler, while the unchecked ones are not. The unchecked exceptions include `java.lang.Error`, `java.lang.RuntimeException`, and their subclasses. These exceptions are exempted from compile-time checking, because they can occur virtually at any point in the program, and checking them at compile time would not necessarily establish the correctness of the program. Any other exception is a checked exception, which must be declared at the method header if the method can possibly throw it.

Figure 1-2 shows a simple Java method which exercises the exception-handling mechanism. The `try` block in lines 5–7 and surrounding the call to method `C` has a `catch` clause in lines 7–9. This code indicates that if `C` throws an `ArithmeticException`, the exception is handled by executing the `cleanupC` method in line 8. Similarly, during any point of execution in lines 4–9, if a `NullPointerException` occurs, it is handled by the `cleanup` method in line 11. Once an exception is handled, the execution continues on from the point after the handler.

When method `f1-2` executes, several possible scenarios can happen:

1. If no exception occurs, methods `A`, `B`, `C`, and `D` are called.
2. If the call to method `B`, the call to method `C`, or the call to method `cleanupC`

[2] In this thesis, when I refer to exception, I do not mean the `Exception` class in Java. Rather, I am using it in a broader sense to include the entire `Throwable` hierarchy in Java. When I refer to the actual `Exception` class, I use "`Exception`" explicitly with a capitalized "E."

```
 1   public void f1-2() {
 2      A();
 3      try {
 4         B();
 5         try {
 6            C();
 7         } catch(ArithmeticException e) {
 8            cleanupC();
 9         }
10      } catch(NullPointerException e) {
11         cleanup();
12      }
13      D();
14   }
```

Figure 1-2: A simple Java method that uses try-catch constructs.

throws a `NullPointerException`, execution completes abruptly at the throw point, and the control jumps to the `catch` clause in line 10. Hence, methods `A`, `B`, `cleanup`, and `D` are always called. Methods `C` and `cleanupC` may also be called depending on which method threw the exception.

3. If method `C` throws an `ArithmeticException`, the control jumps to the `catch` clause in line 7. In this case, methods `A`, `B`, `C`, `cleanupC`, and `D` are called. Method `cleanup` may also be called, if `cleanupC` throws a `NullPointerException`.
4. If any statement in method `f1-2` throws an unchecked exception that is not handled, the exception propagates up to method `f1-2`'s caller, and method `f1-2` completes abruptly with the reason of the thrown exception. (No checked exception can be thrown by this method, or the method would not compile.)

Java's `try` statement may also contain a `finally` clause, which is always executed whether an exception is thrown or not. The execution rule for the `finally` clause gives another set of possibilities of how the `try` statement can turn out. If the `try` block completes normally, and the `finally` clause completes abruptly with the reason S, the `try` statement is considered to complete abruptly with the reason S. If the `try` block completes abruptly with the reason S and the `finally` clause completes normally, the `try` statement completes abruptly with the reason S. If the `try` block completes abruptly with the reason S and the `finally` clause completes abruptly with the reason E, then the `try` statement completes abruptly with the reason E. In other words, the exception thrown by the `finally` clause overwrites the exception thrown by the `try` block.

Static threading model

Java provides support for multithreaded programs with static threads. To interact with the threading model, the programmer explicitly creates several virtual processors and specifies the procedure to be executed by each processor. The binding between a procedure and the processor executing the procedure is determined at compile time and remains unchanged throughout the activations of the procedure. A virtual processor exits and gets garbage-collected once it completes the computation specified by the programmer.

A protocol is involved when creating a virtual processor. A virtual processor in Java is an object, represented by the `Thread` class. The programmer creates a `Thread` object either by declaring a class that extends from the `Thread` class, or by declaring a class that implements the `Runnable` interface. The computation performed by a particular `Thread` object is specified in its `run` method in the same class. When executing instructions that invoke the constructor and then the `start` method of the `Thread` class, the Java virtual machine creates a `Thread` object and invokes its `run` method, which may contain instructions that ask for more `Thread` objects to be created.

The `Thread` objects in Java do not communicate with one another directly. The threading model does not support the passing of exceptions or return values from one `Thread` object back to its "parent" `Thread` object. The Java API interface specifies that the `run` method does not return any value and cannot throw any exceptions. Any potential exception that can be thrown within the `run` method must be handled entirely within the method. An unexpected exception terminates the `Thread` object, propagates the exception up to the root of its "`ThreadGroup`," and kills the entire program. To circumvent this default behavior, the programmer must override the ThreadGroup class. [3] The Thread objects can communicate via shared data structures or share objects allocated in the heap. The programmer must carefully synchronize between different Thread objects accessing the shared data structures or objects. For this purpose, Java provides various high-level mechanisms for synchronization.

1.2 Cilk

Cilk encourages programmers to concentrate on how best to divide and parallelize the computation at hand, leaving the runtime system to worry about load balancing and task scheduling during execution. This idea is well expressed in Cilk via its dynamic threading model and provably good scheduler. While Cilk supports these features, it extends C with only a few keywords for parallel constructs. Since a parallel program inherently contains certain nondeterminism, Cilk also provides thread atomicity to ease the difficulty of reasoning about procedure execution. Furthermore, Cilk provides an abort mechanism to terminate extraneous computation, which is essential for parallel search algorithms that require speculative computing. This section gives an overview on the Cilk language in general.

Programming interface

Cilk's extension of C is simple, requiring only three additional keywords for parallelism and synchronization: `cilk`, `spawn`, and `sync`.

In Cilk, parallelism is created using the keyword `spawn`. When a procedure call is preceded by the keyword `spawn`, it is ***spawned*** and can be executed in parallel with the caller. For instance, `fib(n-1)` in line 6 of Figure 1-3 is spawned by and can be executed in parallel with `fib(n)`. We refer to the spawned procedure as the ***child***, and the spawning procedure as the ***parent***.

The complement of `spawn` is the keyword `sync`, which acts as a local barrier and

[3] In general, this protocol is the only way that a programmer can intercept uncaught exceptions that occur in a Thread object in Java 1.4. Java 1.5 Tiger provides a more convenient mechanism.

```
1    cilk int fib(int n) {
2        if(n<2) {
3            return n;
4        } else {
5            int x, y;
6            x = spawn fib(n-1);
7            y = spawn fib(n-2);
8            sync;
9            return (x+y);
10       }
11   }
12   cilk int main(int argc, char *argv[]) {
13       int n, result;
14       n = atoi(argv[1]);
15       result = spawn fib(n);
16       sync;
17       printf("Result: %d\n", result);
18       return 0;
19   }
```

Figure 1-3: A simple Cilk program to compute the Fibonacci number in parallel (using a naive exponential-time algorithm).

joins the parallelism forked by `spawn` together. The Cilk runtime system ensures that statements after `sync` are not executed until all procedures spawned before the `sync` statement have completed and returned. In Figure 1-3, the values returned by `fib(n-1)` and `fib(n-2)` (i.e., `x` and `y`) are not used until after the `sync` statement in line 8. Without the `sync` statement, the values of `x` and `y` might be used before being computed, which would result in a synchronization bug.

The keyword `cilk` is a function modifier which identifies the declared function as a ***cilk* procedure**. A `cilk` procedure is analogous to a C function except that it can be spawned off to execute in parallel. Only a `cilk` procedure can be spawned, and an ordinary C function cannot.

Cilk extends the semantics of C in a natural way so that every Cilk program has a corresponding legal C program, called the ***serial elision*** of the Cilk program. This serial elision is obtained by eliding all Cilk keywords from the Cilk program. The serial elision of a Cilk program is always one of the correct interpretations of the Cilk program under the parallel semantics.

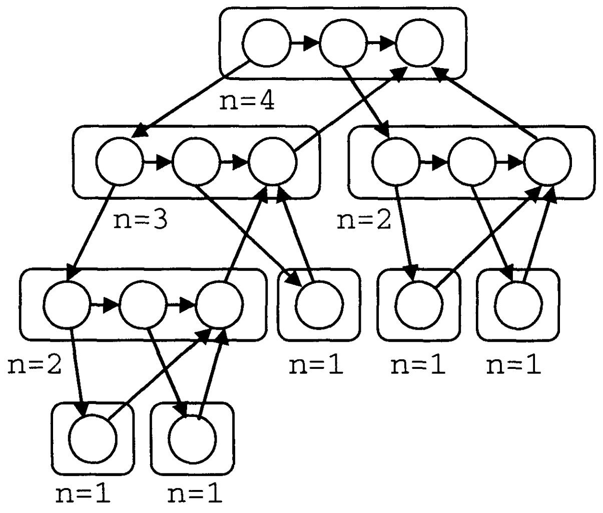

Figure 1-4: The DAG representation of the `fib` program in Figure 1-3, computing `n` = 4. Each rectangle in the dag represents a `cilk` procedure, and each node represents a logical thread. An arrow between two nodes indicates a precedence dependency between the logical threads, where the thread at the tail of the arrow must complete before the thread at the head can begin.

Dynamic threading model

In Cilk, the keywords `spawn` and `sync` specify the logical parallelism of the program rather than the actual parallelism at execution time. A ***logical thread*** is a maximal sequence of nonblocking instructions that ends with a `spawn`, `sync`, or `return` statement. A Cilk program consists of a collection of `cilk` procedures, and each `cilk` procedure consists of a sequence of logical threads. We can model the execution of a Cilk program as a directed acyclic graph, or ***dag***. Such dag is shown in Figure 1-4, which illustrates the execution of `fib` (in Figure 1-3) with argument `n` = 4. Two threads are logically in series if there is a directed path between them. Otherwise, they are logically in parallel. A correct execution of a Cilk program must obey all dependencies in the dag: a thread cannot be executed until all threads it depends on have completed.

The number of virtual processors created at runtime does not necessarily correspond to the number of logical threads specified in a Cilk program. Rather, it is determined at runtime, when the resource availability and processor load are known. As a program executes, the Cilk scheduler dynamically distributes the logical threads across the available processors while maintaining the dependencies among the logical threads. With the abstraction of logical threads, Cilk decouples the parallelism of the program specified at compile time from the virtual-processor creation at runtime.

Provably good scheduler

To take full advantage of the dynamic threading model, a good scheduler is needed to distribute the logical threads dynamically across the available processors. The Cilk runtime system uses a provably good scheduler which implements a "work-stealing" algorithm [8]. The implementation is based on the "work-first" principle [16]. (I present the work-first principle and the work-stealing algorithm formally in Chapter 3.) It has been shown theoretically [8] and empirically [7, 16] that, the Cilk scheduler's work-stealing algorithm schedules threads near optimally. The dynamic threading model and the provably good scheduler come hand in hand. Both are integral components in the Cilk runtime system.

Thread atomicity

Cilk guarantees atomicity between threads interacting within one procedure. In a Cilk program, when a parent spawns off multiple children, the order of the children returning is nondeterministic, meaning the order of returning can be different in each run depending on how the threads are scheduled. This nondeterminism is intrinsic to parallel computation but makes it difficult to reason about the interactions between the parent and the returning children. To ease the difficulty of reasoning about the procedure execution, Cilk guarantees atomicity between the threads that interact within one procedure. That is, two threads in the same procedure never run simultaneously and their instructions don't interleave. Thus, when a child returns and updates a local field of the parent, the update is performed atomically with respect to other updates and to the parent. It is still possible to write code with race conditions, however, since Cilk only guarantees atomicity between threads within one procedure, but makes no guarantee on threads in different procedures.

Cilk also supports a construct called `inlet`, which is essentially a local C function invoked immediately when a spawned child returns, to incorporate the returned value into a parent's local field in a more complex manner. Thread atomicity also applies to inlet threads updating parent's local fields.

Speculative computing

Some parallel search algorithms, such as branch-and-bound and heuristic search [11], require speculative computation. In these algorithms, some computations may be spawned off speculatively, but are later found to be unnecessary. In such cases, one wishes to terminate these extraneous computations as soon as possible so that they do not consume system resources. Cilk provides an `abort` statement to allow speculative computations to be aborted when their work is rendered unnecessary.

The `abort` statement in Cilk is usually used together with `inlet`, since the `inlet` function gives the programmer a chance to examine the returned value from the spawned child and take the appropriate actions accordingly. In the case of speculative computing, the appropriate action can mean aborting other spawned parallel computations. The `abort` statement triggers the abort process: the runtime system chases down all children spawned, and it recursively terminates all descendants of the procedure that initiated the abort.

The existing abort mechanism in Cilk has two weaknesses, however. First, once abort is signaled, the aborted children are simply terminated, and they vanish from the programmer's perspective, leaving the programmer no chance to cleanup. Second, the abort signal terminates only descendants that have already been spawned and have not yet completed, but it does not terminate any future children that have not yet been spawned. It is the programmer's responsibility to take care explicitly not to spawn any future computations. In practice, this situation is often handled by setting some flag when the abort occurs, and the programmer explicitly checks the flag before any subsequent `spawn` statement in order to prevent future children from spawning.

1.3 Contributions of JCilk

JCilk introduces a dynamic threading model into Java, making the following research contributions:

1. defining a faithful extension of Java's exception-handling semantics for multithreading;
2. implementing an implicit abort mechanism as part of those semantics;
3. providing an efficient mechanism to support Java's "lexical-scope" rule in the face of concurrency;
4. building a compiler and runtime system to support the JCilk linguistics.

We now examine each of these contributions in turn.

Exception-handling semantics

JCilk defines a concrete set of exception-handling semantics in a concurrent context. This semantic extension to Java is faithful in that it handles concurrency in spawned methods using semantics that are consistent with the existing semantics of Java's `try` and `catch` constructs.

An exception-handling mechanism facilitates production of robust and fault tolerant software that can behave reasonably under both normal and unusual circumstances. When an unusual circumstance occurs, a procedure might not always have sufficient knowledge of the context to handle it. Its parent procedure, however, is likely to have more knowledge of the context and may be able to cope with the unusual circumstance better. An exception mechanism allows the child procedure to communicate the unusual circumstance to its parent procedure.

Most parallel languages do not provide an exception-handling mechanism. The parallel languages that do provide exception support, such as [21, 30, 41], are built upon serial languages, whose support for exceptions is only semantically defined for serial execution. These parallel languages do not extend the exception-handling semantics to a concurrent context. JCilk is one of the few parallel languages that address the issue of integrating exception handling with multithreading and the first to define a concrete set of semantics consistent with the existing serial semantics of Java's exception mechanism.

Implicit abort mechanism

JCilk implements an implicit abort mechanism as part of JCilk's exception semantics. When multiple JCilk computations are executed in parallel, an exception thrown by one computation signals its sibling computations to abort. This implicit abort yields a clean semantics in which only a single exception from the enclosing `try` block is handled.

In ordinary Java, an exception causes a nonlocal transfer of control to the `catch` clause of the nearest dynamically enclosing `try` statement whose `catch` clause handles the exception. The *Java Language Specification* [20, pp. 219–220] states,

> "During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception."

In JCilk, we have striven to preserve these semantics while extending them to cope gracefully with the parallelism provided by the Cilk primitives. Specifically, JCilk extends the notion of "abruptly completes" to encompass the implicit aborting of any side computations that have been spawned off and on which the "abrupt completion" semantics of the Java exception-handling mechanism depends. It turns out that this abort mechanism is cleaner to code with than the inlet-based one provided by Cilk. In particular, JCilk extends Java's exception semantics to allow exceptions to be passed from a spawned method to its parent in a natural way so that the need for Cilk's `inlet` and `abort` constructs is obviated.

The lexical-scope rule

JCilk provides an efficient mechanism to support Java's "lexical-scope" rule in the face of concurrency. In JCilk, due to the presence of parallelism, a block of code may be executed later in time than its lexically subsequent blocks of code. In such cases, in order to ease the difficulties of reasoning about the local variable states, JCilk enforces Java's lexical-scope rule, where the state of a local variable can be determined lexically. JCilk's mechanism for enforcing the lexical-scope rule is rather efficient: the mechanism avoids creating extraneous copies of local variables. In addition, the implementation of the mechanism exploits the structure of the JCilk scheduler and creates small overhead only when parallelism is present.

Compiler and runtime system

JCilk's compiler and runtime system support the JCilk linguistics. The system implementation serves as an important foundation for JCilk's semantic design. Only after implementing the actual system, can we be certain that our semantic design is "reasonable" and not just some semantics on paper. Furthermore, the system allows us to measure the overhead of supporting mechanisms required by the semantics and understand the tradeoffs between different options.

A unique feature in the JCilk compiler is its support for the code migration required by the dynamic threading model. To support code migration, additional issues were encountered while implementing the JCilk compiler that did not need to be faced when implementing the Cilk compiler. The Cilk compiler was able to use C's `goto` statements and the flexibility of its pointers to support a "continuation" mechanism. Java does not provide these facilities directly, however, and thus, JCilk must support code migration by other means. The eventual design of the JCilk compiler deploys a unique two-stage compilation process which implements a intermediate language GoJava to support continuations. The GoJava extends Java with support for the `goto` statements, but in a limited and specific circumstances. The end result is a JCilk compiler that supports the code migration while maintaining relative portability and easy extensibility.

