The JCilk-1 Runtime System

by

John Danaher

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by . . . .
Charles E. Leiserson
Professor
Thesis Supervisor

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

The JCilk-1 Runtime System

by

John Danaher

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

JCilk extends the Java language to provide call-return semantics for multithreading, much as Cilk does for C. Java's built-in thread model does not support the passing of exceptions or return values from one thread back to the "parent" thread that created it. JCilk imports Cilk's fork-join primitives spawn and sync into Java to provide procedure-call semantics for concurrent subcomputations. It also introduces exceptions into that fork-join structure, leading to some 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. When executing in parallel, an exception thrown by a JCilk computation signals its "side computations" to abort, which yields a clean semantics in which only a single exception from the enclosing try block is handled. Because JCilk uses Java's normal exception mechanism to propagate an abort throughout the side computations, the programmer can handle clean-up by simply catching a thrown CilkAbort exception. JCilk supports these features by introducing the concept of a "catchlet" as a mechanism for handling exceptions in a concurrent context.

In my work, I have implented a runtime system for JCilk which uses a tree structure to track the dynamic state of nested try blocks. Using this tree, the runtime system is able to signal aborts to the proper side computations and determine when the catch block is able to run. The result is an efficient implementation of the JCilk specification.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

In no particular order:

Thanks to everyone in the Supercomputing Technologies group, and particularly to Jim Sukha and Bradley Kuszmaul, for many helpful discussions, ideas, and distractions throughout the past year.

Thanks to C. Scott Ananian for his many helpful discussions, especially at the beginning of the project. His knowledge of the subtleties of Java helped us get off the ground.

Thanks to the Singapore-MIT Alliance program for giving me a chance to get away from MIT for a couple weeks, not to mention the opportunity to present my work to a wider audience than the group. Thanks especially to Wong Weng Fai of National University of Singapore for his suggestions.

Thanks to all the Supertechies of the olden days who produced the original Cilk and who shaped it into the simple, elegant language it is today. JCilk wouldn't exist without their work.

Thanks to my friends for their advice and support, and for putting up with me while I finished writing.

Thanks to Angelina Lee for catching the many gaps in my reasoning over the past year and for implementing the compiler so that this thing would actually go.

Thanks most of all to my advisor, Charles Leiseron. Without his initial support, JCilk wouldn't even have gotten started, and without his ideas and constant support, it couldn't have gotten to where it is today.

This research was supported in part by NSF Grants ACI-0324974 and CNS-0305606 and in part by the Singapore-MIT Alliance.

Contents

1 Introduction 13

1.1 Java 14

1.2 Cilk 16

1.3 JCilk 20

2 JCilk Semantics 25

2.1 Basic JCilk Semantics 26

2.2 Exceptions in Java 28

2.3 Exceptions in JCilk 31

2.4 Advanced JCilk Exceptions 34

2.5 The Queens problem 38

3 The Work-Stealing Scheduler 41

3.1 The Compiler 41

3.2 Workers 46

3.3 Frames 48

3.4 Closures 50

4 Exception Implementation 55

4.1 Writing to the Try Tree 55

4.2 Reading from try tree 62

4.3 Being aborted 66

5 Performance **71**

5.1 Measurements 71

5.2 Sources of Overhead 72

5.3 Difficulties with Java 75

5.4 Alternative Implementations 77

6 Related Work **81**

7 Future Directions **85**

7.1 Connecting Java and JCilk 85

7.2 Future Integration 89

8 Conclusion **91**

List of Figures

1-1 The ancestors of JCilk. 14

1-2 A Cilk Computation DAG. 17

1-3 A recursive Cilk procedure to find Fibonacci numbers 18

1-4 A parallel search tree. 19

2-1 Finding Fibonacci numbers in JCilk. 26

2-2 More complex spawns in JCilk. 27

2-3 Two simple Java methods 29

2-4 The simplest way to catch an exception in JCilk. 31

2-5 A simple JCilk program using exceptions. 31

2-6 Catching a `CilkAbort`. 33

2-7 Handling exceptions with `cilk try` when aborting is unnecessary. . . 35

2-8 Handling exceptions with `cilk try` when aborting might be necessary. 36

2-9 The Queens problem coded in JCilk. 39

3-1 A JCilk method to compute the nth Fibonacci number. 42

3-2 A simplified version of the “fast clone” of the compiled Fibonacci method. 43

3-3 A simplified version of the “slow clone” of the compiled Fibonacci method. 45

3-4 An example of work-stealing. 47

3-5 The Frame method to execute inlets. 51

4-1 A method containing nested `cilk` blocks, each containing a `spawn` statement. 56

4-2 A try tree 57
4-3 The "fast clone" of the compiled Fibonacci method, with support for exceptions 59
4-4 The lookup table method which determines where an exception will be caught 63

List of Tables

3.1 The states of a Closure. 52

5.1 Running times of Java and JCilk programs. 73

5.2 Speedup of JCilk programs. 73

5.3 Performance running `Fib` with one worker on a multiprocessor machine 74

Chapter 1

Introduction

With such recent innovations as multiprocessor machines, multicore processors, and hyperthreading, more and more desktop machines have some ability to execute parallel programs. Writing programs to take advantage of that available parallelism remains, however, a difficult task. New programming paradigms, developed over the past few decades to make large-scale programming easier, tend not to give much aid to the parallel programmer. In this thesis, I explain how two particular languages, Java [13] and Cilk [11], can be fused into a new language called JCilk (pronounced "jay-silk") which simplifies parallel programming by incorporating both Java's modern language features and Cilk's simple parallel semantics.

Over the past year, I and the rest of the JCilk design team have collectively hammered out the multitude of semantic questions to produce the JCilk-1 semantics described in Chapter 2 of this thesis. We have also developed a system for compiling and running JCilk programs. As part of that development, I have implemented the JCilk-1 Runtime System, which is the main focus of this thesis. My collaborator, Angelina Lee, has implemented the complementary portion of the system, the JCilk-1 Compiler. That work is described in her upcoming thesis [22].

Before I explain in detail how JCilk combines Cilk and Java, it is important to understand what makes each of the two languages independently valuable. In this chapter, I give an overview of what JCilk takes from the two languages. Java gives several significant features that make programming easier, including portability, auto-

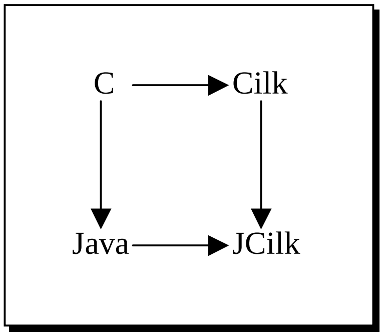

Figure 1-1: The ancestors of JCilk. Java extends C with many modern language features. Cilk extends C with parallel language features. JCilk does for Java what Cilk does for C, and at the same does for Cilk what Java does for C.

mated memory management, and exceptions. Cilk offers a provably-good threading mechanism and speculative execution. These features mesh together to give JCilk the combined power of both languages, though some don't interact in a completely straightforward manner. In particular, letting exceptions occur in a parallel environment introduces new complications and raises new questions. JCilk answers those questions with a novel and powerful exception mechanism, in which exceptions are used to support speculative execution. I conclude the chapter by explaining this new mechanism and giving an outline of the thesis.

1.1 Java

The Java language was itself derived from an earlier language, C, which is described in [20]. Java inherits its basic syntax from C, but adds to it many features designed to make programming easier: portability, automatic garbage collection, object orientation, exceptions, threading, and so on. This section describes some of those features, including Java's version of parallelism built on static threads.

Portability

Portability was one of the major design goals of the Java language [13, p.219]. Rather than compiling to a native binary, the Java compiler outputs a program in an intermediate language called Java bytecode. This language is then interpreted by the Java Virtual Machine (JVM) [23]. Since this process precludes any platform-dependent code, implementing the JVM on a machine platform once suffices to be able to later run any Java program on that platform. Since at least one implementation of the JVM already exists for most major and minor platforms, a language based on Java can run basically anywhere.

Automatic Garbage Collection

The JVM is specified to contain an automatic garbage collector [23, Sec. 3.5.3]. In other languages such as C, every pointer must be meticulously tracked and eventually deallocated in order to avoid memory leaks. Java, on the other hand, restricts what pointers a programmer can access, always keeps an eye on what memory is being used, and efficiently deallocates memory when it has been abandoned. This automation simplifies reasoning about the safety of memory accesses and the total amount of memory being used, and it eliminates any need to deallocate memory.

Object Orientation

Although portability and automatic garbage collection are important properties of the language, Java's most visible feature is that it is completely object-oriented. Aside from a few primitive types, every piece of data in a Java program is represented by an object. Every procedure in a Java program is expressed as a method belonging to a particular class. The object-oriented paradigm eases the writing of modular code, by making a clear division between the work associated with different kinds of data. It also encourages code reuse by allowing one class to extend another class (its "superclass") and inherit methods from it without reimplementing them.

Exceptions

Exceptions give a way to indicate an unusual or abnormal situation and allow the situation to be handled outside the normal control flow of the program. When a Java program encounters an unusual situation, it can "throw" an exception rather than trying to correct the problem locally. That exception is then "caught" by a handler at some other point in the code. Java implements the termination model of exception handling [7], under which all work intervening between the throw and the catch is terminated.

Static Threading

Java also contains some built-in support for multithreaded programs. This support takes the form of ***static threads***, `Thread` objects which are constructed with a method that they should execute. Once a method begins on one thread, it always completes on the same thread. This style of threading is especially suited to any environment with persistent concurrent tasks: displaying multiple independent animations, processing input while doing background computation, and so on.

1.2 Cilk

The Cilk language also extends C, but in mostly perpendicular directions. Its main goal is to support "dynamic threading," an alternative threading model which gives more flexibility to the runtime scheduler in order to obtain provably good performance. This section introduces Cilk's expression of this idea. Cilk also contains several supporting features that are necessary to make dynamic parallelism practical. This section touches on two of those: thread atomicity, which enables reasoning about execution of procedures despite the potential for nondeterminism; and speculative execution, which allows extraneous work to be aborted.

Dynamic Threading

Static threading is inconvenient for expressing large computations involving an arbitrary number of processors. To see why, imagine that you have a large computation to complete and have several processors at your disposal. You don't care how the work is divided up, but you want it all to get done eventually. Any division you might make at compile-time to divide the work up among processors would be arbitrary. To

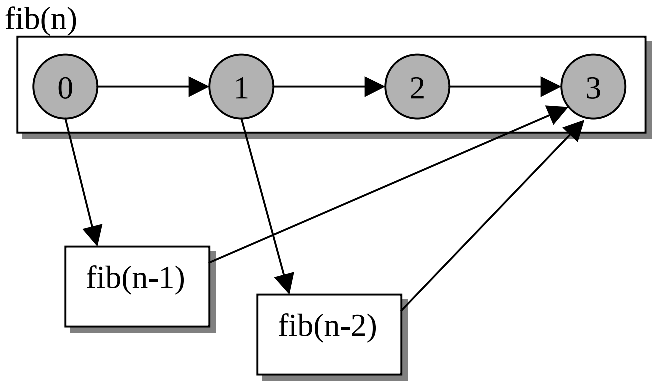

Figure 1-2: A Cilk computation DAG. A thread precedes all threads which it points to. This particular DAG shows the execution of the program in Figure 1-3.

create one thread explicitly for every small subcomponent of the task would drown your work in the overhead of scheduling all the low-level thread objects, but to leave all of the work in a single thread would squander the resources available on your computer.

To work around this problem, the dynamic threading model instead divides the work up into many small "logical threads," which may execute in parallel at the scheduler's discretion. It does *not*, however, create a heavyweight low-level thread objects for each of them. Instead, the threads are dynamically distributed by the scheduler across a small number of heavyweight threads, which the scheduler can do at runtime when full information about processor load is available.

Cilk implements dynamic parallelism with a fork-join model, using the `spawn` and `sync` keywords. As the program runs, its threads are connected by a dependency DAG, such as the one shown in figure Figure 1-2. A thread does not begin to execute until all of the threads which logically precede it have completed. The logical threads in a Cilk program are ***Cilk threads***, pieces of code which execute serially, that is, maximal sequences of executed statements that do not contain one of the parallel keywords.

In a Cilk program, a statement can be preceded by the keyword `spawn`, as in line 7 of Figure 1-3. This keyword indicates that the procedure being ***spawned***, here `fib(n-1)`, can execute in parallel with the remainder of the procedure spawning it,

```
 1   cilk int fib(int n) {
 2     int x, y;
 3     if(n < 2) {
 4       return n;
 5     }
 6       /* thread 0 */
 7     x = spawn fib(n-1);
 8       /* thread 1 */
 9     y = spawn fib(n-2);
10       /* thread 2 */
11     sync;
12       /* thread 3 */
13     return x + y;
14   }
```

Figure 1-3: A recursive Cilk procedure to find Fibonacci numbers, according to the formula $fib(n) = fib(n-1) + fib(n-2)$.

here `fib(n)`. For brevity, we often refer to spawned method as the ***child***, and the spawning method as the ***parent***.

