Programming with Exceptions in JCilk

John S. Danaher I-Ting Angelina Lee Charles E. Leiserson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

November 10, 2005

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. This paper shows how JCilk integrates exception handling with multithreading by defining semantics consistent with the existing semantics of Java's `try` and `catch` constructs, but which handle concurrency in spawned methods.

JCilk's strategy of integrating multithreading with Java's exception semantics yields some surprising semantic synergies. In particular, JCilk extends Java's exception semantics to allow exceptions to be passed from a spawned method to its parent in a natural way that obviates the need for Cilk's `inlet` and `abort` constructs. This extension is "faithful" in that it obeys Java's ordinary serial semantics when executed on a single processor. When executed in parallel, however, an exception thrown by a JCilk computation signals its sibling computations to abort, which yields a clean semantics in which only a single exception from the enclosing `try` block is handled. The decision to implicitly abort side computations opens a Pandora's box of subsidiary linguistic problems to be resolved, however. For instance, aborting might cause a computation to be interrupted asynchronously, causing havoc in programmer understanding of code behavior. To minimize the complexity of reasoning about aborts, JCilk signals them "semisynchronously" so that abort signals do not interrupt ordinary serial code. In addition, JCilk propagates an abort signal throughout a subcomputation naturally with a built-in `CilkAbort` exception, thereby allowing programmers to handle clean-up by simply catching the `CilkAbort` exception.

The semantics of JCilk allow programs with speculative computations to be programmed easily. Speculation is essential for parallelizing programs such as branch-and-bound or heuristic search. We show how JCilk's linguistic mechanisms can be used to program a solution to the "queens" problem and an implemention of a parallel alpha-beta search.

This research was supported in part by the Singapore-MIT Alliance and by NSF Grants ACI-0324974. I-Ting Angelina Lee was supported in part by a Sun Microsystems Fellowship.

1 Introduction

JCilk is a Java-based multithreaded language for parallel programming that extends the semantics of Java [13] by introducing "Cilk-like" [10, 37] linguistic constructs for parallel control. JCilk supplies Java with the ability for procedures to be executed in parallel and return results, much as Cilk provides call-return semantics for multithreading in a C language [20] context. These facilities are not available in Java's threading model [13, Ch. 11] or in the Posix pthread specification [18] for C threading libraries. When embedding new linguistic primitives into an existing language, however, one must ensure that the new constructs interact nicely with existing constructs. Java's exception mechanism turns out to be the language feature most directly impacted by the new Cilk-like primitives, but surprisingly, the interaction is synergistic, not antagonistic.

The philosophy behind our JCilk extension to Java follows that of the Cilk extension to C: the multithreaded language should be a true semantic parallel extension of the base language. JCilk extends serial Java by adding new keywords that allow the program to execute in parallel. (JCilk currently omits entirely Java's multithreaded support as provided by the `Thread` class, but we hope eventually to integrate the JCilk extensions with Java threads.) If the JCilk keywords for parallel control are elided from a JCilk program, however, a syntactically correct Java program results, which we call the ***serial elision*** of the JCilk program. JCilk is a ***faithful*** extension of Java, because the serial elision of a JCilk program is a correct (but not necessarily the sole) interpretation of the JCilk program's parallel semantics.

To be specific, JCilk introduces three new keywords — `cilk`, `spawn`, and `sync` — which are the same keywords used to extend C into Cilk, and they have essentially the same meaning in JCilk as they do in Cilk. The keyword `cilk` is used as a method modifier to declare the method to be a ***`cilk` method***, which is analogous to a regular Java method except that it can be spawned to execute in parallel. When a parent method spawns a child method, which is accomplished by preceding the method call with the `spawn` keyword, the parent can continue to execute in parallel with its spawned child. The `sync` keyword acts as a local barrier. JCilk ensures that program control cannot go beyond a `sync` statement until all previously spawned children have terminated. In general, until a `cilk` method executes a `sync` statement, it cannot safely use results returned by previously spawned children.