Another mechanism in the JCilk runtime system, which is not found in Cilk, is the support for implicit abort required by JCilk's exception semantics. To be specific, the runtime system builds a data structure, called a "try tree," during execution to keep track of any spawned children executing on other processors that may potentially throw exceptions. This data structure is maintained throughout the execution, but it imposes only a small overhead.

1.4 Structure of this thesis

This section provides an outline of the remainder of the thesis. This thesis contains three parts: semantics, implementation, and future perspective.

The first part of the thesis addresses JCilk's semantics. Chapter 2 presents JCilk's full semantics, the design decisions we made throughout the process, and their justifications. The discussion of JCilk's semantics focuses on the exception-handling semantics in concurrent context and how they facilitates the abort mechanism.

The next part of the thesis describes JCilk's system implementation, which includes Chapter 3 and Chapter 4. Chapter 3 first reviews the concepts that JCilk inherits from Cilk and the implementation of Cilk's compiler. The chapter then explains how JCilk spins these concepts to implement them in the context of Java. Chapter 4 presents the new features provided by the JCilk compiler in order to support semantics found in JCilk but not Cilk. Since I am the main implementer of the JCilk-1 compiler, I focus heavily on the implementation of the compiler and the structure of the compiled output. The complementary portion of the system, the JCilk-1 runtime system was implemented by my collaborator, John Danaher. His work is described in his thesis [12], which includes detail about the JCilk-1 runtime system implementation.

The last part of the thesis puts the JCilk project into perspective. Chapter 5 presents related work, placing JCilk and its exception-handling mechanism into the context of parallel programming languages. Chapter 6 proposes possible future directions for JCilk and offers some concluding remarks.

Much of this thesis represents collaborative work with John S. Danaher and Charles E. Leiserson. In particular, Chapter 2 is based on our joint written work [13].

Chapter 2

JCilk Semantics

JCilk extends the Java language to provide call-return semantics for multithreading, much as Cilk does for C. Although the original Cilk language provided the basic semantics for `spawn` and `sync`, it provided no semantics for exceptions, because a spawned Cilk procedure can only return, just as a C function can only return. Java, however, allows a method to signal an exception, rather than return normally, and JCilk's semantics must cope with this eventuality. This chapter shows how JCilk integrates exception handling with multithreading by defining a semantics consistent with the existing semantics of Java's `try` and `catch` constructs, but which handles the concurrency provided by spawned methods.

This chapter, which describes joint research with John S. Danaher and Charles E. Leiserson, is organized as follows. Section 2.1 describes JCilk's basic syntax and semantics and the concepts underlying the language, including the assumption of "implicit atomicity" and the built-in exception class `CilkAbort`. Section 2.2 elaborates on how JCilk uses these concepts in its linguistic design and discusses JCilk's exception semantics in depth. Section 2.3 offers two examples to demonstrate how JCilk's parallel constructs can be used to write applications that involve speculative computing.

2.1 Basic semantics

This section concentrates on the basic semantics of JCilk, including JCilk's philosophy, basic syntax, and the concepts underlying the language. In particular, the concepts include JCilk's guarantee of implicit atomicity and its built-in exception class `CilkAbort`.

Philosophy

The philosophy behind the JCilk extension to Java follows that of the Cilk extension to C: the multithreaded language should be a true semantic parallel extension of the base language. JCilk extends Java[1] by adding new keywords that allow the program to execute in parallel. If the JCilk keywords for parallel control are elided from a JCilk program, however, a syntactically correct Java program results, which we call the ***serial elision*** of the JCilk program. JCilk is a ***faithful*** extension of Java, because the serial elision of a JCilk program is a correct (but not necessarily sole) interpretation of the JCilk program under its parallel semantics.

Syntax

Specifically, JCilk introduces three new keywords — `cilk`, `spawn`, and `sync` — which are the same keywords used to extend C into Cilk, and they have essentially the same meaning in JCilk as they do in Cilk (see Section 1.2). For the purpose of completeness, these keywords are presented here again, this time in the context of JCilk.

Analogous to Cilk, in JCilk the keyword `cilk` is used as a method modifier, and `spawn` and `sync` cannot be used in a Java method unless the method is a `cilk` method. In order to make parallelism plain to the programmer, JCilk enforces the constraint that `spawn` and `sync` can only be used inside a method declared to be `cilk`. A `cilk` method can call a Java method, but a Java method cannot spawn (or call) a `cilk` method. When a parent method spawns a child method, which is accomplished by preceding the method call with the `spawn` keyword, the parent can continue to

[1]Actually, JCilk extends the serial portion of the Java language, and it omits entirely Java's support for static multithreading as provided by the `Thread` class.

```
1   cilk int f2-1() {
2      int w = spawn A();
3      int x = B();
4      int y = spawn C();
5      int z = D();
6      sync;
7      return w + x + y + z;
8   }
```

Figure 2-1: A simple JCilk program.

execute in parallel with its spawned child. The `sync` keyword acts as a local barrier. The JCilk runtime system ensures that program control cannot go beyond a `sync` statement until all previously spawned children have terminated. In general, until a `cilk` method executes a `sync` statement, it cannot safely use results returned by previously spawned children.

Different from Cilk, however, in JCilk the `cilk` keyword can also be used as a modifier for a `try` statement. JCilk enforces the constraint that `spawn` and `sync` keywords can only be used within a `cilk try` block, but not within an ordinary `try` block. Placing `spawn` or `sync` keywords within a `catch` or `finally` clause is illegal in JCilk, whether the `catch` or `finally` clause belongs to a `cilk try` statement or to an ordinary `try` statement. The reason `try` blocks containing `spawn` and `sync` must be declared `cilk` is that when an exception occurs, these `try` statements may contain multiple threads of control during exception handling. Although the JCilk compiler could detect and automatically insert a `cilk` keyword before a `try` statement containing `spawn` or `sync`, we feel the programmer should be explicitly aware of the inherent parallelism. We disallow `spawn` and `sync` within `catch` or `finally` clauses for implementation simplicity, but we might consider revisiting this decision if a need arises.

To illustrate how we have introduced these Cilk primitives into Java, consider the simple JCilk program in Figure 2-1. The method `f2-1` spawns off the method `A` to run in parallel in line 2, calls the method `B` normally (serially) in line 3, spawns `C` in parallel in line 4, calls method `D` normally in line 5, and then itself waits at the `sync` in line 6 until the subcomputations `A` and `C` have completed. When they both complete, `f2-1` computes the sum of their returned values as its own returned value in line 7.

Locus of control

When a `cilk` method is spawned, a ***locus of control*** is created for the method instance, which is more-or-less equivalent to its program counter. When the method returns, its locus of control is destroyed. For instance, in the simple JCilk program from Figure 2-1, the spawning of `A` and `C` in lines 2 and 4 create two new loci of control that can execute `A` and `C` independently from their parent `f2-1`.

A `cilk` method contains only one ***primary*** locus of control. When it calls an ordinary Java (non-`cilk`) method, we view the Java method as executing using the `cilk` method's primary locus of control. In Figure 2-1, for example, the methods `B` and `D` in lines 3 and 5 execute using `f2-1`'s primary locus of control.

JCilk allows ***secondary*** loci of control to be created as well. In particular, when a `cilk` method is spawned, it may return a value that must be incorporated within the parent method in a more involved way than by simple assignment. Figure 2-2 illustrates a program in which the returned values from spawned methods `B` and `C` and called method `D` augment the variable `y`, rather than just assign to it as the return value from `A` does to the variable `x`. Although a child's locus of control normally stays within the child, for circumstances such as those in lines 4 and 5, the child's locus of control operates for a time in its parent `f2-2` to perform the update. JCilk encapsulates these secondary loci of control using a mechanism from the original Cilk language, called an ***inlet***, which is a small piece of code that operates within the parent on behalf of the child. Although Cilk's `inlet` keyword does not find its way into the JCilk language, as we shall see in Section 2.2, the concept of an inlet is used extensively when handling exceptions in JCilk.

```
1   cilk int f2-2() {
2      int x, y = 0;
3      x = spawn A();
4      y += spawn B();
5      y += spawn C();
6      y += D();
7      sync;
8      return x + y;
9   }
```

Figure 2-2: Implicit atomicity allows programmers to reason about multiple JCilk threads operating within the same method.

Implicit atomicity

Since reasoning about race conditions between an inlet and the parent, or between inlets, could be problematic, JCilk supports the idea of ***implicit atomicity***. To understand this concept, we first define a ***JCilk thread***[2] to be a maximal sequence of instructions executed by the same locus of control that includes no parallel control. From a syntactic point of view, a JCilk thread contains no `spawn`, `sync`, or exit from a `cilk` block (`cilk` method or `cilk try`).

For example, the method `f2-1` in Figure 2-1 contains four threads:

1. from the beginning of `f2-1` to the point in line 2 where the `A` computation is actually spawned off,
2. from the point in line 2 where the `A` computation is actually spawned off to the point in line 4 where the `C` computation is actually spawned off,
3. from the point in line 4 where the `C` computation is actually spawned off to the `sync` in line 6,
4. from the `sync` in line 6 to the point where `f2-1` returns.

In Figure 2-2, similar threads can be determined, but in addition, when a spawned method, such as `B` in line 4, returns, an inlet runs the updating of `y` as a separate thread from the others. JCilk's support for implicit atomicity guarantees that all

[2]Although JCilk is implemented using Java's static threads, JCilk threads and Java's static threads are different concepts. Generally, when we say "thread," we mean a JCilk thread. If we mean a Java's static thread, we shall say so explicitly.

JCilk threads executing in the same method instance operate atomically with respect to each other; that is, the instructions of the threads do not interleave. Said more operationally, JCilk's scheduler performs all its actions between thread boundaries, and it executes only one of a method's threads at a time. In the case of `f2-2`, the updates of `y` in lines 4, 5, and 6 are all executed atomically. The updates caused by the returns of `B` and `C` are executed by JCilk's built-in inlets, and the update caused by `D`'s return is executed by `f2-2`'s primary locus of control.

Implicit atomicity places no constraints on the interactions between two JCilk threads in different method instances, however, and it still may be possible to write code with data races within a single method instance. It is the responsibility of the programmer to handle those interactions using synchronized methods, locks, nonblocking synchronization (which can be subtle to implement in Java due to its memory model — see, for example, [19, 31, 42]), and other such techniques. These synchronization issues are not addressed here, because they appear to be orthogonal to the control issues discussed in this thesis.

The `CilkAbort` exception

Because of the havoc that can be caused by aborting computations asynchronously, JCilk leverages the notion of implicit atomicity by ensuring that aborts occur ***semisynchronously***. That means, when a method is aborted, all its loci of control reside at thread boundaries. JCilk provides a built-in exception[3] class `CilkAbort`, which inherits directly from `Throwable`, as do the built-in Java exception classes `Exception` and `Error`. When JCilk determines that a method must be aborted, it causes `CilkAbort` to be thrown in the method. The programmer can choose to catch `CilkAbort` if clean-up is desired, but the exception always appears to have been thrown semisynchronously.

Semisynchronous aborts ease the programmer's task of understanding what happens when the computation is aborted, limiting the reasoning to those points where

[3]In keeping with the usage in [20], when we refer to an exception, we mean any instance of the class `Throwable` or its subclasses.

parallel control must be understood anyway. For example, in Figure 2-1 if `C` throws an exception when `D` is executing, then the thread running `D` will return from `D` and run to the `sync` in line 6 of `f2-2` before possibly being aborted. Since aborts are by their nature nondeterministic, JCilk cannot guarantee that when an exception is thrown, a computation always immediately aborts when its primary locus of control reaches the next thread boundary. What it promises is only that when an abort occurs, the primary locus of control resides at *some* thread boundary, and likewise for secondary loci of control.

2.2 Exception semantics

This section discusses the semantics of JCilk exceptions. The section begins with a simple example of the use of `cilk try` to illustrate two important notions. The first is the concept that a primary locus of control can leave a `cilk try` statement before the statement completes. The second is the idea of a "catchlet," which is an inlet that executes the body of the `catch` clause of a `cilk try`. The section then describes a complete semantics for `cilk try`, and it concludes with an explanation of how the `CilkAbort` exception can be handled by user code.

The `cilk try` statement

Figure 2-3, which shows an example of the use of `cilk try`, demonstrates how this linguistic construct interacts with the spawning of subcomputations. The parent method `f2-3` spawns off the child `cilk` method `A` in line 4, but its primary locus of control continues within the parent, proceeding to spawn off another child `B` in line 9. After spawning off method `B`, the primary locus of control continues in `f2-3` until it hits the `sync` in line 13, at which point `f2-3` is suspended until the two children complete.

Observe that `f2-3`'s primary locus of control can continue on beyond the scope of the `cilk try` statements even though `A` and `B` may yet throw exceptions. If the primary locus of control were held up at the end of the `cilk try` block, writing a

```
1   cilk int f2-3() {
2      int x, y;
3      cilk try {
4         x = spawn A();
5      } catch(Exception e) {
6         x = 0;
7      }
8      cilk try {
9         y = spawn B();
10     } catch(Exception e) {
11        y = 0;
12     }
13     sync;
14     return x + y;
15  }
```

Figure 2-3: Handling exceptions with `cilk try` when aborting is unnecessary.

`catch` clause would preclude parallelism.

In the code from the figure, if one of the children throws an exception, it is caught by the corresponding `catch` clause. The `catch` clause may execute long after the primary locus of control has left the `cilk try` block, however. As with the example of an inlet updating a local variable in Figure 2-2, if method `A` signals an exception, `A`'s locus of control must operate on `f2-3` to execute the `catch` clause in lines 5–7. This functionality is provided by a ***catchlet***, which is an inlet that runs on the parent (in this case `f2-3`) of the method (in this case `A`) that threw the exception. A catchlet runs only after all loci of control executing in the corresponding `cilk try` block have completed. As with ordinary inlets, JCilk guarantees that the catchlet runs atomically with respect to other loci of control running on `f2-3`.

If the `cilk try` statement contains a `finally` clause, a ***finallet***, which is similar to a catchlet, runs only after all loci of control in the `cilk try` block have completed and after the corresponding catchlet has completed. In addition, a finallet runs atomically with respect to the method's other loci of control.

Aborting side computations

We are almost ready to tackle the full semantics of `cilk try`, which includes the aborting of side computations when an exception is thrown, but we require one key concept from the Java language specification [20, Sec. 11.3]:

> "A statement or expression is ***dynamically enclosed*** by a `catch` clause if it appears within the `try` block of the `try` statement of which the `catch` clause is a part, or if the caller of the statement or expression is dynamically enclosed by the `catch` clause."

In Java code, when an exception is thrown, control is transferred from the code that caused the exception to the nearest `catch` clause of a dynamically enclosing `try` statement that handles the exception.

JCilk faithfully extends these semantics using the notion of "dynamically enclosing" to determine, in a manner consistent with Java's notion of "abrupt completion," which method instances should be aborted. (See the quotation in Section 1.3.) Specifically, when an exception is thrown, JCilk delivers a `CilkAbort` exception semisynchronously to the ***side computations*** of the exception. The side computations include any method that is also dynamically enclosed by the `catch` clause of the `cilk try` statement that handles the exception. The side computations also include the primary locus of control of the method containing that `cilk try` statement if that locus of control still resides in the `cilk try` statement. JCilk thus throws a `CilkAbort` exception at the point of the primary locus of control in that case. Moreover, the `CilkAbort` is not caught within a to-be-aborted `cilk` block until all its children have completed, allowing the side computation to be "unwound" in a structured way from the leaves up.

Figure 2-4 shows a `cilk try` statement. If method `A` throws an exception that is caught by the `catch` clause beginning in line 6, the side computation that is signaled to be aborted includes `B` and any of its descendants, if it has been spawned but hasn't returned. The side computation also includes the primary locus of control for `f2-4`, unless it has already exited the `cilk try` statement. It does not include `C`, which is

```
cilk int f2-4() {
   int x, y, z;
   cilk try {
      x = spawn A();
      y = spawn B();
   } catch(Exception e) {
      x = y = 0;
      handle(e);
   }
   z = spawn C();
   sync;
   return x + y + z;
}
```

Figure 2-4: Handling exceptions with `cilk try` when aborting might be necessary.

not dynamically enclosed by the `cilk try` block.

JCilk makes no guarantees that the `CilkAbort` is thrown quickly or even at all when it signals an exception's side computation to abort. It simply offers a best-effort attempt to do so. In fact, it would be correct for the signaling of a side computation to abort to be implemented as a no-op. Linguistically, the side computations are executed speculatively, and the overall correctness of a programmer's code must not depend on whether the "aborted" methods complete normally or abruptly. As we shall see in Chapter 4, however, JCilk does have a particularly efficient mechanism for signaling side computations to abort.