Only after a `sync` statement, such as the one in line 11, can the program be assured of seeing the state after the spawned procedures complete. The `sync` statement acts as a join; execution does not proceed in `fib(n)` until all of the procedures spawned from `fib(n)` (that is, all the children of `fib(n)`) have completed execution and their return values have been received.

This style of parallelism works well for computations with a large but fixed amount of work. A method to recursively compute the Fibonacci numbers or to compute matrix products ([32]) would fall into this category. This easily-achievable parallelism is the fundamental power that JCilk intends to carry over into Java.

Provably Good Scheduler

Cilk takes advantage of its dynamic parallelism to implement a scheduler which is provably good. That is to say, the Cilk designers have proven (in [11]) that a program running under the Cilk scheduler on an arbitrary number of processors will complete

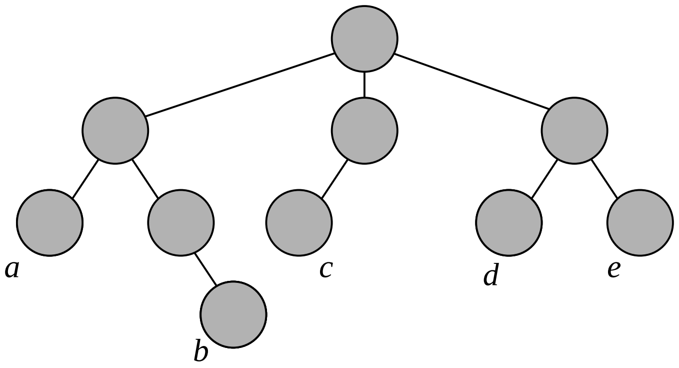

Figure 1-4: A parallel search tree. Every node represents a procedure called in the parallel search, and the edges, represent those calls. The nodes labeled a through e are executing simultaneously.

in time within a constant factor of the optimal time the program would have taken if it were being run by an omniscient scheduler. The implementation details of Cilk's scheduler can be found in [11]. Chapter 3 shows how I have adapted it into JCilk.

Thread Atomicity

In order to simplify reasoning about the interactions among threads, Cilk implements atomicity between threads belonging to the same procedure. Two threads in the same procedure never run simultaneously or interleave; execution always proceeds according to some sequential order of threads. This constraint lets the programmer reason about a Cilk program's local execution without having to worry about data races, although it makes no guarantees about data races between two different procedures.

Speculative Computation

A major gap in the dynamic parallelism model, as I have described it so far, is the lack of support for speculative computation. This facility is important for parallelizing programs such as branch-and-bound or heuristic search [9], in which some computations likely to contribute to a solution might turn out to be unnecessary or redundant after they have been spawned. Once the program learns that a subcomputation is unnecessary, it should abort that subcomputation to avoid wasting processor time.

As an example, consider the search tree in Figure 1-4, which shows five processors concurrently working pieces of a parallel search. Imagine that the processor node *a* finds a solution that is better than any solution that could possibly exist at node *b*; in this case, the processor working at node *b* might as well give up. Similarly, if a processor were to find a solution it knows to be the global optimal, than all of the other processors should stop their work.

Cilk supports speculative computation by extending what can be done with the result of a spawned procedure. Specifically procedure `f` can be called with an ***inlet***: a local procedure that executes when `f` returns. In the inlet, the program has the opportunity to examine the return value and take any appropriate action, including aborting any remaining subcomputations.

The `abort` statement (which is generally executed in an inlet) initiates the abort process. This process is essentially opaque from the programmer's point of view. It traces the spawn tree, recursively aborting all children of the procedure which initiated the abort, and all of its children's children, and so on. An aborted procedure has no chance to clean up; it simply halts and vanishes from the program's perspective. The only procedures to be aborted are those that were previously spawned and not completed, and the programmer must take explicitly care not to later spawn more side procedures.

Compared to the elegance and simplicity of the `spawn`/`sync` mechanism, this abort protocol is complex, awkward, and opaque, making it a prime target for improvement in JCilk.

1.3 JCilk

This thesis describes how JCilk extends Java to include support for dynamic parallelism via the `spawn` and `sync` statements, along with all of the other features of Cilk. This extension is, in fact, a faithful extension: eliding the parallel keywords from a JCilk program leaves a correct serial Java program (the "serial elision"), and the JCilk program running on one processor gives the same behavior as the serial elision would.[1] In general, the two languages peacefully coexist in JCilk, but there are some conflicts between the two sides. One example, discussed first in this section, is static versus dynamic threading, which is resolved in favor of Cilk's mechanism. Exceptions present a more interesting picture, giving a new way to handle speculative execution. I conclude the section by giving an outline of the remainder of the thesis.

Threading

Since the motivation behind JCilk is to add dynamic threading to Java, JCilk naturally uses Cilk's dynamic threading as its primary threading mechanism. What does that imply about JCilk's ability to support static threading as well? At the moment, that remains an open question. The JCilk-1 implementation simply disallows any use of Java's built-in threading mechanism in a JCilk program, but in Section 7.1 I give some thoughts on how the two styles of threading could interact in future versions of JCilk.

Exceptions and Aborting

It's not immediately clear how exceptions should behave under dynamic threading. In particular, when a method throws an exception to its parent, how should it be handled? It turns out that, using the semantics proposed for JCilk, exceptions provide the key to crack the abort dilemma. To see how, let's reconsider the goals of the abort mechanism.

First, we want a way to signal that the remaining children of a method are no longer necessary. Any which have already been spawned but have not completed should be halted, and any which have not yet been spawned should be skipped. That sounds remarkably like what would an exception does by terminating blocks between its throw-point and its handler, skipping all subsequent statements. The abort mechanism also needs to retroactively "skip" procedure calls still in progress

[1] This claim isn't strictly true in all cases, because the shortcut assignment operators (such as `+=`) follow must slightly different semantics to allow them to interact correctly with `spawn` statements. The serial elision of the statement `x += spawn f()`, for example, should be `x = spawn f() + x`.

by aborting them; this idea gives some hints of what exceptions should mean in a parallel context.

Second, we want a way for a parent to signal all of its descendants that they should abort. Exceptions don't help with getting the abort signal to the children in the first place, but they do help to add transparency once it gets there. After all, sudden and inexplicable failure (as appears to occur at an abort in Cilk) is exactly the kind of situation that exceptions are intended to avoid. Thus in JCilk, once the parent has traced down the spawn tree to each outstanding spawned procedure, instead of halting execution immediately, it merely causes an exception (a `CilkAbort`, to be precise) to be thrown in those procedures.

This thesis argues that an exception-based abort mechanism is much more organic and elegant than an inlet-based one. Modifying Cilk to support this functionality is a natural next step in the evolution of the language. Adding exceptions directly into Cilk is unreasonable, however, since the C language contains no concept of exceptions. Thus, we are back to where we started: at Java, which does support exceptions. By adding the Cilk parallel language features into Java, we create a new language JCilk which gives exceptions to Cilk and gives dynamic threading to Java—improving on both languages.

Structure of this Thesis

This thesis begins by going into more detail about the JCilk language. Chapter 2 gives a more through discussion of JCilk's new exception-handling semantics in particular, including a formal expression of those semantics.

The remainder of this thesis focuses on the implementation of JCilk and, in particular, the implementation of the JCilk Runtime System. Chapter 3 describes the underlying work-stealing model, which was ported over from C to Java essentially intact but with several implementation details changed. Chapter 4 dives into the runtime system modifications to support JCilk's exception-handling semantics, and I explain my implementation of them.

Certain aspects of the JVM make an efficient implementation of the runtime system difficult. In Chapter 5, I explain how these affect JCilk's performance and attempt to improve that performance. In Chapter 6, I discuss how JCilk fits into the family of parallel languages. Finally, in Chapter 8, I suggest future directions in which JCilk might go.

Much of this thesis represents collaborative work with Angelina Lee and Charles E. Leiserson. Chapter 2, in particular, was based on our joint written work.

Chapter 2

JCilk Semantics[1]

What actually happens when an exception is thrown? How can a programmer tell what will be aborted an what won't? All of these questions require a full specification of the semantics of the JCilk language, which this chapter provides. In cases where there are no parallel interactions, JCilk's semantics are the same as those of Java. An additional part of the semantics, covered in Section 2.1, is inherited from Cilk: the behavior at `spawn` and `sync` statements and the meaning of the `cilk` keyword.

The new exception semantics, in which parallel "side computations" can be aborted, are entirely unique to JCilk, although JCilk does maintain Java's exception syntax and is consistent with Java's semantics for serial executions. Section 2.2 describes how Java exceptions work and gives the baseline for understanding JCilk exception behavior. JCilk provides "semisynchronous" aborts to simplify the reasoning about program behavior when an abort occurs. JCilk also allows aborts themselves to be caught by defining a new subclass of `Throwable`, called `CilkAbort`, thereby allowing programmers to clean up an aborted subcomputation. The last two sections of this chapter explain those ideas.

[1] This chapter describes joint work with Angelina Lee and Charles E. Leiserson.

```
public class Fib {
  public static cilk void 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 2-1: A simple JCilk procedure. Compare to Figure 1-3

2.1 Basic JCilk Semantics

In JCilk, as in Cilk, a program expresses its parallelism through the `spawn` and `sync` statements described in Section 1.2. This section describes these keywords and their ramifications. First I give more complex usages of the `spawn` keyword which require new threads called inlets to execute. Then I explain the `cilk` keyword, which indicates parallel code, and the concept of the locus of control, which can be used to describe a JCilk program's execution.

Spawn and Sync

The examples of Cilk procedures given in Section 1.2 transfer to JCilk essentially without modification. For example, the sample procedure given in Figure 1-3 can be adapted into the JCilk method in Figure 2-2 with minimal modifications: only the trappings of Java's object-orientation must be added.

As in Cilk, the `spawn` keyword also appears in slightly more complex contexts not discussed in Section 1.2. Lines 5–7 of Figure 2-2 show alternative ways to spawn a procedure when the procedure being spawned returns a value. The calls to `B` and `C`, because they are spawned, are logically in parallel with the remainder of `main`.

```
 1   public class Demo {
 2     public static void main(String[] argv) {
 3       int x = 0, y = 0;
 4       spawn A();
 5       x = spawn B(2);
 6       y += spawn C(4, 5);
 7       y += spawn C(6, 7);
 8       System.out.println(x + " " + y);

 9       sync;
10       System.out.println(x + " " + y);
11     }
12   }
```

Figure 2-2: More complex spawns in JCilk.

The result of the call to `B(2)` is stored into `x` after the call completes. Similarly, `y` is increased by `C(4, 5)` after that call completes. Because line 8 is logically in parallel with the execution of `B` and `C`, the output that appears from line 8 is nondeterministic; it could see the old values, the new values, or any combination thereof.

Inlets

Although JCilk does *not* allow an arbitrary inlet method to be executed when a spawned method completes.[2], the concept of an inlet remains to support the assignment operators. For every assignment of the form `x = spawn f()` or `x += spawn f()` the operation (if any) and the assignment for the spawn statement are implicitly performed as an inlet. The inlet counts as its own thread, so that it can execute long after the original `spawn` statement it is a part of has finished. It is also considered part of the parent method, so it executes atomically with respect to the other threads in the parent method. Without the atomicity guarantee, the two `+=` statements in lines 6 and 7 could read `y` before either writes, rendering the `+=` idiom useless when dealing with `spawn` statements.

[2] Similar behavior can, however, be obtained through the catchlet and finallet mechanisms described in Section 2.4.

The Cilk Keyword

JCilk inherits from Cilk one other keyword related to parallelism: the `cilk` keyword itself. Every method that is used as a "cilk method," (that is, every method that can be spawned and can spawn other methods) must be declared with the `cilk` modifier. This keyword indicates that the method could be run in parallel with other methods, and lets the programmer now and that he or she should carefully consider parallel effects accordingly. A non-cilk method cannot call or spawn a cilk method, but cilk methods can call non-cilk methods (such as those in the Java APIs).

The Locus of Control

To follow along with a particular execution of a Cilk or JCilk program, we often describe a ***locus of control***, a point at which execution can occur, similar to the concept of a program counter in an assembly-language program. Every `spawn` statement creates a new locus of control belonging to the child method, called that method's ***primary*** locus of control. That locus of control moves through the child until that method completes. It then returns to the parent method to execute the inlet (if any) and finally disappears. We can thus express thread atomicity by saying that only one locus of control at a time can execute in one method at a time. Any locus of control executing an inlet in a method is considered a ***secondary*** locus of control in that method.

2.2 Exceptions in Java

Java contains an object-based exception mechanism which gives an alternate way for a block of code to conclude. Rather than exiting the block normally (or returning a value, if the block is a method), the block can throw an exception: an object, and in particular, an instance of some subclass of `Throwable`. An exception is generally thrown to indicate that an abnormal or illegal event has occurred and that the method which is throwing the exception has not completed normally. (Indeed, such a