To illustrate how we have introduced these Cilk primitives into Java, consider the simple JCilk program shown in Figure 1. The method `f1` spawns the method `A` to run in parallel in line 2, calls the method `B` normally (serially) in line 3, spawns `C` in parallel in line 4, calls method `D` normally in line 5, and then itself waits at the `sync` in line 6 until all the subcomputations `A` and `C` have completed. When they both complete, `f1` computes the sum of their returned values as its returned value in line 7.

```
1   cilk int f1() {
2      int w = spawn A();
3      int x = B();
4      int y = spawn C();
5      int z = D();
6      sync;
7      return w + x + y + z;
8   }
```

Figure 1: A simple JCilk program.

The original Cilk language provided these simple semantics for `spawn` and `sync` but with no semantics for exceptions, because spawned functions in Cilk can only return, just as C functions can only return. Java, however, allows a method to signal an exception rather than return normally, and JCilk's semantics must cope with this eventuality. How should the code in Figure 1 behave when one or more of the spawned or called methods signals an exception?

In ordinary Java, an exception causes a nonlocal transfer of control to nearest dynamically enclosing `catch` clause that handles the exception. The *Java Language Specification* [13, pp. 219–220] states,

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

In JCilk, we have striven to preserve these semantics while extending them to cope gracefully with the parallelism provided by the Cilk primitives. Specifically, JCilk extends the notion of "abruptly completes" to encompass the implicit aborting of any spawned side computations along the path from the point where the exception is thrown to the point where it is caught. Thus, for example, in Figure 1, 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?

We believe that JCilk provides good solutions to these subsidiary problems. JCilk provides for "semisynchronous" aborts to simplify the reasoning about program behavior when an abort occurs. The semantics of JCilk make it easy to understand the behavior of parallel code when exceptions occur, while faithfully extending Java semantics. JCilk provides for aborts themselves to be caught by defining a new subclass of `Throwable`, called `CilkAbort`, thereby allowing programmers to clean up an aborted subcomputation.

As a testament to how well JCilk integrates Java's exception mechanism with Cilk's `spawn` and `sync` constructs, programming speculative applications in JCilk is even more straightforward than in Cilk. Speculation is essential for parallelizing programs such as branch-and-bound or heuristic search [7, 9, 22]. The Cilk language provides the keywords `inlet` and `abort`, which allow speculative computation to be managed. JCilk's integration of Cilk's parallel control with `spawn` and `sync` and Java's exception-handling semantics make Cilk's `inlet` and `abort` keywords unnecessary for programming speculative applications such as the so-called "queens" problem and a parallel alpha-beta search. As we shall see, however, the inlet and abort mechanisms still exist conceptually within the JCilk language.

In this paper, we describe JCilk semantics and how Cilk-like multithreading is integrated with Java's existing exception semantics. (For descriptions of the implementation of JCilk's compiler and runtime system, see [8, 25].) Section 2 describes the basic concepts underlying JCilk, and Section 3 explains JCilk's exception semantics more precisely. Section 4 shows how JCilk's linguistic constructs can be used to program a search for a solution to the queens problem. Section 5 presents a parallel alpha-beta search application coded in JCilk, which demonstrates the use of JCilk's linguistics constructs in more depth. Section 6 presents related work, and Section 7 provides some concluding remarks.

2 Basic JCilk concepts

This section describes the basic concepts underlying the JCilk language beyond the simple `cilk`, `spawn`, and `sync` keywords described in Section 1. We present the language's syntax, its assumption of "implicit atomicity," and its built-in exception class `CilkAbort`. Section 3 will elaborate on how JCilk uses these concepts in its linguistic design.