The semantics of `cilk try`

After an exception is thrown, when and how is it handled? We can decompose exception handling into six actions:

1. An exception is selected to be handled by the `catch` clause of the nearest dynamically enclosing `cilk try` statement that handles the exception.
2. Its side computation is signaled to be aborted.
3. All dynamically enclosed spawned methods complete, either normally or abruptly by dint of Action 2.
4. The primary locus of control for the method exits the `cilk try` block, either

normally or by dint of Action 2.

5. The catchlet associated with the selected exception is run.
6. If the `cilk try` contains a `finally` clause, the associated finallet is run.

These actions operate as follows. If one or more exceptions are thrown, Action 1 selects one of them. Mirroring Java's cascading abrupt completion, all dynamically enclosed `cilk try` statements between the point where the exception is thrown and where it is caught also select the same exception, even though they do not handle it. Action 2 is then initiated to signal the side computation to abort. Action 5 is initiated by Action 1, but it does not run until Actions 3 and 4 complete. Finally, Action 6 is run. If no exception is thrown, Actions 1, 2, and 5 are not run. The only dependency is that Action 6 runs only after both Actions 3 and 4 complete.

We made the decision in JCilk that if multiple concurrent exceptions are thrown to the same `cilk` block, only one is selected to be handled. In particular, if one of these exceptions is a `CilkAbort` exception, the `CilkAbort` exception is selected to be handled. The rationale is that the other exceptions come from side computations, which will be aborted anyway. This decision is consistent with ordinary Java semantics, and it fits in well with the idea of implicit aborting.

The decision to allow the primary locus of control possibly to exit a `cilk try` block with a `finally` clause before the finallet is run reflects the notion that `finally` is generally used to clean up [20, Ch. 11], not to establish a precondition for subsequent execution. Moreover, JCilk does provide a mechanism to ensure that a `finally` clause is executed before the code following the `cilk try` statement: simply place a `sync` statement immediately after the `finally` clause.

Secondary loci of control within loops

When a primary locus of control executes a loop containing a `cilk try` statement, the primary locus of control can exit the `cilk try` block and proceed to the next iteration before its `catch` clause or `finally` clause is run. A secondary locus of control eventually executes the `catch` clause or the `finally` clause as a catchlet or a finallet. As in the Cilk language, this situation requires the programmer to reason

```
 1   cilk int f2-5() {
 2      for(int i=0; i<10; i++) {
 3         int a = 0;
 4         cilk try {
 5            a = spawn A(i);
 6         } finally {
 7            System.out.println("In iteration "
 8                  + i + " A returns " + a);
 9         }
10      }
11      sync;
12   }
```

Figure 2-5: A loop containing a `cilk try` illustrating a race condition.

carefully about the code.

In particular, it is possible to write code with a race condition, such as the one illustrated in Figure 2-5. The programmer is attempting to spawn `A(0)`, `A(1)`, ..., `A(9)` in parallel and print out the values returned for each iteration with the iteration number `i`. Unfortunately, the primary locus of control may change the value of `i` before a given child completes, in which case the secondary locus of control created when the child returns will use the wrong value when it executes the print statement in line 8 in the `finally` clause.

This situation is called a ***data race*** (or, a ***general race***, as defined by Netzer and Miller [37]), which occurs when two threads operating in parallel both access a variable and one modifies it. In this case, `f2-5`'s primary locus of control increments the value of `i` in line 2 in parallel with the secondary locus of control executing the `finally` block which reads `i` in line 8. Whereas JCilk's support for implicit atomicity guarantees that the `finally` block executes atomically with respect to `f2-5`'s primary locus of control, it does not guarantee that data races do not occur. In this case, the data race makes the code incorrect.

The race condition in the code from Figure 2-5 can be fixed by declaring a new loop local variable `icopy`, as shown in Figure 2-6. The only differences between code in Figure 2-5 and Figure 2-6 are the additional declaration of a loop variable, `icopy` in line 4 and the reading of `i` replaced with the reading of `icopy` in line 9. Every time

```
cilk int f2-6() {
   for(int i=0; i<10; i++) {
      int a = 0;
      int icopy = i;
      cilk try {
         a = spawn A(icopy);
      } finally {
         System.out.println("In iteration "
               + icopy + " A returns " + a);
      }
   }
   sync;
}
```

Figure 2-6: JCilk's lexical-scope rule is exploited to fix the race condition from Figure 2-5.

f2-6 iterates its loop, a new copy of variable icopy is created and initialized with the current value of i. When the finally clause executes on behalf of an iteration i, the finally clause reads and prints the corresponding value of icopy as determined by the ***lexical-scope rule*** [5, Sec. 7.4]. The JCilk compiler and runtime system provide an efficient implementation of the lexical-scope rule which avoids creating many extraneous versions of loop variables (see Section 4.3).

Handling aborts

In the original Cilk language, when a side computation is aborted, it essentially just halts and vanishes without giving the programmer any opportunity to clean up partially completed work. JCilk exploits Java's exception semantics to provide a natural way for programmers to handle CilkAbort exceptions.

When JCilk's exception mechanism signals a method in a side computation to abort, it causes a CilkAbort to be thrown semisynchronously within the method. The programmer can catch the CilkAbort exception and restore any modified data structures to a consistent state. As when any exception is thrown, pertinent finally clauses, if any, are also executed.

The code in Figure 2-7 shows how CilkAbort exceptions can be caught. If any of A, B, or C throws an exception while others are still executing, then those others are

```
cilk void f2-7() {
   cilk try {
      spawn A()
   } catch(CilkAbort e) {
      cleanupA();
   }
   cilk try {
      spawn B()
   } catch(CilkAbort e) {
      cleanupB();
   }
   cilk try {
      spawn C()
   } catch(CilkAbort e) {
      cleanupC();
   }
   sync;
}
```

Figure 2-7: Catching `CilkAbort`.

aborted. Any spawned methods that are aborted have their corresponding cleanup method called.

2.3 Examples of exception semantics

To demonstrate some of the JCilk extensions to Java, this section presents two algorithms coded in JCilk, both of which employ speculative computing. The first example is a solution to the "Queens" puzzle. The second example is a parallelized version of the alpha-beta search algorithm [29], which implements the "young brothers wait" [15] strategy.

The Queens problem

The first example illustrates how the so-called Queens puzzle can be programmed in JCilk. The goal of the puzzle is to find a configuration of `n` queens on an `n`-by-`n` chessboard such that no queen attacks another. That is, no two queens occupy the same row, column, or diagonal. Figure 2-8(a) illustrates a legal configuration of queens

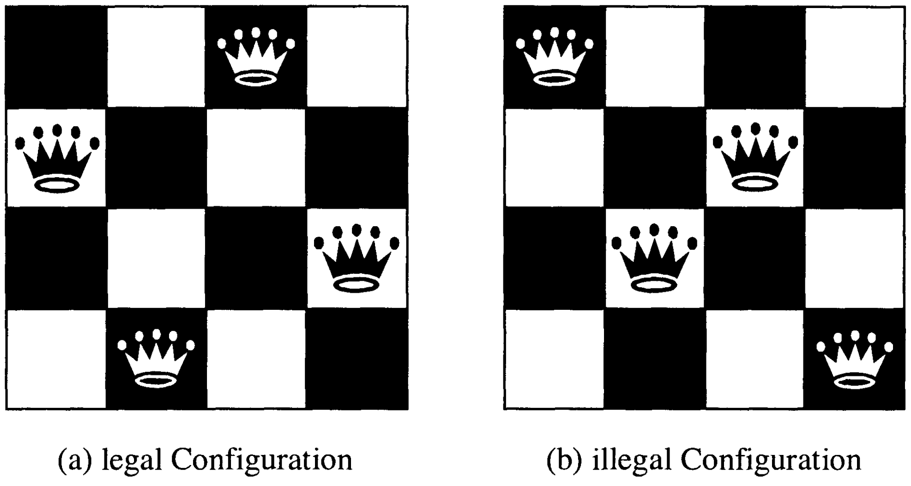

Figure 2-8: Safety of configurations for the Queens puzzle. **(a)** A legal configuration that solves the Queens puzzle. **(b)** An illegal configuration. Both configurations are displayed on a 4-by-4 board. The crown icons represent the placement of the queens.

on a 4-by-4 board. Figure 2-8(b) illustrates an illegal configuration of queens: the queens on column 2 and column 3 are attacking each other diagonally, and similarly, so are the queens on column 1 and column 4.

Figure 2-9 shows how a solution to the Queens puzzle can be programmed in JCilk. The program would be an ordinary Java program if the three keywords `cilk`, `spawn`, and `sync` were elided, but the JCilk semantics make this code a highly parallel program. This implementation uses a speculative parallel search. It spawns many branches in the hopes of finding a "safe" configuration of the `n` queens, and when one branch discovers such a configuration, the others are aborted. JCilk's exception mechanism facilitates programming this strategy.

```
public class Queens {
   private int n;
   ⋮
   private cilk void queen(int[] cfg, int row) throws Result {
      if(row == n) {
         throw new Result(cfg);
      }

      for(int col = 0; col < n; col++) {
         int[] ncfg = new int[n];
         System.arraycopy(cfg, 0, ncfg, 0, n);
         ncfg[row] = col;

         if(safe(row, col, ncfg)) {
            spawn queen(ncfg, row+1);
         }
      }
      sync;
   }

   public static cilk void main(String argv[]) {
      ⋮
      int n = Integer.parseInt(argv[0]);
      int[] cfg = new int[n];
      int[] ans = null;

      cilk try {
         spawn (new Queens(n)).queen(cfg, 0);
      } catch(Result e) {
         ans = (int[]) e.getValue();
      }
      sync;

      if(ans != null) {
         System.out.print("Solution: ");
         for(int i = 0; i < n; i++) {
            System.out.print(ans[i] + " ");
         }
         System.out.print("\n");
      } else {
         System.out.println("No solutions.");
      }
   }
 }
```

Figure 2-9: The Queens problem coded in JCilk. The program searches in parallel for a single solution to the problem of placing `n` queens on an `n`-by-`n` chessboard so that none attacks another. The search quits when any of its parallel branches finds a safe placement. The method `safe` determines whether it is possible to place a new queen on the board in a particular square. The `Result` exception (which inherits from class `Exception`) is used to notify the `main` method when a result is found.

The Queens program works as follows. When the program starts, the `main` method constructs a new instance of the class `Queens` with user input `n` and spawns off its `queen` method to search for a safe configuration. Method `queen` takes in two arguments: `cfg`, which contains the current configuration of queens on the board, and `row`, which contains the current row to be searched. The `queen` method loops through all columns in the current row (lines 7–14) to find safe positions to place a queen in the current row. The ordinary Java method `safe` in line 11, whose definition we omit for simplicity, determines whether placing a queen in row `row` and column `col` conflicts with other queens already placed on the board. If there is no conflict, another `queen` method is spawned to perform the subsearch with the new queen placed in the position (`row`, `col`) in line 12.

The newly spawned subsearch runs in parallel with all other subsearches spawned so far. The parallel search continues until every row contains a queen, at which point `cfg` contains a legal placement of all `n` queens. The successful `queen` method throws the user defined exception `Result` (whose definition is not shown for simplicity) to signal that it has found a solution (lines 4–6). That exception is used as a means of communication between the `queen` and the `main` methods.

The program exploits JCilk's implicit abort semantics to avoid extraneous computation. When one legal placement is found, some outstanding `queen` methods might still be executing; those subsearches are now redundant and should be aborted. The implicit abort mechanism does exactly what we desire when a side computation throws an exception: it automatically aborts all sibling computations and their children dynamically enclosed in the catching `cilk try` block. In this example, since the `Result` exception propagates all the way up to the `main` method, all outstanding `queen` methods are aborted automatically. Notice that there is a `sync` statement (line 26 in the `main` method) before it proceeds to print out the solution to ensure that all side computations have terminated.

Alpha-beta search

We now examine another search algorithm, the parallel minimax search [44]. Like the Queens problem, this algorithm exploits JCilk's exception handling mechanism to abort speculative computations that are found to be unnecessary. In addition, this JCilk program provides an example which exploits the implicit lexical-scope rule to ensure correct execution.

Alpha-beta search [29,50] is often used when programming two-player games such as chess or checkers. It is basically a "minimax" [44] search algorithm applied with "alpha-beta pruning" [44], a technique for pruning out parts of the game tree so that more ply of depth can be searched within a given time bound. Alpha-beta search maintains two values, `alpha` and `beta`, which represent the minimum score that the evaluating player is assured of and the maximum score that the opponent is assured of, respectively. As the algorithm recursively searches down the game tree, the range between `alpha` and `beta` becomes smaller. When `alpha` becomes greater than `beta`, it means that the current player has reached a position where a move can be made so good that the opponent will not make the move leading to the position. Hence, there is no point exploring more moves from that position. This situation is referred as a ***beta cutoff***, and the rest of the search from that position is pruned. The basic alpha-beta search algorithm is inherently serial, because the information from searching one child of a node in the game tree is used to prune subsequent children. It is difficult to use information gained from searching one child to prune another if one wishes to search all children in parallel.

One key observation helps to parallelize alpha-beta search: in an optimal game tree, either all children of a node are searched (the node is ***maximal***), or only one child needs to be searched to generate a cutoff (the node is ***singular***). This observation suggests a parallel search strategy called ***young brothers wait*** [15]: if the first child searched fails to generate a cutoff, speculate that the node is maximal, and thus searching the rest of the children in parallel wastes no work. To implement this strategy, the parallel alpha-beta algorithm first searches what it considers to be the best child. If the score returned by the best child generates a cutoff, the rest of the children are pruned, and the search returns immediately. Otherwise, the algorithm speculates that the node is maximal, and spawns off searches of all the remaining children in parallel. If one of the children returns a score that generates a beta cutoff, however, the other children are aborted, since their work has been rendered unnecessary.

Figure 2-10 shows the core of the parallel alpha-beta search, which is taken from part of a checkers program implemented in JCilk. The algorithm uses the negamax strategy [29], where scores are always viewed from the perspective of the side to move in the game tree. Therefore, when subsequent moves are searched, the `alpha` and `beta` values are reversed, and the scores returned are negated. The `search` method is assumed to be called with the current board configuration, the depth to search, and the `alpha` and `beta` values that bound the search of the current node. When invoked, it first checks for the base case by calling the method `isDone`, which basically returns true if this node is at the leaf of the game tree (meaning the depth is reached), the board configuration is a draw, or one side has lost. (The definition for `isDone` is omitted for simplicity.) If `isDone` returns `true`, the algorithm evaluates and returns the score of the current board configuration. Otherwise, it generates a list, named the `successors`, of legal moves to search, which can be made from the current board configuration. This `successors` list hopefully contains the moves in best-first order.

The search begins from ostensibly the best child, which corresponds to the first move stored in the `successors` list. When this child returns with a score, `alpha` is updated, and the condition for a beta cutoff is checked. If this score generates a beta cutoff (meaning this node is singular), the score for this node (which is stored in `beta` in this case) is returned. If this score does not generate a beta cutoff, the algorithm then proceeds to spawn off the rest of the children in parallel, with the remaining moves stored in the `successors` list. As each of these children returns, the `alpha` value is again updated and the condition for a beta cutoff is checked. If any of these children happens to generate a beta cutoff, a user-defined exception, `ResultException` (whose definition is omitted) is thrown, causing all children

```
private cilk int search(Board board, int depth, int alpha, int beta)
throws ResultException {
   int score1;

   if(isDone(board, depth)) {
      return eval(board);
   }
   List successors = board.legalMoves();
   List move = (List) successors.pop_front();
   Board nextBoard = (Board) board.copy();
   nextBoard.move(move);
   cilk try {
      score1 = spawn search(nextBoard, depth + 1, -beta, -alpha);
   } catch(ResultException e) {
      score1 = e.getValue();
   }
   sync;
   score1 = -score1;
   if(score1 > alpha) {
      alpha = score1;
      if(score1 >= beta) {
         return beta;
      }
   }

   while(mayPlay (successors)) {
      int score2 = -Integer.MAX_VALUE;
      move = (List) successors.pop_front();
      nextBoard = (Board) board.copy();
      nextBoard.move(move);
      cilk try {
         score2 = spawn search(nextBoard, depth + 1, -beta, -alpha);
      } catch( ResultException e) {
         score2 = e.getValue();
      } finally {
         score2 = -score2;
         if(score2 > alpha) {
            alpha = score2;
            if( score2 >= beta ) {
               throw new ResultException(beta);
            }
         }
      }
   }
   sync;

   return alpha;
}
```

Figure 2-10: The core of a parallel alpha-beta search.

spawned in parallel by this node to be aborted. The `ResultException` contains one field, which stores the score of the node so that the score can be communicated back to its parent.