```
 1   int read(int n) throws IOException {
 2     if(n < 0 || n > 5) {
 3       throw new IndexOutOfBoundsException();
 4     }
 5     data[n] = System.in.readln(); // throws IOException
 6     return data[n];
 7   }

 8   public static void main(String argv[]) {
      ⋮
 9     try {
10       int x = read(n);
11       System.out.println("Read " + x);
12     } catch(IOException e) {
13       // Handle bad file, etc.
           ⋮
14     } catch(IndexOutOfBoundsException e) {
15       // Handle bad user input of n.
           ⋮
16     }
17     // Continue execution
         ⋮
18   }
```

Figure 2-3: Two simple Java methods. Method `read` takes an integer. If that integer is out of range, it throws an exception. Otherwise, it attempts to read from standard input, which could itself throw an exception. Method `main` calls `read` and handles both kinds of exceptions.

method is formally described as having completed abruptly [13].) `try-catch-finally` statements are used for exception-handling in JCilk, just as they are in Java.

Figure 2-3 gives an example of a Java program utilizing exceptions. In this program, there are two ways for method `read` to complete.

In a "normal" completion, the return statement in line 6 executes. The return value is stored into `x` in line 10, and is printed in line 11. Since exception is thrown, the two `catch` clauses do not execute. After the `try` block is done, execution immediately skips to after the end of the `try-catch` statement, that is, to line 17.

Sometimes, however, something will go wrong somewhere, causing the method to complete "abruptly." Rather than failing outright, the method throws an exception to allow another part of the program (which may be better equipped to understand the failure) to handle the failure. In this case, method `read` could either throw an exception directly (as in line 3) or propagate an exception that it cannot handle itself (if the call to `readln` in line 5 throws an exception). In both cases, that exception is passed back to `main`.

When `main` receives an exception from `read`, it does not store any value into `x`. Instead, execution in `main` immediately leaves the `try` block and enters one of its `catch` clauses. Notice that since execution leaves the `try` block, line 11 is skipped and the `try` block's output is not printed. Instead, the body of the `catch` clause executes.

In general, when an exception is thrown, the Java Language Specification [13] states:

> When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically-enclosing catch clause of a try statement (§14.19) that handles the exception.
>
> 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...
>
> The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a catch clause is encountered that can handle the exception; execution then continues by executing the block of that catch clause. The code that caused the exception is never resumed.

```
cilk try {
   spawn f();
} catch(Exception e) {
}
sync;
```

Figure 2-4: The simplest way to catch an exception in JCilk.

```
cilk int f1() throws Exception {
   int w = spawn A();
   int x = B();
   int y = spawn C();
   int z = D();
   sync;
   return w + x + y + z;
}
```

Figure 2-5: A simple JCilk program using exceptions.

2.3 Exceptions in JCilk

JCilk retains Java's exception syntax and its general exception behavior, and extends them to encompass the cases where exceptions are thrown while code is executing in parallel. This philosophy means that an exception thrown in JCilk will act exactly like one thrown in Java. This section begins by exploring this idea. The main new feature introduced to the language is the concept of aborting, represented by the `CilkAbort` exception. These simple cases can be understood without even having to worry about how the exception is caught. For this section, I assume that every exception thrown in the JCilk program is caught using a `cilk try` block as in Figure 2-4. In Section 2.4 I cover more complex cases.

Philosophy

The design of JCilk strives to preserve Java's exception semantics while extending them to cope gracefully with the parallelism provided by the Cilk primitives. In particular 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. Thus, for example, in Figure 2-5, if `A` and/or `C` is still executing when `D` throws an exception, then they are aborted.

A little thought reveals that the decision to implicitly abort side computations opens a Pandora's box of subsidiary linguistic problems to be resolved. Aborting might cause a computation to be interrupted asynchronously [13, Sec. 11.3.2], causing havoc in programmer understanding of code behavior. What exactly gets aborted when an exception is thrown? Can the abort itself be caught so that a spawned method can clean up? Can the mechanism be implemented efficiently?

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 is, 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 a `CilkAbort` to be thrown in the method. The programmer can choose to catch a `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 parallel control must be understood anyway. For example, in Figure 2-5 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 `f1` 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

[3] In keeping with the usage in [13], when I refer to an exception, I mean any instance of the class `Throwable` or its subclasses.

```
cilk void f2() {
   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 2-6: Catching a `CilkAbort`.

primary locus of control resides at *some* thread boundary, and likewise for secondary loci of control.

Handling aborts

JCilk also give the programmer more flexibility in reacting to an abort. In the original Cilk language, when a side computation is aborted, it just halts and vanishes without giving the programmer any opportunity to clean up partially completed work. In contrast, 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.

JCilk exploits Java's exception semantics to provide a natural way for programmers to handle `CilkAbort` exceptions. A program can catch the `CilkAbort` exception and restore any modified data structures to a consistent state. The code in Figure 2-6 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 aborted. Any spawned methods that abort have their corresponding catch blocks executed and, in this case, their cleanup methods called.

2.4 Advanced JCilk Exceptions

Exceptions can be thrown into more complex contexts than the one given in Figure 2-4. In particular, the `cilk try` statement might not be immediately followed by a `sync` statement, in which case the following statements might be executed before an exception is thrown. For these cases, the concept of an inlet must be extended to support a thrown exception. A `cilk try` block that contains multiple `spawn` statements also complicates the question of what, exactly, should be aborted. This complication extends even further when the `CilkAbort` exception itself is being caught as well. In this section I confront all of these questions and give the final pieces of the JCilk exception semantics.

The `cilk try` statement

Figure 2-7 shows an example of how the `cilk try` statement interacts with the spawning of subcomputations. The parent method `f3` 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. As before, the primary locus of control continues in `f3` until it hits the `sync` in line 13, at which point `f3` is suspended until the two children complete.

Observe that `f3`'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 this ability were not present and the primary locus of control were held up at the end of every `cilk try` block, then writing a `catch` clause would always 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 `f3` 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 `f3`) of the method (in this case `A`) that threw the exception. As with

```
cilk int f3() {
   int x, y;
   cilk try {
      x = spawn A();
   } catch(Exception e) {
      x = 0;
   }
   cilk try {
      y = spawn B();
   } catch(Exception e) {
      y = 0;
   }
   sync;
   return x + y;
}
```

Figure 2-7: Handling exceptions with `cilk try` when aborting is unnecessary.

ordinary inlets, JCilk guarantees that the catchlet runs atomically with respect to other loci of control running on `f3`.

Similar to a catchlet, a ***finallet*** runs atomically with respect to other loci of control if the `cilk try` statement contains a `finally` clause.

Aborting side computations

We are now ready to tackle the full semantics of `cilk try`, which includes the aborting of side computations when an exception is thrown. We refer back to one key concept in the Java language specification [13, 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," what method instances should be aborted. (See the quotation in Chapter 1.) Specif-

```
 1   cilk int f4() {
 2      int x, y, z;
 3      cilk try {
 4         x = spawn A();
 5         y = spawn B();
 6      } catch(Exception e) {
 7         x = y = 0;
 8         handle(e);
 9      }
10      z = spawn C();
11      sync;
12      return x + y + z;
13   }
```

Figure 2-8: Handling exceptions with `cilk try` when aborting might be necessary.

ically, when an exception is thrown, JCilk delivers a `CilkAbort` exception semisynchronously to the ***side computations*** of the exception. The side computations include all methods that are 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, no `CilkAbort` is caught in a to-be-aborted `cilk` block until all that block's children have completed, allowing the side computation to be "unwound" in a structured way from the leaves up.

Figure 2-8 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 `f4`, unless it has already exited the `cilk try` statement. It does not include `C`, which is not dynamically enclosed by the `cilk try` block.

JCilk makes no guarantees that the `CilkAbort` is thrown quickly (or even at all) after 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`

When an exception is thrown, when and how is it handled? 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 after both Actions 3 and 4 complete.

If multiple concurrent exceptions are thrown to the same `cilk` block in JCilk, only one 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 [13, 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.

2.5 The Queens problem

To demonstrate some of the JCilk extensions to Java, this section illustrates how the so-called "Queens" puzzle can be programmed. 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 2-9 shows how a solution to the queens puzzle can be implemented 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 program highly parallel.

The program 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 makes this strategy easy to implement.

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 `q` method to search for a safe configuration. Method `q` 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. It loops through all columns in the current row to find safe positions to place a queen in the current row. The regular Java method `safe`, 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 `q` method is spawned to perform the subsearch

```
public class Queens {
   private int n;
   ⋮
   private cilk void q(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 q(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)).q(cfg, 0);
      } catch(Result e) {
         ans = (int[]) e.getValue();
      }
      sync;

      // At this point, the answer is in ans.
      ⋮
   }
 }
```

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.

with the new queen placed in the position (row, col).

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

The program exploits JCilk's implicit abort semantics to avoid extraneous computation. When one legal placement is found, some outstanding q 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 statement. In this example, since the Result exception propagates all the way up to the main method, all outstanding q methods are aborted automatically. Notice that there is a sync statement in the main method before it proceeds to print out the solution to ensure that all side computations have terminated.

Chapter 3

The Work-Stealing Scheduler

The implementation of JCilk is, following the example set by Cilk [11], divided into two major components: the Compiler and the Runtime System. Although a fully-featured JCilk compiler certainly could be written to produce a Java bytecode output, with all of the JCilk scheduler features directly inserted into that bytecode, that implementation would be needlessly complicated. Instead, most of the scheduler is implemented as methods in the runtime system, and the compiler (as one of its tasks) adds the appropriate runtime system calls into the compiled code. Section 3.1 discusses this process and gives an example of its results. The scheduler itself uses a work-stealing algorithm in which each processor steals work from another processor whenever it completes its own work. The runtime system is implemented with three major classes, described in the remainder of this chapter: Workers, which provide an interface to the runtime system and which manage the process of work-stealing; Frames, which maintain a shadow of the call stack to allow stealing to occur; and Closures, which represent frames that has been stolen and allow children to return their values to their parents.

3.1 The Compiler

A JCilk program is compiled in two stages: first, from JCilk to an intermediate language GoJava, and second, from GoJava into Java bytecode. For the purposes

```
private cilk 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 3-1: A JCilk method to compute the nth Fibonacci number.

of this thesis, only the first stage is significant; the main feature of the compiler is that it creates the interface between the user code and the runtime system. To start things off, it creates a `main` method which initializes the runtime system. It replaces parallel keywords with calls into the runtime system, allowing the program to inform the scheduler of its current status and to find out the scheduler's current status. The compiler also uses static analyis to create the helper procedures and lookup tables, which are used whenever the runtime system needs to call a method in the user's program or find out information about the structure of the user's program. Finally, the compiler maintains more complex information about variable usage which is beyond the scope of this thesis; see the forthcoming thesis of Angelina Lee for more information.

The first phase of the compilation process is a source-to-source compilation, from JCilk to an intermediate language called GoJava. This language is, in itself, an extension of Java. It adds to Java very limited use of the `goto` keyword, which is a reserved keyword in Java but is never used. Such support for `goto` is necessary to include a low-overhead continuation mechanism into Java, which in turn is necessary for the thread migration mechanism described in Section 3.2.

The runtime system and the compiled code share a narrow interface. Compiled-in method calls like the ones in Figure 3-2, provide the main part of the interface. At

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

   Fib_fib_frame thisFrame = new Fib_fib_frame(n, 0, returnEntry);
   worker.pushFrame(thisFrame);

   if( n < 2 ) {
     return n;
   }

   x = this.fib(worker, n-1, 1);
   if(worker.popFrameCheck(new Integer(x))) {
      return 0;
   }

   thisFrame._x = x;
   y = this.fib(worker, n-2, 2);
   if(worker.popFrameCheck(new Integer(y))) {
      return 0;
   }

   return x + y;
}
```

Figure 3-2: A simplified version of the “fast clone” of the compiled Fibonacci method.

the other end, the runtime makes a single initial call into the user code's `main` method to get the program started. These are the only places where the compiled code and the runtime system interact. The user's compiled code doesn't need to worry about scheduling; it simply needs to call into the runtime system every so often to make sure that everything is in order.

For every method, the compiler also outputs several helper methods which are used by the runtime system. These belong to the Frame class (see Section 3.3) corresponding to the method they are annotating. The most commonly used such method is `cilkRun`, which calls the user method that the frame belongs to. This extra level of procedural indirection allows the runtime system to call any user method without having to explicitly use reflection to look up its name.

The Two Clones

Again following the example set by Cilk, each method in a JCilk program is compiled into two different copies (or "clones") of that method. One, the ***fast clone***, is streamlined. It is the clone that is intended to be executed most of the time, and is in fact the only clone which is ever directly called by the compiled GoJava program. It contains only the bare minimum of parallel support, as can be seen in Figure 3-2.

The ***slow clone*** contains much more support for parallel execution, as shown in Figure 3-3. It accesses all of its local variables through the frame to ensure that it always has the most up-to-date versions of those variables. In more complex methods (like those we consider in Chapter 4), other overheads are also be added to the slow clone.