Figure 2-11 shows the code for `rootSearch`, which initiates the searches from the root node. The definition of `rootSearch` is similar to the definition of `search`, except that no checks for beta cutoffs are performed, because no beta cutoff can happen at the root of the game tree. The value for `beta` is initialized to the maximum value that can be represented with an `int` type. One could merge the two methods, `rootSearch` and `search`, with a flag indicating whether the current node is the root node or not. I decided to separate them into two methods for simplicity.

The code shown in Figure 2-10 and Figure 2-11 exploits three important features provided by JCilk:

1. implicit abort semantics,
2. the lexical-scope rule,
3. implicit atomicity.

We examine how each feature is exploited in turn.

First, the code exploits JCilk's implicit abort semantics to abort extraneous computations in a way similar to how the Queens code works, as explained in the Queens problem earlier in this section.

Second, the code exploits JCilk's support for the lexical-scope rule. More specifically, the `search` method in Figure 2-10 contains a `cilk try` with a `finally` clause within the containing loop. The `finally` clause (lines 33–41) refers to the loop local variable `score2`. Since `score2` is declared within the loop (in line 25), the lexical-scope rule applies. When each `finally` clause refers to `score2`, it resolves to the version corresponding to the iteration to which the `finally` belongs lexically. This "correct" resolution of `score2` is crucial to the correctness of the alpha-beta code.

Third, the code exploits JCilk's guarantee of implicit atomicity. In particular, in the same `finally` clause in Figure 2-10 (lines 33–41), an assignment to the local variable `alpha` is made (line 36). Even though `alpha` is written simultaneously

```
private cilk List rootSearch(Board board) {
   List bestMove = null;
   int score1, bestScore = -Integer.MAX_VALUE;
   int alpha = -Integer.MAX_VALUE, beta = Integer.MAX_VALUE;
   List successors = board.legalMoves();
   List move = (List) successors.pop_front();
   Board nextBoard = (Board) board.copy();

   nextBoard.move(move);
   cilk try {
      score1 = spawn search(nextBoard, 1, -beta, -alpha);
   } catch(ResultException e) {
      score1 = e.getValue();
   }
   sync;
   score1 = -score1;
   if(score1 > bestScore) {
      bestScore = score1;
      if(score1 > alpha) {
         alpha = score1;
      }
      bestMove = move;
   }

   while(mayPlay(successors)) {
      int score2 = -Integer.MAX_VALUE;
      move = (List) successors.pop_front();
      nextBoard = (Board) board.copy();
      nextBoard.move(move);
      cilk try {
         score2 = spawn search(nextBoard, 1, -beta, -alpha);
      } catch(ResultException e) {
         score2 = e.getValue();
      }
      score2 = -score2;
      if(score2 > bestScore) {
         bestScore = score2;
         bestMove = move;
         if(score2 > alpha) {
            alpha = score2;
         }
      }
   }
   sync;

   return bestMove;
}
```

Figure 2-11: The root search of parallel alpha-beta.

by multiple secondary loci of control (executing `finally` clauses from different iterations), no data races exist. JCilk's guarantee of implicit atomicity allows all the instantiations of the `finally` clause to execute atomically with respect to one another. As long as the order of their execution does not affect the correctness of the result, no data races exist.

This parallel alpha-beta search demonstrates the expressiveness of JCilk's language features and their semantics. Without the support of any one of these three features, the parallel alpha-beta search could not be programmed so easily. Compared to the parallel alpha-beta search coded in Cilk, this implementation is arguably cleaner and simpler.

Chapter 3

Fundamentals of the JCilk-1 Compiler

The JCilk-1 compiler translates JCilk source code into Java bytecode containing library calls to the runtime system, which schedules threads dynamically according to available system resources. The blueprint of the JCilk-1 compiler is largely based on the implementation of the Cilk-5 compiler. This chapter reviews the concepts JCilk inherits from Cilk and describes how they are expressed in JCilk-1's compiler implementation, including the compilation process and the compilation strategy. In particular, this chapter describes how the JCilk compiler supports the code migration required by the work-stealing algorithm and how the result is returned from a spawned child on one processor to its parent on another processor. The complementary portion of the system, the JCilk-1 runtime system, was mainly implemented by my collaborator, John Danaher. His work is described in his thesis [12], where the readers can find more detail about the implementation of the JCilk-1 runtime system.

3.1 Concepts inherited from Cilk

There are three major concepts that JCilk inherits from Cilk: the "work-first" principle, the "work-stealing" algorithm, and the "two-clone" compilation strategy. The work-first principle is used for allocating overhead costs in the implementation. The work-stealing algorithm is used by the scheduler. The two-clone compilation strategy is used by the compiler. This section outlines these key concepts. For a more complete presentation, see [8, 16].

The work-first principle

The ***work-first principle*** [16] states:

> "Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path."

Two important parameters dictate the performance of a Cilk computation: its ***work***, which is the execution time of the computation on one processor, and its ***critical-path length***, which is the execution time of the computation on an infinite number of processors. In a dag representation of a Cilk computation, one can think of the work as the sum of the execution times of all logical threads, and the critical path length as the longest path between the first node (the initial thread) and the last node (the last thread).

With these two parameters, we can give two fundamental lower bounds as to how fast a Cilk program can run. Let us denote the execution of a given computation on P processors as T_P. Then, the work of the computation is T_1, and the critical-path length is T_∞. The first lower bound is $T_P \geq T_1/P$, because at each time step, at most P units of work can be executed, and the total work is T_1. The second lower bound is $T_P \geq T_\infty$, because a finite number of processors cannot execute faster than an infinite number of processors. Assuming an ideal parallel computer, Cilk's work-stealing scheduler executes in expected time

$$T_P = T_1/P + O(T_\infty). \tag{3.1}$$

We refer to the first term on the right hand side of Equation 3.1 as the ***work term***, and the second term as the ***critical-path term***. The important point is that, all the costs induced by communication and scheduling are borne by the critical-path term.

To make these overheads explicit, we define the ***critical-path overhead*** to be the smallest constant c_∞ such that

$$T_P \leq T_1/P + c_\infty T_\infty. \tag{3.2}$$

Define the ***(average) parallelism*** as $\overline{P} = T_1/T_\infty$, which corresponds to the maximum possible speedup that the application can obtain. Also define the ***parallel slackness*** to be the ratio $\overline{P}/P$. Assuming sufficient parallel slackness, meaning $\overline{P}/P \gg c_\infty$, then it follows that $T_1/P \gg c_\infty T_\infty$. Hence, from Inequality 3.2, we obtain that $T_P \approx T_1/P$, which means that we achieve linear speedup when the number of processors, P is much smaller than the average parallelism $\overline{P}$. Thus, when sufficient parallel slackness exists, the critical-path overhead has little effect on performance.

As mentioned in Section 1.2, for every Cilk program, there is a legal C elision, which one can use to measure against the 1-processor performance of the Cilk program. Define the ***work overhead*** to be $c_1 = T_1/T_s$, where T_s is the execution time of the computation by the C elision. Then, incorporating the critical-path and work overheads into Inequality 3.2 yields

$$\begin{aligned} T_P &\leq c_1 T_s/P + O(T_\infty) \qquad (3.3) \\ &\approx c_1 T_s/P, \end{aligned}$$

assuming the parallel slackness is sufficient (which is the common case). To restate the work-first principle more precisely, *minimize c_1, even at the expense of a larger* c_∞, because c_1 has a larger impact on performance.

The work-first principle pervades the implementation of Cilk-5. Specifically, the implementation of Cilk's work-stealing algorithm and compiler are both designed to minimize the work overhead, sometimes at the cost of a larger critical-path overhead. JCilk's implementations follow the same strategy.

The work-stealing algorithm

Cilk's runtime system uses a work-stealing algorithm to schedule threads dynamically according to the available system resources. The basic idea of work-stealing is that if a processor runs out of work, it takes the initiative and attempts to "steal" work from other processors. If most processors have work, the migration of threads happens infrequently, thereby avoiding associated overheads.

Cilk's work-stealing algorithm operates as follows. A collection of POSIX threads [26], called ***workers***, schedule and execute the logical threads. Each worker maintains a ***ready deque*** (doubly-ended queue), which mirrors C's call stack and contains procedure instances ready to be executed. These procedure instances are represented as the ***activation frames***, each of which contains information required to resume execution of a particular `cilk` procedure. Every ready deque has two ends, a ***head*** and a ***tail***, from which activation frames can be added or removed. A worker operates locally on the tail of its own deque, treating it much as C treats its call stack, pushing and popping spawned activation frames. When a worker runs out of work to do, it becomes a ***thief*** and attempts to steal an activation frame from another worker, called its ***victim***. The thief steals the activation frame from the head of the victim's deque, which is the opposite end from which the victim is working. Thus, when the thief steals from the head of the deque, the activation frame stolen is always the oldest one in the deque. In a dag representation of a Cilk computation, one can think of the worker as operating in a depth-first fashion and the thief as operating in a breath-first fashion.

Figure 3-1 shows a simple example of work-stealing. Initially, no workers have work to do except for worker $W1$, which has two activation frames in its deque, representing procedures `A` and `B`. Since worker $W2$ has no work to do, it randomly chooses $W1$ to be its victim and attempts to steal from $W1$. At this point, $W1$ is busy working on `B`, and `A` (which spawned off `B`) is available to be stolen. $W2$ steals activation frame for `A` from $W1$ and starts working on it by resuming the execution from where $W1$ left off, which in this case, is the instruction right after spawning

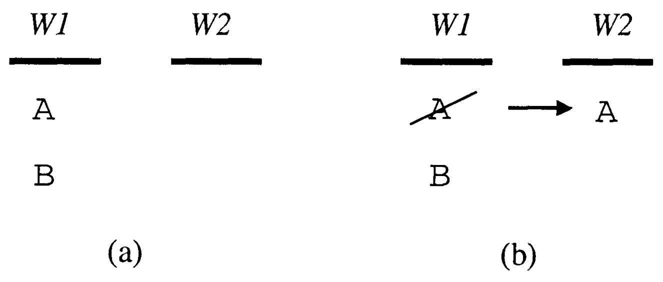

Figure 3-1: An example of work-stealing. **(a)** The ready deques of two workers $W1$ and $W2$. **(b)** The ready deques after the steal. `A` and `B` are method instances ready to be executed. Method `A` is stolen by $W2$ in (b).

of `B`. Later, when $W1$ finishes working on `B`, it attempts to return control back to `B`'s parent `A`. Since `A` has been stolen and is no longer in $W1$'s deque, instead of doing an ordinary return, $W1$ notifies `A`'s current worker that its subcomputation has completed, and passes `B`'s result to `A`'s current worker. Instead of saying $W2$, I say `A`'s current worker, because it is possible for `A`'s current worker to be some worker other than $W2$, for example, if `A` spawns off another procedure and is stolen again.

As the work-first principle suggests, one should move most overhead from the worker to the thief, because actions by the worker contribute to the work term, whereas actions by the thief contribute heavily to the critical-path term and less to the work term. The Cilk's scheduler implements the "THE" protocol [16] which employs the Dijkstra's protocol for mutual exclusion [14]. The key idea is that, to resolve conflicts between two thieves stealing from the same victim, a simple but more heavyweight lock is used, because this overhead contributes to the critical path term. To resolve conflicts between a worker and a thief when there is only one frame left in the ready deque, a more complicated but lightweight Dijkstra-like protocol is used, and this overhead contributes minimally to the work term.

The two-clone compilation strategy

Cilk's compiler implements a ***two-clone*** compilation strategy, designed to reduce the work overhead, as dictated by the work-first principle. The strategy is to make two clones of each `cilk` procedure — a ***fast*** clone and a ***slow*** clone. The fast clone operates just like the C elision and has little support for parallelism, except for a small amount of essential bookkeeping. The slow clone has full support for parallelism, along with its concomitant overhead.

Whenever a procedure is spawned, the fast clone runs. Whenever a procedure is stolen, however, the stolen procedure is converted into a slow clone. When a steal happens, we are making progress on the critical path, and hence the overhead of executing a slow clone is borne by the critical-path term. In addition, since the Cilk scheduler guarantees that the number of steals is small when sufficient slackness exists, fast clones are executed most of the time. Therefore, to minimize the work overhead, we should try to minimize the cost of executing fast clones.

The structure of the Cilk scheduler maintains two invariants which allow us to minimize overhead in fast clones. First, a fast clone has never been stolen, because whenever a procedure is stolen, it is converted into a slow clone. Second, the fast clone's descendants are also fast clones and have never been stolen, because the strategy of stealing from the heads of ready deques guarantees that parents are always stolen before their children. These facts allow many optimizations to be performed in fast clones.

Since the fast clone has never been stolen, it does not need to provide general support for parallelism, except for a small amount of essential bookkeeping. A `spawn` statement in the fast clone is simply converted into an ordinary method call, and the activation frame of the executing procedure is pushed into the ready deque (via a call to the runtime system). The bookkeeping required is simply filling the activation frames with the program counter and the current local variable values. A `sync` statement is converted into an empty statement, because when a fast clone is executing, it has no extant children.

The slow clone is similar to the fast clone except that it provides support for parallel execution, since when a slow clone is executed, it must have been stolen and is being executed in parallel with its children. The thread atomicity in Cilk guarantees that a procedure can only be stolen between thread boundaries. Hence, a slow clone always resumes at a thread boundary. In Cilk, this "continuation" is accomplished

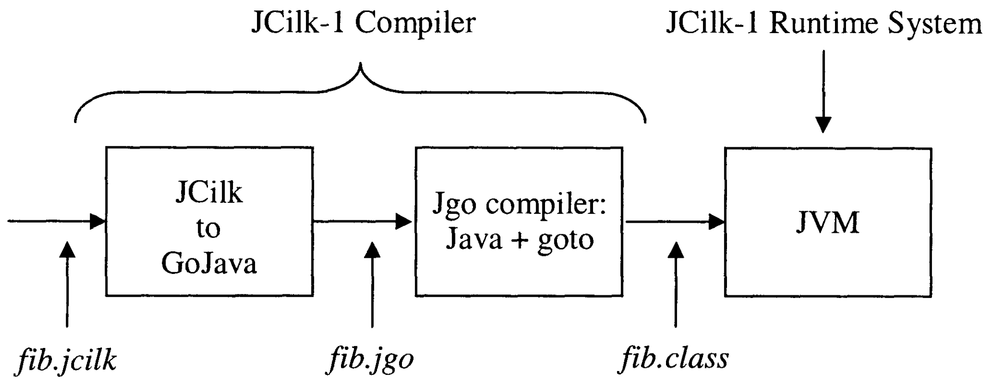

Figure 3-2: The two-stage compilation process for JCilk programs.

by using a goto statement to jump to the point of resumption, restoring the values of local variables, and resuming execution. The slow clone knows where to resume and how to restore the values of local variables, because they are recorded in the activation frame by the fast clone.

3.2 JCilk-1's two-stage compilation process

JCilk follows the same concept as Cilk and implements the two-clone compilation strategy to minimize the work overhead. The implementation is not as straightforward as in Cilk, however. This section explains what complications I encountered when implementing the JCilk compiler and how I solved them. To be specific, the JCilk compiler compiles code in two stages. The first stage compiles a given JCilk program into GoJava, an intermediate language that I created. The second stage then compiles the GoJava code into Java bytecode that can be executed by any standard Java virtual machine. This two-stage compilation process allowed me to implement the "continuation" mechanism required by the work-stealing algorithm.

Support for continuations

The work-stealing algorithm requires the compiler to support the migration of code between worker threads. Remember from Section 2.1 that the JCilk keywords define the boundaries of JCilk threads. These boundaries are points where a migration of code can potentially happen; that is, the two JCilk threads separated by a boundary may be executed by two different workers. When a migration occurs during a steal, the thief must establish its primary locus of control at the point where the victim left off, as well as gain access to the state of local variables at that point. The mechanism to allow this resumption is called a ***continuation***.

The Cilk system, which is implemented in C, supports a continuation mechanism using `goto` statements. Cilk's compiler does source-to-source translation, translating only Cilk keywords while leaving ordinary C code intact. The generated C code has the same structure as the original Cilk program, but the keywords `cilk`, `spawn`, and `sync` are expanded into the C statements and runtime-system calls necessary to accomplish their functionalities. The `goto` statements are inserted at places where a continuation needs to happen, allowing the control of program to jump from the beginning of a `cilk` procedure to any point where the continuing logical thread begins.

Ideally, we would have liked to follow the Cilk strategy of performing a JCilk-to-Java translation, translating only JCilk keywords while leaving ordinary Java code intact. This approach is consistent with the philosophy that a user should pay the overhead of running parallel code only when using JCilk's parallel extensions. A JCilk program should be compiled so that blocks containing only ordinary Java code run without slowdown. Unfortunately, adopting this approach for JCilk is problematic, because the Java language provides no `goto` statement.

To support a continuation mechanism, I created an intermediate language called GoJava. This language is a minimal extension of Java that allows the `goto` statement in limited and specific circumstances. Since Java bytecode already contains jump instructions, compiling a GoJava program into standard Java bytecode required minimal changes to a Java compiler. Once the support for continuations was present, the rest of the implementation issues were relatively straightforward.

The eventual design for the JCilk's compiler contains two stages, illustrated in Figure 3-2. The first stage is a source-to-source translation from JCilk to GoJava. This stage expands all JCilk keywords into their effects and leaves ordinary Java code unaffected. This translation is performed using Polyglot [40], a compiler toolkit designed especially for Java language extensions. The second stage of the compilation process translates GoJava to Java bytecode using Jgo, a compiler for GoJava which I created by minimally modifying GCJ (the Gnu Compiler for Java) [18]. This second stage adds no additional overhead as compared to using GCJ directly, maintaining the property that pure Java code suffers no slowdown.

The advantages of the two-stage approach

When designing the compilation process, we considered other alternatives, but in comparison, the two-stage approach gives us better extensibility and portability. Not only does this approach add up to a better research platform, it requires considerably less engineering effort. There are two main alternatives that we considered.

The first alternative that we considered was to modify an existing Java compiler, such as GCJ (Gnu Compiler for Java [18]) to translate from JCilk to bytecode directly. This approach is not ideal because the JCilk compiler would be highly dependent on the GCJ release. The resulting compiler would have intermixed my code with the GCJ source and would have required reimplementing JCilk for every new release of GCJ in order to keep up-to-date. Not long ago, GCC (Gnu Compiler Collection which includes GCJ) made a new release and reengineered the internal representation used in its compiler collection. Bringing the compiler current would have caused a JCilk compiler reimplementation if I had implemented the compiler solely based on modifying GCJ. Also, since GCJ itself is a complex piece of software, the resulting JCilk compiler would be more complicated and difficult to maintain or extend than the one based on Polyglot.

One other alternative that we considered was to modify an existing open source Java virtual machine and support continuation as a primitive within the virtual machine. This approach requires substantial engineering effort and has similar problems as the first alternative. Even though most of the existing open source virtual machines do not change versions quickly (and thus are easier to maintain), they do not support the latest version of JDK (Kaffe [1] or LaTTe [36]). Hence, keeping the resulting compiler up-to-date would still be difficult.

The two-stage compilation strategy gives us several advantages over other strategies: less engineering effort, cleaner implementation, relative portability, and easy extensibility. The two-stage compilation strategy has a clean separation between the translation part (the JCilk-to-GoJava translation) and the compilation part (the GoJava-to-bytecode compilation). The result is a clean implementation, since all translation required is done in Polyglot, translating JCilk keywords only. Polyglot provides us a clean interface in which to build almost anything one would want in a compiler for a Java language extension. The JCilk compiler is relatively portable, because Polyglot is implemented in Java and is platform independent. Finally, it is also easily extensible, because making new extensions simply means implementing them in Polyglot. These advantages sum up to give us a better research platform than other alternatives.

JCilk's use of `goto` statements is safe

Putting `goto` statements into Java often raises people's eyebrows and concerns: does it violate Java's safety properties? The short answer is no. The strongest evidence is that the bytecode generated by the JCilk compiler passes Java's bytecode verification and can be executed by any standard Java virtual machine. Safety in Java is enforced at the bytecode level by the JVM interpreter. Since the JCilk compiler produces legal Java bytecode, even though it does so through the unorthodox means of GoJava, no safety properties are sacrificed.

To give a longer answer, JCilk's use of `goto` statements is safe, because the JCilk compiler only inserts them in limited and specific circumstances. The standard Java virtual machine does provide jump instructions at the bytecode level, and in fact, one of the jump instructions is called `goto`, which does an unconditional jump. Therefore, using the `goto` bytecode instruction is indeed legal, as long as it does not violate the constraints on Java bytecode [32, Sec. 4.8]. More specifically, there are two types of ***bytecode constraints***: ***static constraints*** [32, Sec. 4.8.1] and ***structural constraints*** [32, Sec. 4.8.2]. The JCilk compiler follows both constraints strictly.

With respect to static constraints, the Java Virtual Machine Specification [32, pg. 134] states,

> "The target of each jump and branch instruction (*jsr*, *jsr_w*, *goto*, *goto_w*, *ifeq*, *ifne*, *ifle*, *iflt*, *ifge*, *ifgt*, *ifnull*, *ifnonnull*, *ificmpeq*, *ificmpne*, *ificmple*, *ificmplt*, *ificmpge*, *ificmpgt*, *ifacmpeq*, *ifacmpne*) must be the opcode of an instruction within this method. The target of a jump or branch instruction must never be the opcode used to specify the operation to be modified by a *wide* instruction; a jump or branch target may be the *wide* instruction itself."

JCilk's use of `goto` statements follows this static constraint. First, JCilk only uses `goto` statements to make local jumps, meaning jumps within the same method. Second, since the JCilk compiler inserts `goto` statements at the Java source-code level, the `goto` bytecode can only be inserted between two Java statements and never in the middle of a statement. In other words, JCilk's use of `goto` statements ensure that the target of the `goto` instruction is never the opcode used to specify the operation to be modified by a *wide* instruction. (The *wide* instruction modifies the behavior of another instruction, which can be generated only as part of a statement but never as a whole statement [32, pp.360-361].)

With respect to structural constraints, the Java Virtual Machine Specification [32, pg. 137] states,

> "No local variable (or local variable pair, in the case of a value of type `long` or `double`) can be accessed before it is assigned a value."

JCilk's use of `goto` statements follows this structural constraint as well. The JCilk compiler ensures that all variables are properly initialized with the correct values at the jump destination. Consequently, its use of `goto` statements never causes errors such as uninitialized local variables.

The Jgo compiler is designed to be a component of the JCilk compiler, and normally, it should only be used to compile GoJava code output from the first stage of JCilk compilation. When the Jgo compiler is used as part of the JCilk compilation process, the bytecode it generates is safe and conforms to all bytecode constraints on the use of `goto` instructions. These constraints are enforced by JCilk's first compilation stage when translating from JCilk to GoJava, but not by the second compilation stage when compiling from GoJava to bytecode, where Jgo is used.

If GoJava were to be used as a stand-alone programming language, however, no mechanism would enforce a programmer to obey these bytecode constraints, because the Jgo compiler imposes no restrictions on the use of `goto` statements. Consequently, the Jgo compiler should not be used as a general-purpose compiler. Otherwise, it would be too easy for a programmer to use a `goto` statement improperly.

When building the JCilk compiler, we decided to implement GoJava as an intermediate language rather than a general-purpose language. This decision was made for two reasons. First, we did not need GoJava to be a general-purpose language for the use of JCilk. JCilk only needs to insert `goto` statements in limited circumstances. Second, it is not straightforward for a programmer to reason about the proper usage of `goto` statements due to the subtleties of Java's bytecode constraints, which are not apparent at the Java source-code level. Bringing `goto` statements into Java for general use would introduce unnecessary complexities into the language.

To demonstrate why the Jgo compiler should not be used as a general-purpose compiler, consider the following bytecode constraint. The Java Virtual Machine Specification [32, pg. 137] states,

> "If an instruction can be executed along several different execution paths, the operand stack must have the same depth (§3.6.2) prior to the execution of the instruction, regardless of the path taken."

It is easy to write code that violates this constraint and causes the Jgo compiler to generate bytecode that is rejected by Java's bytecode verifier. For instance, a `finally` clause is usually invoked with the bytecode instruction *jsr*, which pushes its return address when executed [32, pg. 149]. That means, when executing the first instruction in a `finally` clause, the return address is expected to be on the top of the stack, regardless the execution path taken to reach the instruction. The constraint is violated if a `goto` statement is used to jump into the middle of a `finally` clause, because a `goto` statement does an unconditional jump and does not push the return address onto the stack before the jump. Hence, the stack depth can differ depending on how the program control reaches the `finally` clause. While it is unsafe to use the Jgo as a general-purpose compiler, the JCilk compiler is still safe, because JCilk semantics simply do not require the use of `goto` statements in such cases. That is, the first stage of the JCilk compiler never needs to generate GoJava that uses `goto` statements that violate this constraint. Hence, the JCilk compiler enforces this constraint automatically.

3.3 Compiler support for `spawn` and `sync`

With the knowledge of how work-stealing works and how JCilk compiler supports continuations, this section describes in more detail how the JCilk compiler interacts with the runtime system to support the basic semantics for `spawn` and `sync`. This section answers questions such as how a method instance is represented in a ready deque, how the slow clone of a `cilk` method supports parallelism, and how the inlet mechanism is implemented in JCilk.

Interacting with the JCilk runtime system

The JCilk compiler interacts with the JCilk runtime system via two classes: the `Worker` class and the `CilkFrame` class. The `Worker` class represents the workers in the work-stealing algorithm. The `CilkFrame` class represents the activation frames in the ready deque.

An instance of the `Worker` class acts as a worker described in the work-stealing algorithm from Section 3.1. The class implements library calls that the compiler invokes to communicate with the runtime system. Although the runtime system schedules the JCilk threads and manages the ready deque for each worker, the JCilk compiler generates calls in the GoJava output to trigger actions to operate on the ready deque.

An instance of the `CilkFrame` class acts as an activation frame in a worker's ready deque. The class is, however, a superclass for all activation frames and provides only an abstraction so that the worker can use it without knowing all the details of a particular method instance. Since the activation frame contains fields pertinent to the `cilk` method which it represents, each `cilk` method has its own type of activation frame. An activation frame for a particular `cilk` method is implemented as a private inner class (or nested class if the `cilk` method is static) which extends the `CilkFrame` class in the GoJava output generated by the compiler. Each activation frame contains fields mirroring all local variables of its corresponding `cilk` method.

To allow a thief to resume execution of the stolen activation frame, the thief needs to know which method the frame represents and where the execution was left off, that is, the latest ***program counter*** for the method execution. The `CilkFrame` provides the abstractions (overridden by each activation frame accordingly) so that the thief can retrieve the information from the activation frame it steals, but at the same time be oblivious about the precise fields contained in the frame. The thief figures out where to resume by reading a field called `pc` in the `CilkFrame`, which assimilates the latest program counter of the method. The thief resumes the correct method by calling the `cilkRun` instant method in the `CilkFrame`. The `cilkRun` method is overridden by each activation frame to invoke the appropriate slow clone of its corresponding `cilk` method. For brevity, when I say a method is defined in the `CilkFrame`, the readers can automatically assume that the method is overridden appropriately by each activation frame (its subclass), tailored specifically for the frame's corresponding `cilk` method.

The `cilk` method, `fib` in Figure 3-3 calculates the Fibonacci number of `n` by spawning off the calculation for `fib(n-1)` and `fib(n-2)` in parallel and returning their sum after the `sync` statement. Its corresponding activation frame `Fib_Frame` is shown in Figure 3-4. The `pc` field is inherited from `CilkFrame`, since it is needed across all frames. The `Fib_Frame` object contains other fields, where each corresponds to a local variable in the `fib` method, including the method argument `n`. The `cilkRun` method, invoked by the worker from the runtime system to resume execution of a stolen method, is overridden to call the slow clone, which is `fib_Slow` in this case.

```
cilk private int fib(int n) {
   int x, y;
   if( n < 2 ) {
      return n;
   }
   x = spawn fib(n-1);
   y = spawn fib(n-2);
   sync;
   return (x + y);
}
```

Figure 3-3: A simple JCilk method that computes the Fibonacci number using a naive exponential-time algorithm.

```
private class Fib_Frame extends CilkFrame {
   private int n;
   private int x;
   private int y;

   public Fib_Frame(int n, int pc, int returnEntry) {
      super(pc, returnEntry);
      this.n = n;
   }

   public void cilkRun(Worker worker) {
      fib_Slow(worker, this);
   }
}
```

Figure 3-4: A simplified version of the activation frame for the Fibonacci method shown in Figure 3-3.

JCilk's two-clone compilation

JCilk's compiler, following the design of Cilk's compiler, implements the two-clone compilation strategy to minimize the work overhead. Figure 3-5 and Figure 3-6 illustrate the two clones in the GoJava output produced when the `fib` method is compiled.

Figure 3-5 shows the simplified GoJava output of `fib`'s fast clone. The method body structure remains the same except that a few statements for bookkeeping are added. First, at the beginning of the method, a call is made to push the `Fib_Frame`

```
 1  private int fib(int n, Worker worker, int returnEntry) {
 2     int x, y;

 3     Fib_Frame frame = new Fib_Frame(n, 0, returnEntry);
 4     worker.pushFrame(frame);
 5     if(n < 2) {
 6        return n;
 7     }

 8     frame.pc = 1;
 9     x = fib(n - 1, worker, 1);
10     if(worker.popFrameCheckStart() == worker.STOLEN) {
11        worker.popFrameCheckFinish(new Integer(x), true);
12        return 0;
13     }
14
15     frame.x = x;
16     frame.pc = 2;
17     y = fib(n - 2, worker, 2);
18     if(worker.popFrameCheckStart() == worker.STOLEN) {
19        worker.popFrameCheckFinish(new Integer(y), true);
20        return 0;
21     }
22
23     return x + y;
24  }
```

Figure 3-5: A simplified version of the fast clone of the compiled Fibonacci method.

into the ready deque of the worker (lines 3–4). The first `spawn` statement from the original JCilk program (line 6 in Figure 3-3) has been transformed into several statements in the fast clone (lines 8–13 in Figure 3-5). Before the call, the `pc` field is set, to keep track of the current locus of control, in case the `fib` method is stolen. After returning from the first call to `fib(n-1)`, a call is made to the worker in line 10 to see if `fib` has been stolen. If so, hand the return value to the runtime system and leave the method. If not, continue onto the next statement. Again, since this is the fast clone and has never been stolen, there is no need to execute a `sync` statement. The `sync` statement from the original JCilk program has been converted into an empty statement.

Figure 3-6 shows the simplified GoJava output of `fib`'s slow clone. The method

```
 1  private void fib_Slow(Worker worker, CilkFrame frame) {
 2     Fib_Frame fibFrame = (Fib_Frame) frame;
 3     switch(fibFrame.pc) {
 4        case 1:
 5           goto cilk_sync1;
 6        case 2:
 7           goto cilk_sync2;
 8        case 3:
 9           goto cilk_sync3;
10     }
        ⋮
11     fibFrame.pc = 1;
12     fibFrame.x = fib(fibFrame.n - 1, worker, 1);
13     if( worker.popFrameCheckStart() == worker.STOLEN ) {
14        worker.popFrameCheckFinish( new Integer(fibFrame.x), true );
15        return;
16     }
17     cilk_sync1: ;

18     fibFrame.pc = 2;
19     fibFrame.y = fib(fibFrame.n - 2, worker, 2);
20     if( worker.popFrameCheckStart() == worker.STOLEN ) {
21        worker.popFrameCheckFinish( new Integer(fibFrame.y), true );
22        return;
23     }
24     cilk_sync2: ;

25     fibFrame.pc = 3;
26     if( worker.checkSync() == false ) {
27        return;
28     }
29     cilk_sync3: ;

30     worker.setResult( new Integer(fibFrame.x + fibFrame.y) );
31     return;
32  }
```

Figure 3-6: A simplified version of the slow clone of the compiled Fibonacci method.

is similar to the fast clone except that it contains support for parallel execution. A switch statement is added at the beginning of the method (lines 3–10) to allow the thief who stole this method to jump to the appropriate continuation point, which is specified by the pc field in the frame. Labels are added at all possible continuation points, meaning after every `spawn` and `sync` statement. For example, if the thief steals the method after the victim spawns off `fib(n-1)`, the `pc` read by the thief would be 1, causing the thief to jump from line 5 to line 17, where the spawning of `fib(n-1)` ends. The `sync` statement from the original JCilk program is converted into several statements in the slow clone (lines 25–29 in Figure 3-6). A call is made to the worker in line 26 to ensure that all spawned children have returned. If not, the method returns to the runtime and will be resumed sometime later when all children return. When returning a value, the slow clone does not use the ordinary `return` statement in Java. Instead, it hands the return value to the worker (as in line 30). Since the slow clone is always stolen and invoked by the runtime via the `cilkRun` method, its actual parent method is not in the same Java call stack, and special care must be taken to return the slow clone's value to its parent.

Returning a value to a stolen parent

What happens when a spawned child completes execution and tries to return a value to its parent, but discovers that its parent has been stolen? In this case, the parent's frame is no longer in the same Java call stack and is being executed by a different worker. Instead of executing an ordinary Java return, the child must somehow find its way to return the value to the stolen parent. The inlet mechanism, described in Chapter 2, allows the child to return to a parent on a different worker.

Since the parent is stolen, it is being executed as a slow clone, which reads values from its activation frame when referring to local variables. Thus, the child can simply return by updating the parent's activation frame with its return value. But, how does the child know which field in the frame to update? This issue does not arise in Cilk — a spawned child in Cilk is simply handed the memory address of the variable into which the return value should be stored. We cannot play the same trick in JCilk, however, because Java provides no explicit way to refer to a memory address.