The main feature of the slow clone, though, is its continuation support. It is only called by a thief which has stolen the method and is now continuing it. It takes a special frame argument that the thief passes in, which contains the saved local state from the previous execution of the method on the victim. It also contains a ***program counter*** (PC) which determines the point at which execution should continue. A `switch` statement (like the one in line 5) branches to the appropriate label in the method.

```
private void fibSlow(Worker worker, CilkFrame frame) {
   int tmp;
   Fib_fib_frame thisFrame = (Fib_fib_frame)frame;

   switch(thisFrame._pc) {
   case 1:
      goto _cilk_sync1;
   case 2:
      goto _cilk_sync2;
   case 3:
      goto _cilk_sync3;
   }

   _cilk_sync1: ;

   thisFrame._pc = 2;
   tmp = this.fib(worker, thisFrame._n-2, 2);
   if(worker.popFrameCheck(new Integer(y));
      return;
   } else {
      thisFrame._y = tmp;
   }
   _cilk_sync2: ;

   thisFrame._pc = 3;
   if(!worker.sync()) {
      return;
   }
   _cilk_sync3: ;

    retVal = x + y;
    worker.setReturnResult(new Integer(retVal));
    return;
}
```

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

Notice that the slow clone contains `goto` statements, which are not valid statements in Java. Although the Java language supports many goto-like idioms, such as breaking out of a named loop, it provides no way to jump *into* a loop. This is where GoJava comes in. GoJava contains (by design) support for `goto` statements in exactly the ways that the continuation mechanism needs [22].

3.2 Workers

The JCilk implementation contains a provably good scheduler built on top of a work-stealing algorithm. (For more details on Cilk's implementation, from which the JCilk implementation is adapted, see [11].) At its lowest level, the JCilk runtime system is composed of a number of Java `Thread` objects, known as the ***workers***, which are described in this section. Each worker represents one processor on which the program is running. The code statically assigned to each worker `Thread` is a simple loop: find work to do, do it, repeat.

Where does this work come from? Initially, it is passed in by the `main` method generated by the compiler that the compiler generates; this first piece of work represents the `main` method of the user code. One worker begins by executing that method from top to bottom. Whenever it encounters a `spawn` statement, it first executes the child method, then returns normally and continues the parent method, exactly as in the serial execution. If there is only one worker, the runtime system doesn't have to do anything else. Eventually the worker reaches the end of the JCilk program's `main` method and the program terminates.

On multiple workers, the story gets more complicated. The workers beyond the first have no initial method to execute, and instead attempt to steal work. When a worker W (the thief) attempts to steal work, it first randomly chooses another worker V to be its victim and queries that worker to see if it has any work available to steal. If V is currently in the middle of executing some spawned method `B`, as it is in Figure 3-4, then that method's parent `A` is available to be stolen. Worker W then continues executing the user's code for `A` from where V left off, that is, immediately

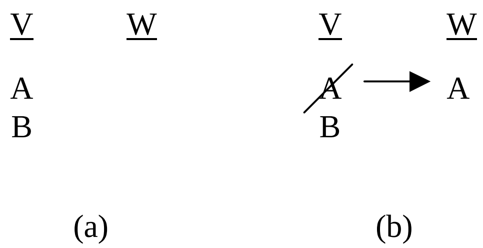

Figure 3-4: An example of work-stealing. In (a), before the steal occurs, `A` and `B` are on worker V's call stack, while worker W has no work. In (b), W has stolen `A`.

after the `spawn` statement which spawned `A`. (Recall that a `spawn` statement indicates that the child method `B` can run in parallel with the remainder of its parent `A`, which is exactly what happens here.) This theft introduces two ways for a worker to run out of work and regress to stealing.

First, as W executes its new method `A`, it might encounter a `sync` statement. If encounters a `sync` statement before V finishes executing `B`, then we say that `A` still has an outstanding child. The `sync` fails and `A` has to be suspended until `B` finally finishes. In this case, W is left with no work to do, so it goes stealing.

On the other hand, V might also finish `B` before W reaches a `sync` statement. In this case, V cannot follow its normal control flow and return to executing `A`, because `A` is already being executed by W. Instead, V attempts to steal work from another worker.

Notice that in all of these cases, any given JCilk thread executes from start to finish on a single worker. Only at the boundaries between threads can the method be migrated or stopped.

The worker also serves another purpose. Every cilk method is called with the worker object which is currently executing that method. That worker argument gives the compiled user program its only interface the compiled user's code has to the runtime system; every call to the runtime system goes through the worker.

3.3 Frames

The discussion in Section 3.2 takes a lot for granted. In particular, the thief needs some way to access the local state of the method it is trying to steal. That means that the method must already have stored a copy of its local state before it spawned its child. To support this extra storage requirement, every `spawn` statement in the original JCilk program must expand into several statements in the compiled program to save the program state, call the spawned method, and then confirm that a steal hasn't occurred. This section describes that expansion. In particular, it explains how the saved state goes on the "ready deque," which presents an interface to allow stealing to occur.

Spawning and Continuations

To allow a method which was begun on one worker to be continued by another worker after a `spawn` statement, the compiler (like the Cilk compiler) expands that statement into four stages:

1. **Save** all local state into public data structure, so that the thief can access the correct values for all local variables. (Line 12 in Figure 3-2.)
2. **Call** the child method. (Lines 8 and 13.)
3. **Confirm** that the parent method was not stolen while the child was running. If it was, halt execution and pass the child's return value (if any) to the runtime system. (Lines 9 and 14.)
4. **Continuation:** present a label so that another worker can continue from immediately after the spawn. (In lines 13, 21, and 26 of Figure 3-3.)

The Confirm stage of a spawn highlights the difference between the logical execution of the JCilk program and the actual execution of the Java method. Java itself clearly doesn't support method migration; When a method like `A` is stolen, it still remains on the victim `Thread`'s Java call stack, and the victim returns to it when it finishes executing `B`. Only with the explicit call into the runtime system does `A` learn from the worker that it has been stolen. Once it finds out that it has been stolen, it immediately halts by returning 0 as in lines 10 and 15.

The Ready Deque

The Save and Confirm stages interact with the worker's ***ready deque***, another concept borrowed from Cilk, which is a deque of the frames belonging to that worker. Each individual frame is an instance of some subclass of the `CilkFrame` class. Whenever a cilk method is compiled, the compiler also produces a corresponding subclass of `CilkFrame` containing, as instance fields, a shadow copy of each of that method's local variables. The subclass also overrides the abstract `cilkRun` method of `CilkFrame` to call its corresponding method, allowing a thief to call `CilkFrame.run()` to execute the stolen method. (The `CilkFrame` class also contains several other methods, which are discussed further in Chapter 4.)

Naturally, the ready deque grows and shrinks as the JCilk program executed. It initially acts as a "shadow stack," perfectly mirroring Java's internal call stack. It is necessary because Java provides no mechanism for accessing the information about the call stack that a thief needs to know. When a method is first spawned, it instantiates its own particular frame and pushes that frame onto the ready deque. Whenever it spawns a child, the Save stage of the spawn ensures that its frame on the deque is up to date with the current values of all its local variables. When the method finally returns, its is popped off of the deque.

Of course, the ready deque wouldn't be much of a deque if it were only accessed at one end. The common case for accesses to the deque is indeed the case already described: spawns and returns pushing and popping at the "working end." A thief, however, tries to access the other end of the deque: the "stealing end."

As it steals, the thief first examines the victim's deque to ensure that it contains enough frames for one to be stolen. If each method pushes its own frame when it begins executing, the examination requires at least two frames must be on the deque: one for the method the thief is taking (`A` in the original example), and one for the method the victim will still be executing (`B` in the example). If there are enough frames, the thief removes the first frame from the stealing end of its victim's deque and initialize its own deque to contain that frame. Otherwise, it fails and moves on to a different victim.

Needless to say, this interaction requires a synchronization protocol to ensure that a thief and its victim don't simultaneously remove the same frame from the deque. One simple protocol is mutual exclusion, using Java's built-in `synchronized` keyword. Unfortunately, mutual exclusion is rather inefficient. In Chapter 5 I present a few more protocols and discuss their performance implications.

3.4 Closures

Migrating a method to a new worker is only half of the problem. If the migrated method returns a value, that method must be able to send its value back to its parent. Although the concept of frames is sufficient for keeping track of the local state of a single method, returning requires a pointer back to the parent frame as well. Similarly, the parent method needs to know whether or not it has outstanding children, so it knows how to behave at a `sync` statement. For methods which need this information, the JCilk runtime system creates a Closure object, which is described in this section. Although Closures were present in Cilk, they take a much more prominent place in the JCilk implementation. This section also shows how JCilk co-opts the inlet mechanism used by Cilk[1], using it to convert these returns from another worker from asynchronous events into synchronous events. Thread atomicity is also implemented at the closure level, by a protocol explained at the end of this section.

Returning

Every worker (except when it has run out of work) always has exactly one closure, representing the first frame at the stealing end of its deque. When a thief performs a steal, it always takes from the stealing end of its victim's deque, so it always takes its victim's closure C. To maintain the invariant of one closure per worker, it creates a

[1] Recall, however, that JCilk does not allow explicit inlets to be written by the user.

```
public void setInletReturn(int retEntry, Object retVal) {
  switch(retEntry) {
    case 1:
      x = ((Integer)retVal).intValue();
      break;
    case 2:
      y = ((Integer)retVal).intValue();
      break;
  }
}
```

Figure 3-5: The method in the `Frame` to execute inlets for the compiled `fib` method of Figures 3-2 and 3-3. Depending on which inlet is being executed, one of the frame's instance variables, `x` or `y`, will be updated.

new closure D to belong to the victim by "promoting" the frame at the new stealing end of the victim's deque.

After the steal, closure D has a ***parent pointer*** pointing back to C, to indicate that the method corresponding to D should return its result back to C when it finishes. In closure C, a corresponding entry is added to indicate that it has D as a child, along with the value of C's PC before the steal. This entry allows C to remember exactly where D's value should be returned back to, that is, which inlet to execute when D returns. It also acts as a "join counter," telling C's worker that it has an outstanding child in case C's method encounters a `sync` statement.

The inlet is tailored to the particular spawn statement so that the return result is always handled as the user specifies. This specialization is implemented by a general-purpose method such as the one in Figure 3-5, which dispatches to the correct inlet based on its arguments. For example, the original spawn statement `x = spawn fib(n-1)` creates an inlet whose body performs `frame.x = val`. Similarly, the statement `o.y += spawn f()` compiles into code to store `o` into `frame.lhs` before the method is called, and creates an inlet to perform `frame.lhs.y += val`. A part of the left-hand side of the original expression is stored in the frame before the child method is spawned to support idioms in which the left-hand side changes, for

State	Decription	Successors
1. READY	Not running, but can immediately begin/resume.	2
2. RUNNING	Currently running.	1, 3, 5
3. RETURNING	Preparing to pass its return value to its parent.	4
4. DONE	Has returned its value to its parent.	
5. SUSPENDED	Waiting on a child at a `sync` statement.	2, 6
6. INLETS	Running an inlet while waiting on a child.	5

Table 3.1: The states of a Closure.

example if `o` is an element repeatedly drawn from an `Iterator`.[2]

When the Confirm stage spawning a method `B` fails due to a theft, the return value from `B` is passed into the runtime system. Following the parent pointer, the runtime system can determine which worker is currently executing the stolen parent method `A`. It asynchronously notifies `A`'s worker by setting a flag that an inlet is available to run, and stores the return value in a public place.

To maintain thread atomicity (not to mention efficiency), each worker only checks for available return values when it encounters a thread boundary of its own. At that point (at the Confirm stage of a spawn, for example), it notices the waiting inlet and run it. Method `B`'s return value is handled as appropriate.

Implementing Thread Atomicity

Ensuring that thread atomicity is followed means making sure that no two workers ever try to operate on the same method simultaneously. Since only closures can be stolen, and only closures can have inlets run in them, controlling access to the closures is sufficient. My implementation uses a protocol based on the state field of the closure.

The six states available to a closure are shown in Table 3.1. When the first closure is created when the program begins, it is in state READY. A closure being stolen also its status set to READY. (The only other way a closure's status can be set back to READY is discussed in Chapter 4.) In general, a closure with status READY is one which is available to begin executing, but is not currently being executed on any

[2] This interpretation technically goes against the Java Language Specification [13], which requires that both the left-hand-side's location and its value be looked up before the right-hand-side is evaluated. To obey Java's semantics, the serial elision of `x += spawn f()` should be considered to be `x = f() + x`.

Worker.

Once a worker claims a closure, it sets its status to RUNNING and begins executing it. A closure spends the majority of its time in this state; only a RUNNING closure can be stolen. A RUNNING closure executes inlets only when its execution is at a thread boundary, to ensure that thread atomicity is maintained.

Eventually, the method represented by the closure finishes executing on some worker. At that point, the closure stores that return value and changes its status to RETURNING. Then it sends its return value to its parent, and its status finally becomes DONE.