We solve this problem by passing the current `pc` of the parent when spawning a child. This value is referred to as the child's ***return entry***, which corresponds to where the parent's locus of control is when it spawns off the child. The return

```
 1  public void setInletReturn(int returnEntry, Object returnVal) {
 2     switch (returnEntry) {
 3        case 1:
 4           this.x = ((Integer) returnVal).intValue();
 5           break;
 6        case 2:
 7           this.y = ((Integer) returnVal).intValue();
 8           break;
 9     }
10  }
```

Figure 3-7: The `setInletReturn` method that implements the inlet mechanism for the Fibonacci method from Figure 3-3

entry of a child is stored in its own frame, in the field called `returnEntry` defined in `CilkFrame`. When the child returns, it hands over the return entry along with the return value to the current worker of the parent, which then updates the appropriate field in the parent's frame with the return value. A lookup table in the parent's frame tells which field in the frame a return entry corresponds to and dispatches to the appropriate inlet to update the field. This lookup table is invoked via the method `setInletReturn` defined in `CilkFrame`. As specified by the semantics, the runtime system ensures that the inlet code is executed between thread boundaries only.

Figure 3-7 illustrates how the `setInletReturn` for the `fib` method (Figure 3-3) is defined. When spawning off `fib(n-1)` in the compiled code (line 9 in Figure 3-5), the current `pc` is handed to the child as an argument. This value is stored in the `returnEntry` field in child's frame and is used to return when invoking `setInletReturn` in the parent frame. In this case, `returnEntry` of value 1 corresponds to returning to `x` in the frame, as shown in line 4, Figure 3-7. Even though the inlet mechanism produces a small overhead compared to an ordinary Java return, this mechanism is used only in the slow clone when the method is stolen. Hence, the overhead is borne by the critical-path term, as dictated by the work-first principle.

Chapter 4

Compiler support for exception semantics

This chapter describes how the JCilk compiler supports semantics unique to JCilk, that is, those that are not analogous to linguistics in Cilk. These semantics specifically include the exception semantics presented in Section 2.2. The JCilk compiler provides a mechanism for supporting catchlets and finallets, and it enforces the lexical-scope rule. In addition, the JCilk compiler implements a special mechanism to enforce Java's left-to-right order of evaluation. (The lexical-scope rule and the order of evaluation are not yet fully implemented in the JCilk-1 compiler.)

This chapter is organized as follows. Section 4.1 describes how the compiler supports the mechanisms for catchlets and finallets by generating "lookup-tables" methods which provides information of the lexical hierarchy of nested `cilk try` blocks. Section 4.2 presents the mechanism for supporting implicit abort, which replies on a data structure called "try tree" maintained by the runtime system and a lookup-table method generated by the compiler. Section 4.3 explains the implementation for enforcing the lexical-scope rule. Section 4.4 examines the evaluation order required by the Java semantics and the complications entailed for the compiler to enforce the evaluation order.

```
private int fib(int n, Worker worker, int returnEntry) {
    ⋮
   try {
      x = fib(n - 1, worker, 1);
   }
   catch(RuntimeException e) {
      if(worker.popFrameCheckException(e) == worker.STOLEN) {
         return 0;
      } else {
         throw e;
      }
   }
    ⋮
   if(worker.popFrameCheckStart() == worker.STOLEN) {
      worker.popFrameCheckFinish(new Integer(x), true);
      return 0;
   }
    ⋮
}
```

Figure 4-1: The fast clone of the compiled Fibonacci method, with support for intercepting exceptions.

4.1 Support for exception handling

This section describes how the compiler supports exception handling in JCilk: the catchlet and the finallet mechanism. In addition to regular return values, when a spawned child throws an exception to a stolen parent, more complications are involved for the parent method to handle the exception correctly. In particular, if the throwing child is buried in multiple levels of nested `cilk try` blocks, which `catch` clause handles the exception and which `finally` clause needs to be executed depend on the type of the exception thrown. The compiler generates "lookup-table" methods to help the runtime system make these decisions.

Intercepting exceptions

What happens when a child throws an exception and the parent is stolen? Similar to the returning situation, since the parent's frame is no longer in the same

Java call stack, the child cannot throw the exception normally. Rather, the worker must intercept the exception and pass the exception to the stolen parent appropriately. Figure 4-1 illustrates how the exception can be intercepted in the compiled `fib` method. The JCilk compiler generates a `try` statement (lines 2–11) to wrap around a call to a `cilk` method to catch every exception that the `cilk` method can possibly throw. The exceptions to intercept include both the unchecked exceptions and the checked exceptions declared to be thrown by the called `cilk` method. In the case of `fib`, the `try` statement contains `catch` clauses for `RuntimeException`, `Error`, and `CilkAbort`. (The `CilkAbort` exception is declared to be thrown by every `cilk` method, although it is not shown in Figure 4-1. The programmer can consider `CilkAbort` as an unchecked exception.) If the parent is stolen when an exception occurs, the child returns to the runtime (lines 6–7). Otherwise, the exception is rethrown, allowing the Java mechanism to handle it naturally (line 9).

Executing catchlets

Intercepting and passing the exception is not the difficult part — simply treat the exception as a return value and pass it to the parent's current worker. The difficult part is figuring out where the exception is caught in order to determine which catchlet and finallet(s) (if any) to execute. As an example, consider the method in Figure 4-2. There are three spawns in the method `threeWay`: method `A` is not enclosed by any `cilk try` statement; method `B` is enclosed by one `cilk try` statement; method `C` is enclosed by two nested `cilk try` statements. Depending on what kind of exception the call to `C` throws, a different `catch` clause is executed. For instance, if `C` throws an `ArithmeticException`, the catchlet corresponding to the `catch` clause in lines 7–9 is executed. On the other hand, if `C` throws a `RuntimeException`, then the catchlet corresponding to the `catch` clause in lines 12–14 is executed.

In order to pick the appropriate catchlet to execute, three pieces of information are needed: the throw point (i.e., where the exception occurs) within the parent method, the type of the exception thrown, and the catch range of each catchlet. We already have the first two pieces of information. The throw point corresponds to the

```
 1  cilk void threeWay() throws IOException {
 2     spawn A(); // pc = 1
 3     cilk try {
 4        spawn B(); // pc = 2
 5        cilk try {
 6           spawn C(); // pc = 3, throws exception.
 7        } catch(ArithmeticException e) {
 8           cleanupC(); // pc = 4
 9        } finally {
10           D(); // pc = 4
11        }
12     } catch(RuntimeException e) {
13        cleanupB(); // pc = 5
14     } finally {
15        E(); // pc = 5
16     }
17     F();
18     sync; // pc = 6
19  }
```

Figure 4-2: A method containing nested `cilk try` blocks, each containing a `spawn` statement with `catch` and `finally` clauses. The comments show the `pc` value for each corresponding statement.

return entry of the throwing child. The type of exception is known because we have intercepted the exception. The compiler generates the `setCatchletReturn` method defined in `CilkFrame` to provide the catch ranges of the catchlets.

The `setCatchletReturn` is a ***lookup-table method*** which simulates a lookup table that dispatches to the appropriate catchlet by examining the value of the throw point and the type of the thrown exception. If the exception is not caught within the parent method, the `setCatchletReturn` rethrows the exception to the runtime system. The runtime system then handles the exception by repeating the process and propagating the exception upward in the call stack. Figure 4-3 illustrates the generated `setCatchletReturn` method for the `threeWay` method. The `setCatchletReturn` executes a method call to `cleanupB` if a `RuntimeException` is thrown when the `pc` value is 2, 3, or 4 (which corresponds to the statements in lines 4–11 in Figure 4-2.) Similarly, the `setCatchletReturn` executes a method call to `cleanupC` if an `ArithmeticException` is thrown when the `pc` value is 3 (which

```
 1  public void setCatchletReturn(int returnEntry, Throwable thrown) throws Throwable {
 2     switch(returnEntry) {
 3        case 2:
 4        case 4:
 5           if(thrown instanceof RuntimeException) {
 6              cleanupB();
 7           }
 8           return;
 9        case 3:
10           if(thrown instanceof ArithmeticException) {
11              cleanupC();
12           } else {
13              if(thrown instanceof RuntimeException) {
14                 cleanupB();
15              }
16           }
17           return;
18        default:
19           throw thrown;
20     }
21  }
```

Figure 4-3: The lookup-table method for `threeWay` in Figure 4-2 that dispatches to the appropriate catchlet.

corresponds to the statement in line 6.)

Executing finallets

The mechanism for executing finallets is similar to the ones for executing catchlets. A difference between executing finallets and executing catchlets is that, whether a `cilk try` block completes normally or abruptly (with exception), its finallet must be executed, whereas its catchlet executes only upon the occurrence of a caught exception.

When a `cilk try` block completes abruptly, the return entry of the thrown child is used to dispatch to the correct finallet (by invoking the `setFinalletReturn` method defined in `CilkFrame`, such as the one shown in Figure 4-4.) When a `cilk try` block completes normally, on the other hand, the `pc` value corresponding to the last statement of the `cilk try` block is used. This `pc` value is recorded by the primary

```
public void setFinalletReturn(int pc) throws Throwable {
   switch(pc) {
      case 2:
      case 4:
         E();
         return;
      case 3:
         D();
         return;
   }
}
```

Figure 4-4: The lookup-table method for `threeWay` in Figure 4-2 that dispatches to the appropriate finallet.

locus of control when it leaves the `cilk try` block and realizes that not all children dynamically enclosed by the `cilk try` have completed yet.

A complication arises if the primary locus of control is aborted and resumed outside the `cilk try` block. The primary locus may potentially jump across multiple levels of `cilk try` statements without recording the `pc` for executing the skipped finallets. For instance, when `C` in the `threeWay` method (line 6, Figure 4-2) throws a `RuntimeException`, and the primary locus of control still resides in the outer `cilk try` block (lines 3–12), the primary locus is aborted and resumed at the statement after the `cilk try` statement (line 16 where `pc = 5`). The primary locus might have skipped two `finally` clauses (in lines 9–11 and lines 14–16 respectively) without recording the corresponding `pc` values.

To solve this complication, the compiler provides a method, `getFinalletPickup`, which examines the current `pc` and returns the next `pc` representing the thread boundary immediately after the innermost `cilk try` statement enclosing the current `pc`. The runtime system queries the `getFinalletPickup` method iteratively to simulate the aborted primary locus jumping from the end of a `cilk try` statement to the end of the next `cilk try` statement. By doing so, the runtime system records all `pc` values passed along the way, until it reaches the point where the primary locus resumes. Figure 4-5 shows the `getFinalletPickup` method for the `threeWay` method.

The JCilk compiler generates these lookup-table methods (such as `setCatchletReturn`,

```
public int getFinalletPickup(int pc) {
   switch(pc) {
      case 2:
      case 4:
         return 5;
      case 3:
         return 4;
      default:
         return -1;
   }
}
```

Figure 4-5: The lookup-table method for `threeWay` in Figure 4-2 that returns the `pc` for the end of the next enclosing `cilk try` statement.

`setFinalletReturn`, and `getFinalletPickup`) by walking the abstract syntax tree [5, Sec. 2.5] and performing static analysis on the parsed JCilk code during the compilation stage in Polyglot.

4.2 Support for implicit abort

This section describes how the compiler supports the implicit abort mechanism when an exception occurs. When an exception occurs, JCilk must determine which side computations to abort. A data structure called a "try tree" and a lookup-table method called `getCatchletAltitude` are used to provide information about which spawned children to abort. When an abort takes place, the primary locus of control of the catching method must be aborted as well, if it still resides within the `cilk try` statement. The compiler also helps the runtime system to decide at which thread boundary the primary locus should resume in the catching method. Furthermore, if the programmer catches the `CilkAbort` exception for cleanup, the runtime system relies on the compiler to ensure that the `CilkAbort` exception is rethrown, so that the call stack can unwind properly back to the top-level method where the exception is caught.

The try tree

Due to the potential parallelism in a `cilk` block, the system must be able to keep track of possibly more than one concurrent exception occurring among parallel computations within the same `cilk` block. While selecting only one exception to be handled, the system also needs to know which side computations to signal to abort. Determining which side computations to signal to abort can be complex. Consider the method `threeWay` in Figure 4-2 again. Depending on what kind of exception the call to `C` (line 6) throws, different sets of spawned methods might receive the abort signal. For example, if `C` throws a `RuntimeException`, then `B` could be aborted (assuming it was still running), but `A` would continue normally.

In order to determine which spawned child methods should be aborted, the worker must keep track of where in the parent method they were originally spawned from. This information is maintained using the ***try tree*** data structure. In the same way that the ready deque mirrors the Java call stack, the try tree mirrors the dynamic hierarchy of nested `cilk try` statements. It is sufficient to maintain one try tree per worker, in its ***closure***, which is the top frame of the worker's ready deque. Because of how work stealing operates, the closure is the only frame within the deque that runs as a slow clone and might have children running on other workers. Applying the work-first principle, maintaining the try tree in the closure does not add significant overhead to the work term.

The try tree keeps track of the top-level method's children currently executing on other workers and from which `cilk` block they were spawned. The try tree contains three different kinds of nodes: internal nodes, leaves, and a cursor. Each internal node represents a `cilk` block, which can be either a `cilk` method or a `cilk try` block; a leaf usually represents a spawned child that is currently executing on a different worker. A node's parent in the try tree represents the `cilk` block most directly containing the node. That means each leaf's parent node corresponds to the innermost `cilk` block from which the child was spawned. The try tree contains only one ***cursor***, which tracks the `cilk` block containing the worker's current locus of control. The cursor can be either a leaf or an internal node. When the cursor is a leaf, it represents a spawned child method which is currently executing on the same worker. When the cursor is an internal node, it represents the `cilk` block currently executing on this worker, which contains child/children executing on different worker(s).

Maintaining the try tree is mainly done in the runtime system, except that the compiler generates calls before entering and after leaving a `cilk try` statement to inform the worker to update the try tree accordingly. How the try tree is maintained in the runtime system is beyond the scope of this thesis. Interested readers can refer to John Danaher's thesis [12] for more information.

Aborting with the try tree and the altitude lookup table

Having the try tree alone is not sufficient enough to tell which side computation to abort, because the try tree does not give information about which node in the tree catches the exception. This information is provided by the JCilk compiler, via a method called `getCatchletAltitude` defined in `CilkFrame`. The `getCatchletAltitude` method essentially contains a lookup table which examines the throw point and the thrown exception type and tells how many levels up the try tree the exception is caught.

The `getCatchletAltitude` method for the `threeWay` method is shown in Figure 4-7. For instance, if the spawned method `C`, whose return entry is 3, throws a `RuntimeException`, it is caught by the outer `cilk try` in line 3 in Figure 4-2. Thus, the `getCatchletAltitude` returns 2. In the dynamic try-tree representation (shown in Figure 4-6), two levels up from node **C**, representing the execution of `C`, is node v representing the outer `cilk try`. Given this information, the runtime system decides that everything below node v should be aborted (which includes method `B`, method `C`, and possibly the primary locus of the method `threeWay`.) If the thrown exception is not caught within this top-level method `threeWay`, `getCatchletAltitude` returns -1. In this case, the worker for the `threeWay` method needs to pass the exception up to its parent.

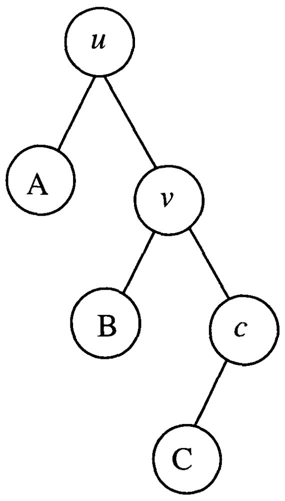

Figure 4-6: The try tree corresponding to an execution of `threeWay` in Figure 4-2. Node u represents the `cilk` method itself. Node v represents the outer `cilk try` block. Node c represents the inner `cilk try` block, which is also where the cursor is. Nodes `A`, `B`, and `C` represents the call to method `A`, `B`, and `C` respectively. Each method is executing on a different worker.

Resuming the primary locus of control after an abort

When an abort takes place, if the primary locus of control of the catching method still resides in the `cilk try` statement, the primary locus of control is a side computation that must be signaled to abort. This case is more complex, because the primary locus is residing in a `cilk try` block which is being aborted. That means, the primary locus must return to the runtime system as soon as possible, skip the rest of the JCilk threads within the aborted block, and resume at the point immediately after the catching `cilk try` statement.

The difficulty is how the worker figures out at which thread boundary the primary locus should resume. The method `getCatchletPickup` defined in `CilkFrame` returns the correct `pc` value with which to resume by examining the old `pc` of the primary locus and the type of the thrown exception.