An alternate possibility is that the closure executes a `sync` statement in its method while it has outstanding children on other workers. In this case, it cannot proceed further. Rather than blocking the worker, the closure's status is set to SUSPENDED and it is set aside. Its worker attempts to steal new work.

The SUSPENDED closure is now left with no worker executing it, but that's to be expected since it has no work to do itself. Its children, however, continue to execute on other workers. When a child D finishes while its parent C is SUSPENDED, something a little different happens. Simply passing the result to D won't accomplish anything, since C has no worker to poll its inlets. Instead, the worker which just finished D takes ownership of D in the INLETS state. It executes any waiting inlets (more may have appeared from other children simultaneously), and then checks again to see if there are still outstanding children. If there are, the closure is re-SUSPENDED. If not, then the closure stays on the same worker but becomes RUNNING again.

Chapter 4

Exception Implementation

On some level, an exception is just another kind of return value, and the JCilk runtime system treats it accordingly. Unlike a returned value, however, a thrown exception could be passed to one of many locations (`catch` clauses) depending on what kind of an exception it is. Returning to the correct location requires a more complex data structure called a "try tree," introduced in this chapter. This data structure, unique to JCilk, shadows the dynamic hierarchy of `cilk try` statements much like the Frame Deque shadows the Java call stack. The try tree is updated every time a `cilk try` block is entered or exited, as described in Section 4.1. It can then be used to determine which methods to abort and which catchlets to execute, a process detailed in Section 4.2. On the other side of the abort, a `CilkAbort` exception appears in the aborted methods, by the technique in Section 4.3.

4.1 Writing to the Try Tree

The try tree is a way of tracking every `cilk` block (cilk methods and `cilk try` blocks) containing the locus of control. Maintaining it is simple. The only time its state can change is when the locus of control enters or exits a try block, at which point the obvious update can be made. This section begins by explaining exactly how that happens. The try tree also stores values and exceptions passed back from the method's children; those updates require slightly more complex procedures, also

```
cilk void threeWay() throws ExceptionThree {
   spawn A();
   cilk try {
      spawn B();
      cilk try {
         spawn C();  //throws exception.
      } catch(ExceptionOne e) {
         cleanupC();
      }
   } catch(ExceptionTwo e) {
      cleanupB();
   }
   D();
   sync;
}
```

Figure 4-1: A method containing nested `cilk` blocks, each containing a `spawn` statement.

described in this section.

Example context

First, as an example of a complex context into which an exception might be thrown, consider the method in Figure 4-1. Depending on what kind of exception the call to `C()` throws, different sets of spawned methods might need to be aborted. If `C` throws a `RuntimeException`, for example, then `B` could be aborted (assuming it was still running), but `A` must continue normally. In order to determine which spawned child methods should be aborted, the worker must keep track of where in the parent method each was originally spawned from and, in particular, what the most directly containing `cilk` block of each is.

Maintaining the try tree

For efficiency, and because only a method running in a slow clone can have children running on other workers, the try tree is maintained only in the slow clone. Since the vast majority of the work is done deeper in the ready deque, maintaining the try tree does not add significant overhead. (See Chapter 5 for a more thorough discussion of the performance implications of exceptions.)

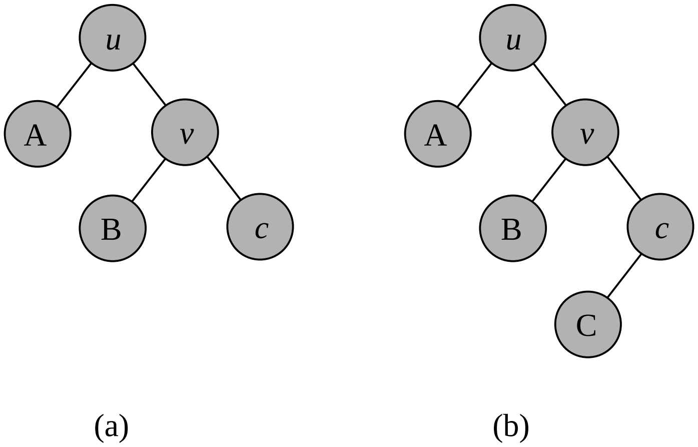

Figure 4-2: The try tree corresponding to an execution of `threeWay` (Figure 4-1). Node u represents the method itself, node v represents the outer `cilk try` block, and node c represents the inner one, where the cursor is. In (a), methods `A` and `B` are executing on different workers from `threeWay`. In (b), `threeWay` has been stolen again, creating a new node in the tree for `C`.

The try tree contains three different kinds of nodes, as shown in Figure 4-2. Internal nodes like u and v each represent a `cilk` block. A node's parent represents the `cilk` block most directly containing that node's block. Most leaves, like `A` and `B`, represent spawned calls that are currently executing on different workers; each leaf's parent node corresponds to the block from which the method was spawned. At most one leaf, here c, might also correspond to the ***cursor***, which tracks the `cilk` block containing the worker's current locus of control. In certain circumstances, a leaf might also be added in order to direct that a `CilkAbort` exception should be thrown at a certain point.

When a frame is promoted into a closure (as its parent is being stolen), a new try tree is created for that frame's method. Conceptually, the new tree consists of one branch from root to the cursor, corresponding to the method itself (the root) and every `cilk` try block containing the current locus of control (the other nodes). In practice, the tree is initialized to only contain a single node, where the cursor is. The later tree grows upwards to contain the other nodes.

Maintaining the try tree is straightforward. Whenever a slow clone enters a `cilk try` statement, the cursor moves down to a new node created as a child of the previous cursor node. Whenever a slow clone method leaves a `cilk try` statement normally, the cursor moves up a level. (If it leaves the `cilk try` as a result of a thrown exception, the cursor doesn't move, allowing the runtime system to track down the point where the exception was thrown.) Whenever the tree's closure is stolen, the try tree is updated to include a pointer to the new child closure as a child of the current cursor node. The tree thus adds structure to the list of children discussed in Section 3.4.

Confirming at an exception

Recall that whenever a method returns after it's been spawned, the first thing it does is confirm that its parent method was not stolen. The worker must make this confirmation after every spawn completes, even when the spawned method throws an exception. The confirm mechanism won't work as described previously, however, since if an exception has been thrown, the code immediately following the `spawn` statement won't be executed. Instead, every `spawn` statement must be wrapped in a `try-catch` statement, as in lines 5–11 of Figure 4-3.

This new confirmation calls a new method, `popFrameCheckException`, so the runtime system knows that the exception was thrown and not returned. If it turns out that the parent method has been stolen, then the worker treats the exception differently from a return value. The question is, what should the worker do with the exception? Where should it put it?

Returning into the try tree

When a child method returns a value, the first step it takes is to find the leaf in the try tree which corresponded to the child method which threw the exception. Once the leaf is found, the returned value is inserted into the return field of that leaf, along with a bit to tell whether the value was returned normally or thrown. Finally, as with the implementation in Chapter 3, the worker who currently owns the closure is flagged that it has a return value.

A returned value from a child is the most common way for values to get into the try tree. There are also four less-common situations when an exception might be

```
private int fib(Worker worker, int n, int returnEntry) {
   ⋮
   try {
     try {
        _tmp = this.fib(worker, n-1, 1);
     } catch(RuntimeException e) {
        if(worker.popFrameCheckException(e) == worker.STOLEN) {
           return;
        } else {
           throw e;
        }
     }
    ⋮
   } finally {
     worker.checkAbort();
   }
   if(worker.popFrameCheck(new Integer(x))) {
      return 0;
   }
   x = _tmp;
   ⋮
   return x + y;
}
```

Figure 4-3: The "fast clone" of the compiled Fibonacci method, with support for exceptions

added directly into the try tree, without being associated with any previous spawned method. After they are added, they are treated exactly as if a method had been spawned which threw that exception. The five times when a value or exception will be written into a try tree are:

1. A value is returned or an exception is thrown from a child on another worker.
2. An exception is thrown on the same worker as the closure and is uncaught.
3. An exception is thrown on the same worker as the closure and is caught, but the `cilk try` block still has outstanding children.
4. An exception is thrown from a catchlet or finallet.
5. A `CilkAbort` needs to be thrown due to an abort.

Case 2 occurs when an exception is thrown on the same worker owning the closure but is not caught at all, and it propagates all the way up through the slow clone and into the worker code which initially called the slow clone. When the worker receives the exception, it inserts it directly the try tree at the cursor (which did not move after the exception was thrown). Note that this exception might have been thrown either from inside the method itself, or from a child method executing on the same worker.

Case 3 occurs when the exception is caught at some `cilk try` statement, but there are still outstanding children spawned from inside that `cilk try` statement. In this case, the exception is similarly added at the cursor's location.

Cases 4 and 5 are discussed in more detail in Section 4.2.

Adding finallet numbers

The runtime system also needs to write to the try tree to support the execution of finallets. Unlike a catchlet, which takes an exception as an argument, a finallet takes no arguments. Thus, there is no need to write an additional value into the try tree node corresponding to the finallet's `cilk try` block. The runtime system does, however, still needs to determine whether a finallet needs to execute and, if so, which finallet.

Only minor additions are needed to support the finallet mechanism. The lookup methods which determine where an exception is caught must also be able to tell where a `finally` clause will be executed. More significantly, each try tree node has an additional field: its finallet number. Once all of that node's children are complete and the cursor has left the node, the runtime system examines the finallet number and, if that number has been set, executes the indicated finallet.

The finallet number must have been set at some point before the finallet executes. Notice that the locus of control must leave the `cilk try` block before that block's finallet executes. In Java, whenever a locus of control leaves a `try` block, the `finally` clause of that `try` block executes. Taken together, these imply that Java always executes a `finally` clause in the slow clone before the runtime system needs to execute its corresponding finallet. This fact makes the `finally` block the ideal place to set the finallet number. It also means that if there are no outstanding children from within the `cilk try` block, the slow clone can execute the original `finally` block directly instead of creating a finallet.

The slow-clone `finally` clause can also execute at other times besides when the locus of control leaves the `try` block. For example, when the slow clone discovers that it has been stolen, it immediately returns. Ordinarily, that return prevents all other execution in the slow clone, but it cannot prevent the `finally` block from executing. The same problem occurs when a `sync` statement fails. Before it sets the finallet number, the slow clone must check that no unusual cases have occurred, and the locus of control is genuinely leaving the `cilk try` block.

Handling an `Error` exception

The implementation I've described generally prevents the worker thread from ever catching any exceptions thrown in the user code. While this behavior is desired for typical cases (since a user program's deliberately thrown exception should not affect a worker), a thrown `Error` exception is handled differently. It does still propagate back up to parent methods, just as any exception would. Because it also describes a fatal condition which the worker itself needs to know about, however, the `Error` is also rethrown at the worker level. It then propagates all the way up through the worker thread, ultimately terminating the worker itself.

4.2 Reading from try tree

The information maintained in the try tree is put into use after an exception is thrown. If the worker finds any exceptions stored in the try tree, it uses the helper methods generated by the compiler to determine where in the tree the exception is caught. The worker then uses the tree to determine which children to abort. The tree also determines whether its method throws an exception to its own parent. This section describes these three ways in which the tree is used.

Polling the try tree

Once a return value has been added into a worker's try tree, it is that worker's responsibility to take the next step. The next time it checks its return flag (either in the Confirm stage of a `spawn` statement, or else at a `sync` statement), the worker searches its try tree looking for completed children. When it finds a returned value at a leaf, it immediately executes the inlet corresponding to that leaf, passing it in the returned value as an argument.

When it finds an exception, the worker's job is more complicated. Its first task is to determine where in the try tree that exception is caught. For this purpose, the compiler has produced (via static analysis) a lookup table like the one shown in Figure 4-4. It passes in the "return entry", the PC value specifying a particular thread boundary in the code (corresponding to the point from which the exception was thrown), and the exception which has occurred. Based on the type of the exception, the method returns how far up the try tree the exception should propagate to reach the node that catches it[1] A return value of `-1` means the exception is not caught; the try tree treats a value of `-1` as being caught at the root of the try tree, which

[1] For the purposes of the try tree, a `finally` block is treated as catching all exceptions. This special case is necessary because the body of a `finally` block could "intercept" an exception by throwing a new, different exception.

```
public int getCatchletAltitude(int retEntry, Exception retVal) {
   switch(retEntry) {
   case 1:
   case 6:
      return -1;

   case 2:
   case 5:
      if(retVal instanceof ExceptionTwo) {
         return 1;
      } else {
         return -1;
      }

   case 3:
   case 4:
      if(retVal instanceof ExceptionOne) {
         return 1;
      } else if(retVal instanceof ExceptionTwo) {
         return 2;
      } else {
         return -1;
      }
   }
   ⋮
}
```

Figure 4-4: The lookup table method which determines where an exception thrown in `threeWay` (see Figure 4-1) will be caught.

corresponds to the cilk method block itself.

Armed with this information, the runtime system climbs the specified number of steps up the try tree. At each node on its way, the runtime system attempts to "choose" the current exception as the unique exception that node will handle. The specification states that every `cilk` block can handle at most one exception; this process of choosing is a concrete representation of that idea. If the choosing fails because another exception has already been chosen for a given node, then the current exception is discarded and climbs any further up the tree. Once the runtime system has climbed the correct number of steps, we say it is at the "catching node." One more time, it attempts to choose the current exception. If it succeeds, then it moves on to aborting any outstanding children.

Aborting with the try tree

The children which should be aborted are those that were spawned from a statement contained in the `cilk` try block catching the exception[2]. These children exactly correspond to the ones whose nodes in the try tree are below the catching node. That is, every leaf in the subtree rooted at the catching node should be signaled to abort. For now, we assume that the cursor is not in that subtree.

The worker initiating the abort traces the tree to find these leaves. The closure pointed to by each leaf gets its abort flag set. The abort flag is set asynchronously from the perspective of the workers being aborted; Section 4.3 describes the other half of the two-phase process, similar to the way a value is returned to a parent, by which the abort is converted to a synchronous event.

The worker initiating the abort also traces the try trees of the children it aborts, looking for any outstanding children they might have. These grandchildren should also have their abort flags set since they, too, are dynamically contained in the try block catching the exception. The worker continues to recursively signal descendants to abort until it has signaled them all.

[2] We can ignore the "dynamically" part of "dynamically enclosed" because only one method is in question, so the static and dynamic states are identical.

It is important to ensure that this process of abortion eventually terminates. One could imagine a case where a method recursively spawns off children which are stolen faster than they can be aborted. The current implementation solves this problem by prohibiting a worker from stealing if its previous closure was aborted, but all of that closure's children have not yet been aborted. Thus aborting all the children of a single method requires signaling at most one closure per worker, putting an upper bound on the amount of work the abort requires. Since closures[3], not methods, are signaled to abort, the amount of work required to signal a method's children to abort is dependent only on the number of different workers those children are running on, and not the depth of the ready deques on those workers.

After the abort

Recall from Chapter 2 that the catch and finally clauses of a `cilk try` block only execute after all of the children spawned from that method have completed. In terms of the try tree, we know that a method has completed when the node representing that `cilk try` block has no children.

Nodes are removed from the try tree when they are no longer necessary. As the worker traces the try tree, whenever it handles a returned value stored in a leaf node, it removes that node from the tree to indicate that the closure it represents has completed. Similarly, whenever the cursor moves up from a node which has no children, that extraneous cursor node is deleted. This deletion maintains an invariant that every leaf in the try tree represents work currently being performed.

When the last child of some internal node u is removed, it is time for the `catch` and `finally` clauses for that node's `cilk try` block to run. If the last child was the cursor and u has not chosen an exception, then the `finally` clause executes following the normal Java control flow. Otherwise, special steps must be taken to run those clauses.

First, if u has chosen an exception to handle, a catchlet runs to handle that exception. The catchlet is implemented by a method (much like an inlet) which takes

[3] or, equivalently, workers

in the exception as a parameter, and whose body performs the work that would have been performed by the `catch` clause on u's `cilk try` block. After that, if there is a `finally` clause in u's `cilk try` block, then it is similarly run as a finallet.

The catchlet or finallet might itself throw an exception. It is, in fact, a common idiom for a `catch` clause to re-throw either the exception that it caught or a more generic related exception. These thrown exceptions are immediately added in to the try tree as a sibling of u. (This addition into the try tree is Case 4 in the list above.)

Note that all of the work to handle inlets, catchlets, and finallets takes place as the worker is polling the try tree. Only the worker owning a method can poll that method's try tree, and it only does it when that method is at a thread boundary[4]. The tree is traced serially, so at most one inlet, catchlet, or finallet is running at a time. Taken together, these behaviors guarantee ensure that thread atomicity holds.

Throwing an exception to the parent

How does an exception get passed to its parent in the first place? The answer is an extra check added into the work that a closure performs when it is in its RETURNING state. Instead of only looking at its stored return value, the closure also looks at whether the root of the try tree has chosen an exception. The root of the try tree represents the cilk method block itself, so any exception being "handled" by that block is really being passed up to the method's parent. The closure cooperates by designating that exception as the method's "return value."

4.3 Being aborted

The chapter up to this point has completely described one side of the abort implementation: the how the method that catches the exception begins the abort process. On the other side, the method being aborted must first receive a signal to abort. It then must throw a `CilkAbort` to enable the user code to do any cleanup necessary as

[4] In fact, it only does it when *every* method on that worker's deque is at a thread boundary, but that's less important.

the work is aborted. This section describes the entire process from the perspective of the closure that is being aborted, and in particular the way the abort begins.

Observing the abort signal

As I described in Chapter 2, aborts occur semi-synchronously from the perspective of the method being aborted. Thus, the check for the abort flag also occurs only at thread boundaries, primarily in the Confirm stage of `spawn` statements and at `sync` statements[5].

To be more precise, the compiled code makes a call into the runtime system (through the `popFrameCheck` method) at those particular points. The runtime system checks the abort flag. If it is set, then the runtime system immediately throws a new `CilkAbort` exception.

Aborting

Looking back at Figure 4-3, we see that the immediate effect of this exception is that the return value from the spawned method never gets stored into its destination `x`. Instead, the `CilkAbort` abruptly completes the block containing that store, skipping line 18. The `CilkAbort` propagates up the Java call stack from there, and the user's code can catch it on the way in order to perform any necessary cleanup.

Ideally, this propagation would be enough to ensure that every Cilk block on the signaled worker would receive the `CilkAbort`, as the specification requires. Unfortunately, the user code might catch the `CilkAbort` without re-throwing it. To work around this problem, the compiler adds its own `try-finally` block as a wrapper around every `cilk try` block. This new `finally` block asks the runtime system to throw a new `CilkAbort` if the method is being aborted. Since that call is in a `finally` block, it executes regardless of what the user's own `catch` or `finally` block might do when a `CilkAbort` is thrown. Code implementing this behavior can be seen in line 13 of Figure 4-3.

[5] In this case, it is important that these statements are thread boundaries for every method on the worker's deque, because all of those methods are being aborted from the bottom up. Contrast with the inlet case, where an inlet was running only in the top frame on the deque.

Because a `CilkAbort` is an exception, it is handled by JCilk exactly as any other exception. In particular, a node of the try tree can choose to handle a `CilkAbort` (and it often does). On the other hand, if a node has already chosen an exception, it does not choose the `CilkAbort`. The catching node which initiated the spawn in the first place always has already chosen an exception, so the `CilkAbort` exception cannot propagate up past the start of the work being aborted.

Even after a child method has been signaled to abort, it may still return an ordinary return value or a non-`CilkAbort` exception. These values are replaced with `CilkAbort` exceptions before they are stored into the try tree, to ensure that every Cilk block being aborted will receive a `CilkAbort` exception.

Spinning off work at an abort

There is still one more critical case to consider. When some method `A` catches an exception, it's possible that `A`'s primary locus of control might still be contained in the `cilk` block being aborted. If that block has spawned methods which are still executing on the same worker, then those methods should be aborted, just as they would be if they were running on other workers. Doing so causes a particular problem, however, when the exception is caught in `A` and only a portion of `A` needs to be aborted.

When the runtime system detects that the try tree node containing the cursor needs to abort, it "spins off" the first child method `B`, exactly as if the closure were being stolen. In this way, the runtime system creates a new closure for `B`, which is immediately signaled to abort. The worker then abandons `A`'s closure; from here on out, it acts exactly as if `A` had been stolen, was executing on another worker and has just signaled `B` to abort.

Moving the locus of control

Something must also be done with the method being spun off to ensure that it behaves correctly. Since the end of a `cilk try` block is not in any way, shape, or form like a `sync` statement, execution in `A` should be allowed to continue, starting from after the end of that `cilk try` block, without waiting for the `catch` clause to execute. Before it abandons that closure, its previous worker (now working on `B`) sets its status to READY to signify that work can, in fact, be done on `A`.

Of course, the default behavior for whichever worker next executes in `A` is to continue from where it was previously left off—in the middle of the `cilk try` block that caused all the trouble in the first place. This behavior is clearly incorrect. To rectify the situation, the `pc` field of `A`'s frame is advanced to a continuation point immediately after the end of the `cilk try` statement. The worker looks up where to place the `pc` in another lookup table produced by the compiler, which takes the previous `pc` and an exception and returns the new `pc`. The cursor is similarly moved up to the level above the catching node in the try tree.

In some rare cases, when the locus of control is at a `sync` statement and there are no outstanding children, a `CilkAbort` is added into the tree at the cursor's old position. Adding a `CilkAbort` in those places (as Case 5) ensures that the `cilk try` blocks containing that `sync` statement get the chance to catch a `CilkAbort` even though there are no children to throw one.

Chapter 5

Performance

In this chapter, I discuss the performance of compiled JCilk programs. The designers of Cilk-5 were able to show that the overhead of adding `spawn` statements to C cost a factor of 2 to 6 over using function calls [11]. The performance of JCilk is unfortunately not quite that good, due to a number of factors in the Java Virtual Machine (JVM) which make good parallel performance difficult. In particular, it turns out that object allocation can be grossly inefficient on a parallel machine. Additionally, there is no efficient way to perform the necessary synchronization between workers to allow correct performance of a steal. In this chapter I begin by discussing the work-first principle, which enabled me to find those sources of overhead and, then I move on to examine the particular performance problems I have found. I conclude by proposing a few alternate implementations which attempt to get around these problems, by using more efficient synchronization or by completely avoiding synchronization in the common case.

5.1 Measurements

Two primary metrics can be used to rate the performance of JCilk programs. The first is the overhead: how much slower does a JCilk program run than its serial elision? The second is the speedup: how well does the JCilk program take advantage of the processors it is allocated? I use two benchmark programs, Fib and All-Queens, to obtain the performance measurements described in this section.

The first program is Fib, a program to recursively compute Fibonacci numbers, using the method given in Figure 3-1. This benchmark gives the worst-case bound on the overhead. Since its only computation is to perform spawn after spawn, `Fib` reveals the true overhead of a `spawn` statement.

The other program is All-Queens, a modified n-Queens program which searches for every possible configuration instead of only one. (The original program would have been too nondeterministic to give reliable timings, since its runtime would depend on the random choices of the scheduler.) This program is a more realistic benchmark, since it performs significant computation and memory allocation of its own.

The measured running times of those programs (using three different JVMs and both the original synchronized ready deque and the atomic-variable ready deque described in Section 5.4) are given in Table 5.1. Notice that in general, the overhead of spawning overwhelms any speedup on Fib, but All-Queens achieves significant speedup as it runs on more processors. It appears from the All-Queens data that the Sun Java 1.5 JVM is the most well suited to running JCilk programs.

From these results, we can derive the relative speedup obtained by JCilk. Those results are given in Table 5.2. Notice, in particular, that the All-Queens program obtains approximately linear speedup: the running time on P processors scales proportionally to $1/P$. This performance is the upper bound on the possible speedup, so matching it is a significant accomplishment. The performance of Fibs does not reach linear speedup, but it too has some noticable speedul.

5.2 Sources of Overhead

The philosophy behind the design of both the JCilk scheduler and the original Cilk scheduler is the work-first principle. This idea is discussed more thoroughly in [11] and summarized in this section, shows that the only overhead to worry about is what is added to the fast clone. In this section I also show that the overhead in the fast clone boils down to two major components: the cost to allocate a new frame and the

JVM implementation	Sun 1.4	IBM 1.4	Sun 1.5	Sun 1.5, atomics
Fib in Java	3.2	2.1	1.9	1.9
Fib in JCilk, 1 worker	88.0	89.8	71.2	58.9
Fib in JCilk, 2 workers	165.2	53.1	57.7	37.8
Fib in JCilk, 3 workers	153.0	41.3	54.6	41.1
Fib in JCilk, 4 workers	154.5	39.7	64.2	29.0
All-Queens in Java	100.7	121.8	60.3	60.3
All-Queens in JCilk, 1 worker	109.1	121.1	76.4	69.6
All-Queens in JCilk, 2 workers	174.4	70.8	39.3	34.9
All-Queens in JCilk, 3 workers	172.3	54.7	24.6	24.2
All-Queens in JCilk, 4 workers	176.2	53.5	18.5	19.0

Table 5.1: The running times of two serial Java programs and the JCilk equivalents of those programs on various numbers of workers. Fib was run with $n = 40$, and All-Queens was run with $n = 14$. The first three columns represent the same Java bytecode running under three different JVM implementations: those distributed with Sun's 1.4 and 1.5 JDKs, and that distributed with IBM's 1.4 SDK. The fourth column also uses the Sun 1.5 JVM but with atomic variables, rather than synchronized blocks, used to implement the ready deque. All times are clock time, measured in seconds. Averages were taken for a few cases with varying run times.

	T_S/T_1	T_1/T_2	T_1/T_3	T_1/T_4
Fib	0.032	1.56	1.43	2.03
Queens	0.867	1.99	2.88	3.66

Table 5.2: The Speedup of JCilk programs, using the Sun Java 1.5 JVM and atomic variables. T_S is the time taken by serial Java program, and T_n is the time taken by the JCilk program on n workers. The first column shows the efficiency of the JCilk program, and the others show the speedup obtained on multiple processors.