Figure 4-8 shows the `getCatchletPickup` method for the `threeWay` method. Using the same example, if the spawned method `C` throws a `RuntimeException` while the primary locus hasn't left the catching `cilk try` (lines 3–12 in Figure 4-2, which contains thread boundaries with `pc = 3` or `pc = 4`), the primary locus should resume af-

```
public int getCatchletAltitude(int returnEntry, Throwable thrown) {
   switch(returnEntry) {
      case 2:
      case 4:
         if(thrown instanceof RuntimeException) {
            return 1;
         } else {
            return -1;
         }
      case 3:
         if(thrown instanceof ArithmeticException) {
            return 1;
         } else {
            if(thrown instanceof RuntimeException) {
               return 2;
            } else {
               return -1;
            }
         }
      default:
         return -1;
   }
}
```

Figure 4-7: A lookup table that tells how many levels up the try tree the exception `thrown` is caught.

ter the catching `cilk try` (line 16) with `pc = 5`. Similar to `getCatchletAltitude`, if the thrown exception is not caught within the top-level method, `getCatchletPickup` returns -1.

The JCilk compiler provides the lookup-table methods for the runtime system to figure out what to abort, where to resume, and which catchlet/finallets to execute. How the runtime system signals abort and ensures that the abort takes place only during thread boundaries are beyond the scope of this thesis. Again, interested readers can refer to John Danaher's thesis [12] for more information.

Ensuring the rethrow of `CilkAbort` exception

When an exception occurs, JCilk delivers a `CilkAbort` exception semisynchronously to the side computations of the exception, which is accomplished by delivering abort

```
public int getCatchletPickup(int oldPC, Throwable thrown) {
   switch(oldPC) {
      case 2:
      case 4:
         if(thrown instanceof RuntimeException) {
            return 5;
         } else {
            return -1;
         }
      case 3:
         if(thrown instanceof ArithmeticException) {
            return 4;
         } else {
            if(thrown instanceof RuntimeException) {
               return 5;
            } else {
               return -1;
            }
         }
      default:
         return -1;
   }
}
```

Figure 4-8: A lookup table that tells where the primary locus of control should resume if it is aborted due to a thrown exception.

signals to the workers working on those side computations. When a worker sees the abort signal, it might be working on a frame buried deeply within the ready deque. A `CilkAbort` exception is thrown by the worker to unwind the call stack (both the shadow stack and the Java call stack) back to the point where the exception is caught. The programmer can catch the `CilkAbort` exception within a method along the aborting path and perform cleanup code. The caught `CilkAbort` must be rethrown properly, however, to ensure the unwinding of the call stack. The compiler generates code in the GoJava output to perform the rethrow.

There are two cases in which the compiler inserts code to ensure the rethrow of `CilkAbort` exception. The first case is when the programmer specifies a `catch` clause catching the `CilkAbort` exception. The second case is when the programmer specifies the `cilk try` statement to contain a `finally` clause. The `finally` clause may potentially throw another exception or return a value that overwrites the `CilkAbort` exception thrown from the `cilk try` block. In either case, a `try` statement with `finally` is generated to wrap around the `cilk try` in the GoJava output. Within the generated `finally` block, a call to the worker is made to see if this method is being aborted. If so, a new `CilkAbort` exception is thrown. In the case where the programmer specifies `cilk try` to contain only the `catch` clause catching the `CilkAbort` exception but no `finally` clause, the generated `finally` is attached to the compiled `cilk try` directly. Doing so ensures the rethrow of `CilkAbort`, no matter what happens in the programmer-specified `catch` clause or `finally` clause.

4.3 Support for the lexical-scope rule

This section describes what the compiler needs to implement in order to support the lexical-scope rule when a loop containing a `cilk try` block with a `catch` clause or a `finally` clause refers to a locally declared loop variable.

Enforcing the lexical-scope rule

When the programmer writes code including a `cilk try` within a loop, and its `catch` or `finally` clause refers to locally declared loop variables (such as the one shown in Figure 2-6), the lexical-scope rule applies. Remember that JCilk allows the execution of the `catch` or `finally` clause from each iteration to be delayed and executed as a catchlet or finallet some iterations later. As each secondary locus of control executes its catchlet or finallet, it refers to the version of the loop local variables associated with the iteration to which the clause belongs lexically. We refer to the version of the loop local variables from an iteration as the ***lexical environment*** for that iteration.

In a concurrent context, it is indefinite how many iterations later the secondary loci of control in the loop will be executed, and the lexical environment for each delayed secondary locus of control must be remembered. That is, the compiler must generate code to record these lexical environments explicitly. The question is, how many lexical environments would that be? One naive answer would be, as many loop iterations as the execution has gone through, which could represent a big overhead.

Actually, JCilk provides an efficient implementation of the lexical-scope rule that avoids creating many extraneous versions of lexical environments. In practice, the number of lexical environments remembered is only as many as the number of steals during the duration of the loop execution, no matter how distant the execution of a secondary locus is. Conforming to the work-first principle, the overhead incurred by enforcing the lexical-scope rule contributes only to the critical-path term.

Whenever a situation arises where the lexical-scope rule applies, the compiler generates code to implement the lexical environment with a C-struct-like private inner class. The class contains only member fields corresponding to the local variables declared within the loop together with one additional member field `stolen`, which is set to `true` when this particular lexical environment is stolen.

In the fast clone, before entering the loop, a lexical environment is created and stored within the method frame. By default, the `stolen` bit in the newly created lexical environment is initialized to `false`. This lexical environment is reused repeatedly as the primary locus of control proceeds to the next iteration of the loop, until a steal occurs. When a steal occurs, the method is resumed as a slow clone. At the beginning of the resumption of the slow clone, the `stolen` bit in the stolen lexical environment is set to `true`. The execution of the method continues where it was left off, which might be in the middle of the loop. At the beginning of the next loop iteration, if the `stolen` bit is set to `true`, a new lexical environment is created for the new iteration. Since the old environment is still alive and stored somewhere within the runtime, it can be read when the secondary loci of control for the older iterations are executed. The newly created environment is again reused over and over until the next steal happens.

Within the runtime, the lexical environments for the older iterations are actually stored within the try tree until the secondary loci execute their corresponding catchlets and finallets. Within the loop, whenever there is a **spawn** statement, the lexical environment for that loop iteration is passed to the spawned child, in the same way we pass the return entry value. When the child returns, if the parent is not stolen, the same lexical environment is reused. If the parent has been stolen, however, the lexical environment is then passed to the runtime and stored in the try tree, in the same node created for the catchlet or finallet. When `setCatchletReturn` or `setFinalletReturn` is invoked, the lexical environment is passed to the method as an additional argument.

4.4 Support for evaluation order

When the assignment operators are used with `spawn` in JCilk, the operands must be handled specially in order to conform to Java's "left-to-right" evaluation order [20, Sec. 15.7]. This section explains what the compiler needs to implement in order to handle the assignment statement correctly. In particular, some complications are involved when the keyword `spawn` is used with the compound assignment operators such as "`+=`."

Spawned child returning to a reference type

The Java Language Specification [20] guarantees that the operands of operators always appear to be evaluated in the ***left-to-right*** evaluation order. That is, the left-hand operand appears to be evaluated before the right-hand operand. For instance, Figure 4-9 demonstrates a Java program whose output depends on the evaluation order. In Java, the only possible output for this program is 12, because the value of the left-hand operand is fetched and remembered before the right-hand operand of the addition is evaluated in line 3. In C, a program with the same code has unspecified behavior, because no evaluation order is strictly enforced.

The enforcement of the evaluation order is especially important when the left-hand operand has a reference type, and the right-hand operand involves a `spawn`, as illustrated in Figure 4-10. In line 3, the primary locus of control spawns off method `A`. The value returned by `A` eventually stores into `x.y[i]` (i.e., the element indexed by `i` in the array field `y` of `x`.) Suppose that the primary locus proceeds to the next statement after the `spawn` and finishes executing the call to method `B` in line 5. When

```
public static void f4-9(String[] args) {
   int a = 7;
   a = a + (a = 5);
   System.out.println(a);
}
```

Figure 4-9: A Java program whose output depends on the evaluation order. The left-hand operand of the addition, a is evaluated before the right-hand operand, (a = 5).

```
1   public static void f4-10(String[] args) {
2      ⋮
3      x.y[i] = spawn A();
4      oldX = x;
5      x = B(); // returns a new myObj
6      ⋮
7   }
```

Figure 4-10: A JCilk program whose result depends on the evaluation order of the assignment operator. The method B returns a brand-new object, which is assigned to x in line 5. In the serial elision, the result of the method does not depend on the evaluation order. In JCilk, however, the result depends on the evaluation order due to the semantics of spawn.

the spawned child finishes executing A and returns, x no longer refers to the same object instance. According to Java's evaluation order, however, the value returned by A should be stored in the original object, as evaluated in line 3. Hence, the compiler must take special care in order to handle this case correctly.

When the left-hand operand of an assignment operator is an array access expression, Java's evaluation order is specified as follows [20, Sec. 15.26.1].

1. Evaluate the array reference subexpression in the left-hand operand.
2. Evaluate the index subexpression in the left-hand operand.
3. Evaluate the right-hand operand.
4. Check if the array reference subexpression indeed refers an array. No assignment occurs and a `NullPointerException` is thrown if the subexpression evaluates to `null`.
5. Check the bound of the index subexpression. No assignment occurs and an `ArrayIndexOutOfBoundsException` is thrown if the subexpression evaluates to

be out of range.

6. Use the index subexpression to select an element in the array and make the assignment. If the types of the left-hand operand and the right-hand operand are not compatible, `ArrayStoreException` is thrown.

The JCilk compiler implements the following in order to follow the exact evaluation order.

1. Store the array reference subexpression

2. Store the index subexpression

3. Evaluate the `spawn`

4. As the spawned child returns, use the array reference subexpression and the index subexpression stored before the `spawn`.

Actions 1–3 in JCilk's evaluation order correspond to actions 1–3 in Java's evaluation order respectively. When action 4 in JCilk's evaluation order is executed, the order of checks are performed naturally, following the order of actions 4–6 in Java's evaluation order.

The array reference subexpression and the index subexpression cannot be stored in the parent's activation frame. Otherwise, we may lose the old values before the `spawn`, because the values in the parent's frame may get updated before the spawned child returns. Instead, their values are stored in a lexical environment, which is passed to the child as an argument when the child is spawned. When the child returns, the lexical environment is then handed to the runtime and is used to invoke `setInletReturn` for returning to the stolen parent. This mechanism is similar to how the JCilk compiler handles the lexical environments to enforce the lexical-scope rule, as described in Section 4.3.

The same mechanism is used in the case where the left-hand operand of the assignment operator is a field reference type, such as `x.y.z`. The prefix before the last field access (i.e., `x.y`) is stored in a lexical environment before the `spawn`.

```
 1   cilk int f4-11() {
 2      ⋮
 3      for(int i=0; i<10; i++) {
 4         array[i] = spawn A(i);
 5      }
 6      oldArray = array;
 7      array = makeNewArray();
 8      ⋮
 9      sync;
10      for(int i=0; i<10; i++) {
11         System.out.println(oldArray[i]);
12      }
13   }
```

Figure 4-11: A program whose output depends on both the lexical-scope rule and the left-to-right evaluation order.

Spawning in a loop

Things can get somewhat complicated when the result of a program depends on both the lexical-scope rule and the evaluation order. Figure 4-11 demonstrates such an example.

Different from Figure 2-5 as presented in Chapter 2, method `f4-11` does not contain a race condition. Assuming the left-to-right evaluation order is applied properly, the read access to `i` when evaluating `array[i]` (line 4) for each iteration must be performed before the write access to `i` when executing `i++` in line 1 for the next iteration. That means, accesses to `i` are properly ordered with respect to one another. With the thread atomicity guaranteed by JCilk, no data races associated with `i` can occur. Similarly, since the read access to `array` in each loop iteration (line 4) must be performed before the write access to it after the loop (line 7), no data races associated with `array` can occur.

To compile method `f4-11` correctly, JCilk must evaluate both `array` and `i` before the `spawn`, store their values in a lexical environment, and renew the lexical environment at the beginning of the loop iteration when steal happens, as described in Section 4.3.

```
 1   cilk private int fib(int n) {
 2      int res;
 3      if( n < 2 ) {
 4         return n;
 5      }
 6      res += spawn fib(n-1);
 7      res += spawn fib(n-2);
 8      sync;

 9      return res;
10   }
```

Figure 4-12: A simple JCilk program that computes the Fibonacci number by summing the returned values from spawning `fib(n-1)` and `fib(n-2)` into one variable, `res` with operator "+=."

Spawning with a compound assignment operator

JCilk obeys Java's evaluation order in all cases, except when a compound assignment operator such as "+=" is used with `spawn`. The Java Language Specification specifies that, a compound assignment expression of the form *E1 op= E2* is evaluated as *E1 = E1 op E2* [20, Sec. 15.26.2]. In JCilk, when a compound assignment operator is used with the `spawn` keyword, however, the serial elision of *E1 op= spawn E2;* should be considered as *E1 = (spawn E2) op E1;* (where *E2* is a `cilk` method call).

When a compound assignment operator is used in JCilk, it is often used in a fashion illustrated in Figure 4-12. Because of JCilk's guarantee for thread atomicity, the `fib` method executes as what the programmer intends to be: the final result is being accumulated with the variable `res`. The order in which the spawned methods return does not affect the correctness of the `fib` program, as long as `res` is being updated atomically.

If we took the alternative and evaluated *E1 += spawn E2* as *E1 = E1 + (spawn E2)*, we might lose one update to *E1*. In the case of Figure 4-12, suppose that the primary locus of control continues to execute `fib(n-2)` in line 7 after spawning off `fib(n-1)` in line 6, resulting `fib(n-1)` and `fib(n-2)` being computed in parallel. When the spawned child that finishes second returns, it would not use the `res` value updated by the child that returned first, but instead, it would use the `res` evaluated before the `spawn`. Consequently, we would lose the update made by the first child. This behavior is not what we desire. Therefore, we decided to go with the evaluation order *`E1 = (spawn E2) + E1`*.

Chapter 5

Related work

This chapter discusses related work and attempts to place JCilk and its exception-handling semantics into the context of parallel programming languages. A key difference between JCilk and other work on concurrent exception handling is that JCilk provides a faithful extension of the semantics of a serial exception mechanism, that is, the serial elision of the JCilk program is a Java program that implements the JCilk program's semantics.

Most parallel languages do not provide an exception-handling mechanism. For example, none of the parallel functional languages VAL [2], SISAL [17], Id [38], parallel Haskell [4, 39], MultiLisp [22], and NESL [6] and none of the parallel imperative languages Fortran 90 [3], High Performance Fortran [43] [35], Declarative Ada [48,49], C* [23], Dataparallel C [24], Split-C [10], and Cilk [46] contain exception-handling mechanisms. The reason for this omission is simple: these languages were derived from serial languages that lacked such linguistics.[1]

Some parallel languages do provide exception support, because they are built upon languages that support exception handling under serial semantics. These languages include Mentat [21], which is based on C++; OpenMP [41], which provides a set of compiler directives and library functions compatible with C++; and Java Fork/Join Framework [30], which supports divide-and-conquer programming in Java. Although

[1]In the case of Declarative Ada, the researchers extended a subset of Ada that does not include Ada's exception package.

these languages inherit an exception-handling mechanism, their designs do not address exception-handling in a concurrent context.

Tazuneki and Yoshida [47] and Issarny [27] have investigated the semantics of concurrent exception-handling, taking different approaches from our work. In particular, these researchers pursue new linguistic mechanisms for concurrent exceptions, rather than extending them faithfully from a serial base language as does JCilk. The treatment of multiple exceptions thrown simultaneously is another point of divergence.

Tazuneki and Yoshida's exception-handling framework is introduced in the context of DOOCE, a distributed object-oriented computing environment. They focus on handling multiple exceptions which are propagated from concurrently active objects. DOOCE adapts Java's syntax for exception handling, extending it syntactically and semantically to handle multiple exceptions. Unlike JCilk, however, DOOCE allows a program to handle multiple exceptions by listing several exception classes as parameters to a single `catch` clause with the semantics that the `catch` clause executes only when all those exceptions are thrown. DOOCE's semantics include a new resumption model as an alternative to the termination model of Java: when exceptions occur and are handled by a `catch` clause, the `catch` clause can indicate that the program should resume execution at the beginning of the `try` statement instead of after the `catch` block.

The cooperation model proposed by Issarny provides a way to handle exceptions in a language that supports communication between threads. If a thread terminates due to an exception, all later threads synchronously throw the same exception when they later attempt to communicate with the terminated thread. Unlike JCilk's model, the cooperation model accepts all of the simultaneous exceptions that occur when multiple threads involved in communication have terminated. Those exceptions are passed to a handler which resolves them into a single concerted exception representing all of the failures.