Modification	Add'l time	Cum. Time
Serial Fibonacci	1.01	1.01
Runtime startup overhead	0.11	1.12
Frame allocation	25.7	26.8
Frame deque usage	13.4	40.2

Table 5.3: Performance running `Fib` with one worker on a multiprocessor machine. All times are in seconds.

cost to synchronize with the ready deque.

The idea for the work-first principle comes from two measures of the length of time a program takes to execute. One measure, the ***work*** T_1 of the computation, describes how much total computation must be done. The work can also be interpreted as the time the program takes to run on a single processor. Ideally, a program running on P processors takes time T_1/P to execute. The other metric is the ***critical path length*** T_∞, which describes the longest chain of dependencies between dependencies in the computation. No matter how many processors the computation runs on, it never finishes in time less than T_∞.

The work-first principle states that the performance of a JCilk program is determined only by its work, and not by its critical path length. Whenever it's possible to lower the total work by lengthening the critical path, the JCilk design makes that choice. In terms of the runtime system, this principle means it is best to add overhead to a steal in exchange for lowering overhead in the normal case, since we can show that every steal corresponds to making progress on the critical path.

Based on this argument, we can determine where to look for overhead and what to try to optimize to improve the overall performance of the system. Any overhead added to the stealing process, or to the initialization of the runtime system itself, can be (assuming it is not inordinately large). Even overhead which is added into the slow clone can generally be lumped in with the steal that caused the slow clone to be called, and thus it can be ignored. Only the fast clone is a truly significant source of overhead.

The fast clone contains only two significant sources of overhead, which are shown in Table 5.3. The first, as might be predicted, is the cost to create a new frame, which is incurred once per spawn. The second is the cost of accessing the frame deque, both to push a new frame on the stack, and to check for a steal and pop the frame off of the stack if necessary. There is also a small amount of overhead to checking for aborts at the end of `cilk try` blocks, although that overhead is insignificant compared to the costs of the `spawn` statements in those blocks.

5.3 Difficulties with Java

In several ways, Java shows itself to be less-well suited to a JCilk-type runtime system than C was. The implementations of the memory allocator in the JVM behave poorly in a parallel context. The memory model is also inflexible and depends on frequent high-cost synchronizations. This section discusses these problems.

Memory Allocation

The parallelism of a JCilk program is squandered if it is run only on a single processor. Only a performance metric from a multiprocessor machine are significant. As a first example, I describe the general process I used to detect these (and other) inefficiencies in the JCilk runtime system.

Unfortunately, the JVM which Sun distributed with its Java 1.4 JDK functions poorly on multiprocessor kernels. It's not a question of lock contention; even with only a single thread running, the performance discrepancy between a two-processor machine and a single-processor machine was astonishingly large. Timings on one particular machine, as different parts of the runtime system were included, are given in Table 5.3. The numbers vary among machines and JVMs, but the general pattern of unacceptable overhead holds.

I was able to informally trace much of the performance hit to the allocator. Examination of the output from the Sun's JVM profiler revealed that the major delay occurs in the statements which were allocating new objects. A little further investigation, by taking snapshot stack traces, gave even more information. In each trace, it was likely that the thread was waiting to synchronize on a `Reference$Lock` object.

Apparently, allocating an object on a multiprocessor system goes through a common memory pool, for which a lock is required.

The Java Memory Model

The Java Memory Model was completely revamped for the Java 5.0 release in September of 2004. The previous memory model was so riddled with holes and unexpected outcomes that the specification was often not even implemented [31], so my work is based on the assumptions of the new memory model, as described in [24, 30].

In general terms, the memory model specifies that "correctly synchronized code" functions according to sequential consistency. Unfortunately, ensuring that code is correctly synchronized is expensive. It requires `synchronized` blocks and/or `volatile` variables, both of which can be major sources of overhead, even without contention. When code is not correctly synchronized, only minimal guarantees are made about its behavior, and those guarantees are not sufficient for the JCilk scheduler's purposes.

All of this synchronization work is necessary because every Java `Thread` is permitted to keep its own cached copies of every variable. That's why incorrectly synchronized code can behave unpredictably. Even on a single-processor machine, Java makes no guarantee that a `Thread` sees the most recent value of a variable if the program is not properly synchronized.

A `synchronized` block takes an object as an argument and performs mutual exclusion using a "monitor" associated with that object. Using a `synchronized` block is the most common way to synchronize a Java program. Leaving the `synchronized` block forces local data to be flushed to main memory. This flushing is the basis of the Java Memory Model's guarantee that when blocks on two different threads synchronize on the same object, every write which occurs before the first thread's block ends must be seen by every read after the second thread enters the block. A read or write on the same `volatile` variable on two threads behaves similarly [30].

Memory barriers

Java's memory model eliminates the usefulness of the THE protocol used in Cilk-5 and described in [11]. That protocol took advantage of the cheapness of a memory barrier to avoid having to obtain a lock in the common case. In Java, locks and synchronization are inseparable. The bond between the two ideas is especially problematic because synchronization is only useful when it is performed by both sides of a protocol. There is no way for one thread to ensure it has the most recent values from another thread, unless that thread has also performed synchronization. Thus guaranteeing that a thief sees the most recent value of a frame requires both the pop *and the push* of that frame to be synchronized, even when the push occurs in a fast clone.

5.4 Alternative Implementations

One solution to Java's faults is simply to improve on the built-in Java mechanisms. In Cilk development, a lot of work went into improving on the C libraries for these same two problems, memory allocation (for example, the Hoard Memory Allocator of [5]) and synchronization costs (for example the THE method of [11]).

In this section, I present a simpler path: two alternatives which allow the synchronization in the ready deque to be pushed out of the work and into the critical path. First I consider atomic variables, a new feature in Java 5.0 designed to allow efficient access to hardware synchronization primitives. Then I consider an acknowledgment-based stealing protocol, in which synchronization is not necessary at all but a worker may block for arbitrarily long while it attempts to steal.

Atomic variables

Starting in Java 5.0, the new `java.util.concurrent.atomic` package contains several classes which represent atomic variables. These classes support the standard non-blocking compare-and-set operations, using hardware support for those operations rather than relying on Java's memory model. Using atomic variables is one possible back door out of Java's linking mutual-exclusion to synchronization.

Atomic variables don't go very far, though. For example, imagine that the ready deque is being implemented as an array. Atomic counters can point to the head and the tail of the deque, and the array itself can be implemented as an atomic array, all of whose references are guaranteed to be atomic. This ensures that an access to the deque always returns a reference to the correct frame. There is still the problem that the array elements themselves aren't atomic; writes that occur after the frame is added to the deque may not be propagated to other workers. Without adding the overhead of atomic variables to every frame object, there is still no guarantee that once the runtime system gets the right frame, it can read the correct values.

The solution is to treat frames as immutable until they are stolen for the first time. One feature of the memory model is that atomic variables act as synchronization points: every read after a read of an atomic variable must see all writes that occurred before the previous write of the same variable. Given this, rather than creating a frame at the beginning of the fast clone and then modifying it in the Save stage of each spawn, why not just push a new frame in each Save stage? Exactly one frame is still pushed on the deque at each spawn, so this new frame protocol adds no extra memory overhead.

Of course, the slow clone must still be able to update the variables in the same frame as it started with, so that inlets (for example) have a single consistent frame to return back to. The slow clone is on the critical path, though, so it can do as much synchronization as it wants; we aren't concerned with its overhead. A potential thief can then perform a synchronization protocol to ensure that it will correctly steal either an original frame or a modified one.

The results of implementing this idea can be seen in the last column of Table 5.1. A deque based on atomic variables eliminates much of the synchronization overhead of Fib, revealing a significant speedup. On All-Queens, on the other hand, the atomic variables appear to introduce their own slight overhead. It remains to be seen whether the atomic variables do more harm or good to performance on a wider range of programs.

Acknowledgment-based stealing

Another alternative stealing protocol is based on requiring acknowledgment from the victim before the thief can proceed. In the original stealing protocol, every worker must synchronize all of its writes to all of its frames at every spawn, in case those frames are stolen. Eliminating the possibility of a thief sneaking in unannounced, allows the runtime system to also eliminate the synchronization from the fast clone.

In the acknowledgment protocol, rather than a thief accessing its victim's deque directly, it merely notifies its victim that it would like to steal from it. The thief then goes to sleep while it waits for a response. At some point, the victim notices the thief's notification. So that this protocol doesn't add more work into the fast clone, the victim only checks for a steal notification when it is already checking all of its other flags, i.e. at thread boundaries. Once it sees a notification, the victim examines its own deque to see if it has a frame available to be stolen. If it does, it synchronizes on that frame and releases it to the thief. Then it wakes the worker up.

If there was a frame to steal, the reawakened thief also synchronizes on that frame, ensuring that it will see the most recent values written by its previous owner. From here, it proceeds exactly as described in Chapter 3: it begins working if it successfully stole a frame, and re-attempts a steal from some other worker otherwise.

The advantage of this protocol is that it eliminates all deque synchronization from the fast clone. Every synchronization that is actually performed can be associated with a steal, so all of the synchronization effort can be lumped into the critical path. This is a big gain.

The downside is that, since a worker only checks for would-be thieves at its thread boundaries, a thief could have an unbounded wait an before it receives an acknowledgment from its victim. Depending on the program, this trade-off may or may not be acceptable. In `Fib`, for example, practically every statement is a spawn—the savings are huge, and the downside is minimal. In a program with more calculation and fewer spawns, on the other hand, this could result in processors being wasted waiting.

Chapter 6

Related Work

In general, other parallel languages tend not to treat exceptions with the same importance that JCilk does. Many, in fact, do not consider them at all. Of the ones which do consider exceptions, all that I am aware of are based on a message-passing structure, and because they follow a different style of parallelism, they focus on different aspects of exception-handling. The cooperation model [19], for example, allows an exception to propagate from one processor to any other processor which attempts to communicate with it. The model used in DOOCE [35] is more similar to JCilk, and even presents the option to abort its equivalent of side computations, but does so in a very different environment from JCilk. Both of these languages also include extensive support for multiple simultaneous exceptions. Neither, however, can be viewed as a faithful extension of the semantics of a serial exception mechanism, as JCilk is. In this chapter I discuss first languages which do not address exceptions, and then present the cooperation model and DOOCE.

Exception-oblivious parallel languages[1]

Most parallel languages do not provide an exception-handling mechanism. For example, none of the parallel functional languages VAL [1], SISAL [12], Id [28], parallel Haskell [3, 27], MultiLisp [15], and NESL [6] and none of the parallel imperative lan-

[1] This section is a joint work with Angelina Lee and Charles E. Leiserson.

guages Fortran 90 [2], High Performance Fortran [33] [26], Declarative Ada [36, 37], C* [16], Dataparallel C [17], Split-C [8], and Cilk [34] contain exception-handling mechanisms. The reason for this omission is simple: these languages were derived from serial languages that lacked such linguistics.[2]

Other parallel languages do provide exception support because they are built upon languages that support exception handling under serial semantics. These languages include Mentat [14], which is based on C++; OpenMP [29], which provides a set of compiler directives and library functions compatible with C++; and Java Fork/Join Framework [21], which supports divide-and-conquer programming in Java. Although these languages inherit an exception-handling mechanism, their designs do not address exception-handling in a concurrent context.

The cooperation model

The cooperation model in [19] also gives a way to handle exceptions in a language which supports message-passing between threads. It is based on the principle of global exceptions. When a process Q terminates exceptionally, it makes its status available to other processes by publicly signaling an exception E. If another process P later attempts to communicate with Q, then the communication causes exception E to be thrown on process P. That exception can then be caught and handled by P in the same way that an ordinary serial exception could be.

Unlike JCilk's model, the cooperation model fully supports multiple exceptions being thrown simultaneously. A single operation in the cooperation model might consist of several processes executing in parallel. If one process Q from the operation tries to communicate with multiple others and discovers that they have terminated exceptionally, then multiple exceptions are simultaneously thrown on Q. To handle this case, the operation provides a ***resolving function*** which accepts a list of exceptions as parameters and returns a single "concerted" exception representing all of

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

the original failures. Allowing the program to consider all of the exceptions helps it understand what the source of the original failures may have been, especially if all those failures are merely symptoms of some greater common failure.

A similar resolution mechanism could ultimately be included in JCilk, should it prove to be worthwhile. Even after a Cilk block has chosen an exception to handle, it could continue to accept exceptions while it waits for its children to return. The major complication would be the interaction with `AbortExceptions`; it would be difficult to even decide when a method being aborted should throw its own exception and when it should throw a `CilkAbort`. Ultimately, I suspect that the smaller window that JCilk gives for "simultaneous" exceptions (between when a method catches the first exception when its children have aborted) would make concerted exceptions less useful.

DOOCE

In [35], an exception-handling framework is introduced in the context of DOOCE, a distributed object-oriented computing environment. It uses C++ style syntax to create a "flat object space" in which objects belonging to different address spaces (for example, on different computers) can pass messages amongst themselves. These messages take the form of method calls. DOOCE also adapts Java's syntax for exception handling, including both `catch` clauses and `finally` clauses.

When one a DOOCE method call returns an exception, that exception is passed to the calling method to be handled. At this point, there are two possible outcomes depending on type of the `try` statement catching the exception. The `catch` clause might wait to execute until all of the method calls from the `try` statement have completed by the methods' objects. In the meantime, other methods might also throw exceptions. DOOCE, like the cooperation model, lets multiple "simultaneous" exceptions be handled together. The catch clause can also send a "notification message" to each of those objects and proceed with the `catch` clause without waiting. JCilk uses a combination of these protocols, both aborting children and waiting for them to complete.

DOOCE's semantics also include an interesting but orthogonal feature: a resumption model of exception-handling as an alternative to the termination model used by C++ and Java. When exceptions occur and are handled by a catch clause, control still jumps directly to the catch clause. After that, though, the catch clause has the option to indicate that the program should either resume execution from the throw point, or that it should retry execution from the beginning of the `try` block.

Chapter 7

Future Directions

Although the JCilk-1 implementation has been completed, the JCilk project is still very much in progress. In this chapter, I give some ideas for future directions that JCilk might take. In particular, it could be valuable to open up an interface to allow a Java program, compiled using an ordinary Java compiler, to access the JCilk runtime system. I explain the need for such an interface in Section 7.1. The interface would create an opportunity to experiment with adaptive parallelism between multiple jobs, which I discuss in Section 7.2. Also, extending JCilk to include transactions could be one way for JCilk programs to get around the synchronization difficulties discussed in Chapter 5.

7.1 Connecting Java and JCilk

The JCilk language is a (largely) faithful extension of Java, and a JCilk program compiles into Java bytecode, so it would make sense for a single program to contain both JCilk and Java code. That's simpler said than done, however. The implementation described in Chapter 3 requires the runtime calls to be compiled into the method being executed, which is impossible for a method that has not been compiled using the JCilk compiler. Some new threading interfaces in Java 5 give a different way to think about Java threads which is much closer to the JCilk model, presenting an opportunity to embed the JCilk runtime system into the Java language. In this section

I explain the difficulties with a straightforward merge of Java and JCilk code, outline the new features of Java 5, and then give my ideas for taking advantage of them to bring Java and JCilk together.

Calling JCilk from Java

One fundamental limitation of JCilk is the inability for non-cilk methods to call cilk methods. (Recall that a cilk method is one which can be spawned and can spawn other cilk methods.) The original Cilk language had the same limitation, but in JCilk it is more severe due to inheritance and polymorphism. These Java language features make it possible for a non-cilk method to unknowingly call a cilk method, without the compiler having any way to statically detect the problem. In fact, the non-cilk method may not have even been passed through the JCilk compiler when it was compiled, and yet could still attempt to call a cilk method.

For example, a programmer may wish to extend the `java.util.Vector` class by overriding its `indexOf(Object elem)` method (which returns the index of the first appearance of the element in the Vector) with a parallelized version written in JCilk. An instance of the new `ParallelVector` class could be passed as an argument to any method taking a `Vector` argument, including those which call the `indexOf(Object)` method of that Vector. Depending on how cilk methods are defined in JCilk, this call could result in (among other possibilities) the original `Vector` method being called, the parallel method being called but executed serially, or the parallel method being executed in parallel. Similar dilemmas could arise if non-cilk methods can override cilk methods.

The current version of the runtime system does not allow a non-cilk method to call a cilk method at all, and does not allow cilk methods to override non-cilk methods. These decisions are necessary because a non-cilk method cannot be migrated. In order for a new worker to continue a method, it must be able to jump into the middle of that method and access the most recent version of that method's local variables. Suppose, for example, that a cilk method `A` calls a non-cilk method `B`, which calls a cilk method `C`. The first difficulty can be worked around because a non-cilk method cannot contain any JCilk thread boundaries, so there would be no reason to ever continue directly into B. The method B could, for example, be "partnered up" with either A or C, and stolen at the same time as its partner cilk method is stolen.

The difficulty accessing local variables, however, presents more complex issues. Even if B is never directly stolen, whichever worker indirectly steals it (for example by stealing A or C) has to eventually execute it. Since B was not compiled as a cilk method, though, its local variables exist only in one place: on the Java call stack of the worker on which it was originally called. Without access to those variables, another worker cannot execute B.

Various schemes have been proposed which allow cilk methods to be stolen, even if they call or were called by non-cilk methods, while leaving the interstitial non-cilk methods like B fixed in place on their original workers. Such ideas might eventually bear fruit, but the extra complexity they introduce (both into the runtime system and the analysis of the efficiency of the work-stealing algorithm) have so far proven daunting.

Java 5 threads

An alternative Java-JCilk interface is suggested presented by the Java 5.0 release, which (among many other changes to the language) offers a number of improvements (described in [25]) to Java's original threading mechanism. Although the new features are ultimately based on `Thread` objects, they go in many of the same directions as JCilk does to provide more power to the programmer.

The most significant new feature is the `Executor` interface, which provides a mechanism to decouple the scheduling from execution. Rather than explicitly creating a `Thread` object to execute a method, the programmer can pass that method into an `Executor` which handles all of the scheduling itself. The Java API provides several implementations of `Executor`, which use various strategies to schedule the methods they are asked to execute.

The new Java platform also introduces the `Callable` interface. Like the `Runnable` interface, it encapsulates a method which can be run at a later time (and on a different thread), but unlike that earlier interface, Callable allows its encapsulated method to return a value or throw an exception. When a Callable is submitted to an Executor, the Executor returns a Future object. The get() method of that Future object waits for the Callable to complete, and then returns the value that the Callable's method returned. If the Callable's method threw an exception, then Future.get() throws a ExecutionException containing the original exception as its cause. (The Future object also provides a non-blocking isDone() method to check whether the Callable is already done.)

The Executor-Callable protocol is very similar to the spawning protocol provided in JCilk. Although it (like everything else in Java) uses an object-based system rather than JCilk's linguistic system, it provides basically the same functionality, with only a slightly more complex interface presented to the programmer. One serious flaw, however, is that the Java Executor lacks the scheduling features which make JCilk so valuable for recursive programs. In particular, there is no way to track a parent's dependencies on its "child" Callables. After a new Callable object is added into the scheduler's queue, the rest rest of the calling method proceeds to be executed. The Callable method immediately execute on another thread if one is available, or it might not. When the calling method reaches the get() statement, it blocks until the Callable method is complete.

Depending on the Executor implementation, the Callable may never complete. For example, in the extreme case of recursion on a single-thread Executor, the single blocking method prevents the execution of all other Callables, including the one that it is blocking on. This problem can be solved by allowing the Executor's thread pool to grow arbitrarily large, but that introduces the significant overhead of allowing an arbitrary number of threads, while still giving none of the scheduling guarantees that JCilk provides.

A JCilk Executor?

Some of the difficulties with the Java 5 threading mechanism could be solved by implementing a better scheduler in the Executor than the ones bundled with the language. And what better scheduler is there than the JCilk scheduler?

Suppose that the JCilk runtime system, instead of starting up only when a JCilk program starts, could also be invoked directly through a hypothetical `JCilkExecutor` interface. When the `Callable` method passed to the `JCilkExecutor` is an ordinary (non-cilk) Java method, it should act as an ordinary `Executor` and execute the method. When the method is a cilk method, compiled with the JCilk compiler, then things get more interesting. The cilk method could be invoked as if it were called directly from another cilk method, by placing it at the top of a new ready deque, available to be executed or stolen. This solution avoids the difficulties of the other JCilk/Java merger schemes since it never allows a non-cilk method to sit between two cilk methods on the deque.

The idea of a `JCilkExecutor` has not yet been fully investigated, let alone implemented, but is a promising avenue for future work.

7.2 Future Integration

Besides the JCilk features I have already discussed, several other projects from the Supercomputing Technologies Group in MIT CSAIL also present opportunities for integration into JCilk. In this section, I describe two of them: Dynamic Processor Allocation, which seeks to find an efficient way to schedule multiple independent jobs on a given number of processors, and Transactional Memory, which gives all memory accesses the same transactional semantics traditionally associated with databases.

Adaptive Parallelism

One weakness of both Cilk and the current JCilk implementation is that although the scheduler will distribute work optimally between its workers, it can only operate with a fixed number of workers. This limitation is acceptable with only one program running, since that one program can simply be alloted all of the computer's processors. When multiple independent jobs run concurrently, however, the parallelism of those jobs often changes as they run. It is thus better to adaptively re-allot processors among the jobs as their execution proceeds. What is a good algorithm for doing these allotments? How is a “good algorithm” even defined?

Dynamic processor allocation is an area of research which focuses on answering these questions. The `JCilkExecutor`, if implemented, would allow multiple JCilk jobs to run simultaneously, providing a platform for testing implementations of some possible answers.

Transactional memory

Transactional memory is a more orthogonal application of some of JCilk's ideas. The basic idea behind transactional memory is that accessing memory in parallel ought to be easy. Rather than using heavyweight locks or intricate non-blocking mechanisms, all memory accesses should be able to use the transactional model traditionally used by databases [4, 10, 18].

One fundamental feature of transactions is that they might fail. If two concurrent transactions conflict (for example, if they try to write to the same location in memory), then at least one of those transactions needs to be rolled back. The transactional memory system provides built-in support for rolling back actions which were writes to memory. For other kinds of actions, however, it falls flat. If file I/O has occurred, for example, there is no easy way to roll it back. Dealing with these irrevocable actions is one problem confronting the current work on XJava, a transactional language which extends Java.

The exception-based abort mechanism introduced in JCilk provides one possible solution. In particular, one can imagine a `TransactionAbort` exception being thrown in a method to signal that it is being aborted but will be retried. This exception would give the program a chance to clean up in any way possible without relying on the language to know how to roll back every kind of action. If JCilk and transactional memory were implemented together as one combined language (XJCilk? JCilX?), it might even be possible to use one unified `Abort` exception to handle all reasons that a method might need to abort.

Chapter 8

Conclusion

JCilk provides a simple and efficient way to write parallel programs in a language that faithfully extends C to include the parallel semantics of Cilk. Mixing the features of these two languages introduces interesting difficulties, but at the same time provides novel, elegant, and powerful mechanisms to solve old problems. The concepts of exceptions and exception-handling, omnipresent in modern computing, are particularly challenging and rewarding to confront in a parallel context.

The JCilk language specifies a semantically consistent model for handling exceptions in either a serial or a concurrent context. In this thesis, I have described that model. I have also implemented the full JCilk-1 Runtime System, including all of its work-stealing and its exception-handling features.

There's still a lot of work left to do. Although the performance of some JCilk programs is acceptable, the JCilk runtime system introduces too much overhead. Until the efficiency can be improved, however elegant the exception mechanism might be, the language itself will not be useful. That will probably mean doing some serious work on the JVM, although I hold out some hope that a more efficient synchronization technique (or a pre-existing JVM implementation) may yet solve JCilk's problems.

Still, overall, JCilk-1 is a promising first look at a modern parallel language that is easy to use, powerful, and provably efficient.

Bibliography

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

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

[3] Aditya, Shail, Arvind, Augustsson, Lennart, Maessen, Jan-Willem, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In Paul Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, pages 35–49, June 1995.

[4] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transational memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, New York, NY, USA, 2000. ACM Press.

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

[7] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng.*, 26(9):820–836, 2000.

[8] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

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

[10] Keir Fraser. Practical lock-freedom. Technical Report 579, University of Cambridge, February 2004.

[11] Matteo Frigo, Charles E. Leiserson, and Keith 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.

[12] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim BHm, Walid Najjar, and Patrick 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.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Massachusetts, 2000.

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

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

[16] Philip J. Hatcher, Anthony J. Lapadula, Robert R. Jones, Michael J. Quinn, and Ray 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.

[17] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Ray J. Anderson, and Robert R. Jones. Dataparallel 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.

[18] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 92–101, Boston, Massachusetts, July 2003.

[19] Valèrie 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.

[20] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988.

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

[22] Angelina Lee. Personal communication, May 2005.

[23] Tim Lindholm and Framk Yellin. *The Java Virtual Machine Specification Second Edition*. Addison-Wesley, Boston, Massachusetts, 2000.

[24] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[25] Brett McLaughlin and David Flanagan. *Java 1.5 Tiger: A Developer's Notebook*. O'Reilly Media, Inc, 2004.

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

[27] Rishiyur S. Nikhil, Arvind, James E. Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem 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.

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

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

[30] Java Community Process. JSR 133: Java memory model and thread specification revision. `http://www.jcp.org/en/jsr/detail?id=133` (Viewed May 2005), September 2004.

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

[32] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

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

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

[35] S. Tazuneki and Takaichi 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.

[36] John Thornley. *The Programming Language Declarative Ada Reference Manual*. Computer Science Department, California Institute of Technology, April 1993.

[37] John 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.