The recent version of the Java Language, known as Tiger or Java 1.5 during development and now called Java 5.0 [34], provides call-return semantics for threads similar on the surface to JCilk. In particular, Java 1.5 provides a protocol that is similar to that of JCilk. Although Java 5.0 (like everything else in Java) uses an object-based semantics for multithreading, rather than JCilk's choice of a linguistic semantics, it does move in the direction of providing more linguistic support for multithreading. In particular, Java 5.0 introduces the `Executor` interface, which provides a mechanism to decouple scheduling from execution. It also introduces the `Callable` interface, which, like the earlier `Runnable` interface, encapsulates a method which can be run at a later time (and potentially on a different thread). Unlike `Runnable`, `Callable` allows its encapsulated method to return a value or throw an exception. When a `Callable` is submitted to an `Executor`, it returns a `Future` object. The `get` method of that object waits for the `Callable` to complete, and then it returns the value that the `Callable`'s method returned. If that method throws an exception, then `Future.get` throws an `ExecutionException` containing the original exception as its cause. (The `Future` object also provides a nonblocking `isDone` method to see if the `Callable` is already done.)

One notable difference between JCilk and Java 1.5 is that JCilk's parallel semantics for exceptions faithfully extend Java's serial semantics. Although Java 1.5's exception mechanism is not a seamless and faithful extension of its serial semantics, as a practical matter, I believe it represents a positive step in the direction of making parallel computations linguistically callable.

Chapter 6

Conclusion

This chapter discusses possible improvements and future directions for the JCilk project. The JCilk project is our first attempt to solve the difficulties of developing parallel software. Although we have taken the first step, we are still quite a distance away from our eventual goal. There is still much more work to do. To be specific, I propose four directions for future research:

1. improve system performance;
2. connect Java and JCilk;
3. provide both static and dynamic threading models in JCilk and allow them to interact;
4. integrate JCilk with other on-going projects in the group.

To conclude this thesis, this chapter offers brief discussion on the possibilities of each proposal, and draws concluding remarks.

Improving system performance

To date, the performance of the JCilk system is far from ideal. Despite the simple programming model and the concrete exception-handling semantics that JCilk offers, until the efficiency of the system can be improved, the language itself is not useful. The overhead in the JCilk system comes from two main sources: object allocation and synchronization.

Object allocation is the first main source of overhead. A JCilk program potentially allocates many more objects than its serial elision. For every call to a `cilk` method, an additional object is created to represent the activation frame in the shadow stack for the `cilk` method. Even though only one extra object is created for each `spawn`, the impairment caused by this overhead is especially apparent when the spawned method does not perform much work. For instance, the only work performed by the `fib` method shown in Figure 3-3 is one addition, which is not enough to compensate for the `spawn` overhead. When executing `fib` with `n = 40`, the method is called 331,160,281 times recursively (meaning, 331,160,281 extra object creations), and the overhead for executing on 1 processor versus executing the serial elision (T_1 / T_s) is roughly 25 times. This object creation is necessary, however, because it enables the thief to steal the method and allows the continuation to take place.

Synchronization is the second main source of overhead. Due to how the Java memory model works, the efficiency of the THE protocol implemented in Cilk-5 [16] is diminished in the JCilk-1 runtime system. The THE protocol takes advantage of a cheap memory barrier to avoid acquiring locks (which cost more overhead) in the common case. In Java, however, in order for the frame update done by one worker to be seen by another worker, both workers must synchronize. This synchronization must be performed even when a worker pushes a frame into its own ready deque. Otherwise, the update in the frame performed by the victim before the push might not be seen by the thief when it steals the frame. This synchronization causes significant overhead, which is borne by the work term.

To save the system from drowning in these overheads, I propose two alternative implementations. The first alternative is to use a lazy frame-creation technique, called the "indolent closure creation" [45] to avoid extraneous frames from being created if no steal happens. The second alternative is to modify the IBM JRE (Java Runtime Environment — IBM version of Java virtual machine) to support continuation as a primitive to eliminate the need for creating frames.

The main idea of the indolent closure creation is the following: instead of eagerly creating and pushing a frame into the ready deque whenever a `spawn` is encountered, only create a frame when a steal attempt happens. That is, no frames are created and pushed into the ready deque if there is no steal. When the first steal happens, the steal attempt fails, because there is nothing in the ready deque to steal. The victim, after detecting the failed steal attempt, unrolls the call stack, creates and pushes the corresponding frames into its ready deque, and resumes execution of where it left off when the steal attempt was detected. Hence, after the first failed steal attempt, the subsequent steals are likely to succeed. Indolent closure creation reduces the overhead of frame creation from the number of spawn calls to the number of unsuccessful steals. In addition, all the synchronization needed when pushing the frame can be eliminated. Since the victim creates and pushes frames into its deque upon the request of a steal, a frame is modified only by the thief after it is stolen. The fields in the frame are not updated between the point of creation and the point of steal. Hence, no synchronization is needed when pushing the frames, saving the overhead in the work term. The indolent closure creation takes the work-first principle to the extreme. There is one downside associated with this technique, however. With indolent closure creation, it is indefinite how long a thief must wait before its steal attempt succeeds. Even though we are minimizing the overhead in the work term, we are also blowing up the overhead in the critical-path term indefinitely. Recall that, the work-first principle replies on the assumption of sufficient parallel slackness, meaning $\overline{P}/P \gg c_\infty$. If we blow up c_∞ indefinitely, the work-first principle no longer stands. Hence, we must be cautious about the waiting time of a steal when applying this technique. Nonetheless, this proposal seems worthy of investigation.

The other proposal is to modify the existing IBM JRE to support continuation as a primitive and thereby eliminate the need to create frames altogether. The reason why we are creating a frame for every spawn call is to allow code migration from one worker to another. The frame is necessary to implement continuation so that a worker thread can access the state of local variables stored in another worker thread's call stack. If we can modify the IBM JRE to support continuation as a primitive, which allows one worker to read and copy another worker's call stack cheaply, we no longer need to create any frames. Of course, this is easier said than done. As discussed in

Section 3.2, there are several disadvantages to modifying a Java virtual machine to support JCilk. This approach makes the JCilk compiler highly dependent on the IBM JRE, less extensible, and harder to keep up-to-date. Nevertheless, modifying the IBM JRE is more feasible than modifying other open-source Java virtual machines, because the IBM JRE is portable (implemented in Java) and fairly up-to-date (supporting Java 1.4). Therefore, this alternative is feasible, although the amount of engineering effort involved might be daunting.

Connecting Java and JCilk

In the current implementation, a Java method cannot call a `cilk` method. Symmetrically, a `cilk` method must be spawned and cannot be called. Even though this constraint is required by the code transformation that JCilk performs at compile time, it leads to a sharp delineation between `cilk` methods and ordinary Java methods. We would like to break this barrier and allow the two languages to be truly connected. This proposal poses some difficulties, however.

From an implementation perspective, this proposal is problematic, because an ordinary Java method cannot be stolen due to its lack of support for continuations. Without support for continuations, the state of local variables of a method exists only in the ordinary Java call stack. If we had allowed a Java method to call a `cilk` method, and a steal happens, the `cilk` method (the child) would have no means to return back to the Java method (the parent). Since a Java `Thread` cannot access the call stack executing on another Java `Thread`, the worker executing the `cilk` method cannot return a value back to another worker executing the Java method. The state of local variables of the Java method are simply not accessible.

From the linguistics perspective, this proposal poses some issues as well. Suppose that we have no implementation difficulties to allow an ordinary Java method to call a `cilk` method. Some linguistic problems arise when considering Java's semantics for inheritance and method overriding. For example, a programmer may want to extend the `java.util.Vector` class in Java API and write a parallel version of the class, named `ParallelVector`, overriding the `indexOf(Object elem)` method (which searches for the first occurrence of `elem`). When executing an ordinary Java method which takes an instance of `Vector` as its argument, an instance of `ParallelVector` might be passed instead. When the `indexOf` method is invoked, the following possibilities (among other ones) may result: the serial version `Vector.indexOf` is invoked; the parallel version `ParallelVector.indexOf` is invoked but executed serially; the parallel version `ParallelVector.indexOf` is invoked and executed in parallel.

Due to Java's portability, it is possible to execute two Java programs together which are compiled separately. Thus, if we allow a Java method to call a `cilk` method, and a `cilk` method to override a Java method, it then becomes possible for a Java method to invoke a `cilk` method and introduce parallelism into the application unexpectedly. This proposal induces a new set of complex linguistic problems to be solved, which can be interesting to consider in the future.

Providing both static and dynamic threading models

For now, JCilk offers only the dynamic threading model but not the static threading model. The static threading model and the dynamic threading model are each suited for different types of applications. (Their differences are summarized in Figure 1-1 in Chapter 1.) We would eventually like to provide both models in JCilk, with semantics that allow them to interact in a sensible manner.

To integrate these two models, one must consider their different hierarchical structures and communication patterns. Each thread in the static threading model is an independent entity. The dynamic model, on the other hand, groups threads with parent-child relations. Threads in the static threading model communicate via shared data structures, whereas threads in the dynamic threading model communicate with call-return semantics. At this point, we are not clear how threads from the two models can communicate with each other, and what form the communication would take.

It will take some thorough consideration to come up with a sophisticated model that encompasses both threading models and incorporates them harmoniously. Nevertheless, we are hopeful that investigating this topic will eventually bear fruit.

Integrating with other projects

There are some other on-going projects in my research group (the Supercomputing Technologies Group in MIT CSAIL) presenting opportunities for integration with JCilk. These projects include adaptive thread scheduling and Transactional Java (XJava). These projects, like JCilk, investigate technologies to support high-performance computing.

Adaptive thread scheduling addresses the problem of scheduling multiple multithreaded jobs on a multiprocessor system. Most systems use static allocation, where each job running on the system is allocated to a fixed number of processors throughout its lifetime. The static allocation policy inevitably leads to unnecessary slow down or resource waste at one point or another during the execution when the parallelism of the jobs changes. This project focuses on designing algorithms that enable each job to estimate its parallelism efficiently and request processors accordingly.

Even though JCilk allows the programmer to determine the number of workers at runtime and distribute work near-optimally across these workers, the program operates with a fixed number of workers throughout its execution. This limitation is acceptable when only one JCilk program is running on the multiprocessor system. The JCilk program can simply be alloted all the processors. When there are multiple independent JCilk programs running simultaneously, however, it is better to adaptively reallocate processors. Integrating the dynamic processor allocator into the JCilk scheduler should provide a good platform for testing different algorithms.

Another project currently being developed in the group is Transactional Java (XJava), a transactional language that extends Java. The project focuses on enforcing atomicity between concurrent threads using transactional memory [25], replacing traditional locking protocols. The basic idea of the transactional memory is that concurrent transactions execute optimistically, assuming they can be executed in parallel without conflict. When a conflict occurs, one of the two conflicting transactions is chosen to roll back and automatically retry. If no conflict occurs, the transaction succeeds and commits its state-altering actions (such as writing to memory) permanently.

Some actions are irrevocable, however, such as printing to the screen.

The exception-based abort mechanism in JCilk provides one possible solution to handle such irrevocable actions. One can imagine that, when a transaction fails, a `TransactionAbort` exception can be thrown to the transactional block to indicate the failure. The exception can be caught, giving the programmer a chance to cleanup and minimize the damage done by the irrevocable actions before the transaction is retried again.

Concluding remarks

The JCilk language is designed to facilitate the development of parallel software. JCilk provides a dynamic multithreading model and a provably good scheduler that handles the task scheduling and communication between threads. By incorporating simple parallel constructs into a widely used language, such as Java, JCilk aims to minimize the learning curve associated with adapting to a new language. At the same time, JCilk strives to retain the rich language features provided by Java. More importantly, JCilk's semantics for exception handling intends to encourage the development of robust and fault-tolerant software.

Even though JCilk-1 is only our initial step, and there is still much more to do, we are nonetheless advancing towards our eventual goal. We are hopeful that, in the foreseeable future, developing parallel software can be as straightforward as developing serial software.

Bibliography

[1] Kaffe.org. `http://www.kaffe.org/`, Jan. 2005.

[2] W. Ackerman and J. B. Dennis. VAL – a value oriented algorithmic language. Technical Report TR-218, Massachusetts Institute of Technology Laboratory for Computer Science, 1979.

[3] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook.* McGraw-Hill, 1992.

[4] S. Aditya, Arvind, J.-W. Maessen, L. Augustsson, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In P. Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, pages 35–49, June 1995.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, Apr. 1993.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[9] P. A. Buhr and W. Y. R. Mok. Advanced exception handling mechanisms. *Software Engineering*, 26(9):820–836, 2000.

[10] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

[11] D. Dailey and C. E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.

[12] J. S. Danaher. The JCilk-1 runtime system. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, June 2005.

[13] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Exception handling in JCilk. Unpublished manuscript, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2005.

[14] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, Sept. 1965.

[15] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. *Advances in Computer Chess 7*, pages 203–219, 1993.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[17] J.-L. Gaudiot, T. DeBoni, J. Feo, W. BHm, W. Najjar, and P. Miller. The Sisal model of functional programming and its implementation. In *PAS '97: Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis*, page 112. IEEE Computer Society, 1997.

[18] The GNU compiler for the Java programming language. `http://gcc.gnu.org/java/`, Oct. 2004.

[19] A. Gontmakher and A. Schuster. Java consistency: nonoperational characterizations for Java memory behavior. *ACM Trans. Comput. Syst.*, 18(4):333–386, 2000.

[20] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Massachusetts, 2000.

[21] A. S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *Computer*, 26(5):39–51, 1993.

[22] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, Oct. 1985.

[23] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson. A production-quality C* compiler for hypercube multicomputers. In *PPOPP '91: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82. ACM Press, 1991.

[24] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, R. J. Anderson, and R. R. Jones. Data-parallel C: A SIMD programming language for multicomputers. In *Distributed Memory Computing Conference, 1991. Proceedings., The Sixth*, pages 91–98. IEEE Computer Society, April 28-May 1 1991.

[25] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Conference on Computer Architecture. (Also published as ACM SIGARCH Computer Architecture News, Volume 21, Issue 2, May 1993.)*, pages 289–300, San Diego, California, 1993.

[26] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Std 1003.1, 1996 Edition.

[27] V. Issarny. An exception handling model for parallel programming and its verification. In *SIGSOFT '91: Proceedings of the Conference on Software for Critical Systems*, pages 92–100, New York, NY, USA, 1991. ACM Press.

[28] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Inc., 1988.

[29] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, Winter 1975.

[30] D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43. ACM Press, 2000.

[31] D. Lea and D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley, Boston, Massachusetts, 1999.

[32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition.* Addison-Wesley, Boston, Massachusetts, 2000.

[33] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6):546–558, Nov. 1979.

[34] B. McLaughlin and D. Flanagan. *Java 1.5 Tiger: A Developer's Notebook.* O'Reilly Media, Inc, 2004.

[35] J. Merlin and B. Chapman. High Performance Fortran, 1997.

[36] S.-M. Moon, B.-S. Yang, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, and S. Kim. LaTTe: An open-source Java virtual machine and just-in-time compiler. `http://latte.snu.ac.kr`, Jan. 2005.

[37] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[38] R. Nikhil. ID language reference manual. Computation Structure Group Memo 284-2, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, July 1991.

[39] R. S. Nikhil, Arvind, J. E. Hicks, S. Aditya, L. Augustsson, J.-W. Maessen, and Y. Zhou. pH language reference manual, version 1.0. Technical Report CSG-Memo-369, Computation Structures Group, Massachusetts Institute of Technology Laboratory for Computer Science, 1995.

[40] N. Nystrom, M. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, Apr. 2003.

[41] OpenMP C and C++ application program interface. `http://www.openmp.org/drupal/mp-documents/cspec20.pdf`, 2002.

[42] W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

[43] H. Richardson. High Performance Fortran: history, overview and current developments, 1996.

[44] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education Inc., Upper Saddle River, New Jersey, 2003.

[45] V. Strumpen. Indolent closure creation. Technical Report MIT-LCS-TM-580, Massachusetts Institute of Technology Laboratory for Computer Science, June 1998.

[46] Supercomputing Technologies Group,, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139. *Cilk 5.3.2 Reference Manual*, Nov. 2001.

[47] S. Tazuneki and T. Yoshida. Concurrent exception handling in a distributed object-oriented computing environment. In *ICPADS '00: Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, page 75, Washington, DC, USA, 2000. IEEE Computer Society.

[48] J. Thornley. *The Programming Language Declarative Ada Reference Manual.* Computer Science Department, California Institute of Technology, Apr. 1993.

[49] J. Thornley. Declarative Ada: parallel dataflow programming in a familiar context. In *CSC'95: Proceedings of the 1995 ACM 23rd Annual Conference on Computer Science*, pages 73–80. ACM Press, 1995.

[50] P. H. Winston. *Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, third edition, 1992.