Syntax

JCilk inherits its basic mechanisms for parallelism from Cilk. As mentioned in Section 1, JCilk includes three new keywords: `cilk`, `spawn`, and `sync`. The keyword `cilk` is used as a method modifier, and `spawn` and `sync` cannot be used in a Java method unless the method is a `cilk` method. In order to make parallelism plain to programmers, JCilk enforces the constraint that `spawn` and `sync` can only be used inside a method declared to be `cilk`. A `cilk` method can call a Java method, but a Java method cannot spawn (or call) a `cilk` method. Similarly, a `cilk` method can only be spawned but cannot be called. In addition to being a method modifier, the `cilk` keyword can be used as a modifier of a `try` statement, and JCilk enforces the constraint that `spawn` and `sync` keywords can only be used within a `cilk try` block, but not within any `catch` or `finally` clauses of the `cilk try` statement. Placing `spawn` or `sync` keywords within an ordinary `try` block is illegal in JCilk. The reason `try` blocks containing `spawn` and `sync` must be declared `cilk` is that when an exception occurs, these `try` statements may contain multiple threads of control during exception handling. Although a JCilk compiler could detect and automatically insert a `cilk` keyword before a `try` statement containing `spawn` or `sync`, we feel the programmer should be explicitly aware of the inherent parallelism. We disallow `spawn` and `sync` within `catch` or `finally` clauses for implementation simplicity, but we might consider revisiting this decision if a need arises.

Loci of control

When a `cilk` method is spawned, a ***locus of control*** is created for the method instance, which is more-or-less equivalent to its program counter. When the method returns, its locus of control is destroyed. For example, in the simple JCilk program from Figure 1, the spawning of `A` and `C` in lines 2 and 4 creates new loci of control that can execute `A` and `C` independently from their parent `f1`.

A `cilk` method contains only one ***primary*** locus of control. When it calls an ordinary Java (non-`cilk`) method, we view the Java method as executing using the `cilk` method's primary locus of control. In Figure 1, for example, the methods `B` and `D` in lines 3 and 5 execute using `f1`'s primary locus of control.

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

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

JCilk allows ***secondary*** loci of control to be created as well. In particular, when a `cilk` method is spawned, its return value is incorporated into the parent method by a secondary locus of control. Incorporating a return value may be more involved than for a simple assignment, as is shown in Figure 1 for variables `w` and `y`. Figure 2 illustrates a program in which the returned values from spawned methods `B` and `C` and called method `D` augment the variable `y`, rather than just assigning to it, as the return value from `A` does to the variable `x`. Although a child's locus of control normally stays within the child, for circumstances such as those in lines 4 and 5, the child's locus of control operates for a time in its parent `f2` to perform the update. JCilk encapsulates these secondary loci of control using a mechanism from the original Cilk language, called an ***inlet***, which is a small piece of code that operates within the parent on behalf of the child. Although Cilk's `inlet` keyword does not find its way into the JCilk language, as we shall see in Section 3, the concept of an inlet is used extensively when handling exceptions in JCilk.

Implicit atomicity

Since reasoning about race conditions between an inlet and the parent, or between inlets, could be problematic, JCilk supports the idea of ***implicit atomicity***. To understand this concept, we first define a ***JCilk thread***[1] to be a maximal sequence of instructions executed by the same locus of control that includes no parallel control. From a syntactic point of view, a JCilk thread contains no `spawn`, `sync`, or exit from a `cilk` block (`cilk` method or `cilk try`).

For example, when the method `f1` in Figure 1 runs, four threads are executed by `f1`'s primary locus of control:

1. from the beginning of `f1` to the point in line 2 where the `A` computation is actually spawned,
2. from the point in line 2 where the `A` computation is actually spawned to the point in line 4 where the `C` computation is actually spawned,
3. from the point in line 4 where the `C` computation is actually spawned to the `sync` in line 6,
4. from the `sync` in line 6 to the point where `f1` returns.

In addition, two threads corresponding to the assignments of `w` and `y` in lines 2 and 4 are executed by secondary loci of control.

In Figure 2, similar threads can be determined, but in addition, when a spawned method such as `B` in line 4 returns, an inlet runs the updating of `y` as a separate thread from the others. JCilk's support for implicit atomicity guarantees that all JCilk threads executing in the same method instance execute atomically with respect to each other, that is, the instructions of the threads do not interleave. Said more operationally, JCilk's scheduler performs all its actions at thread boundaries, and it executes only one of a method's threads at a time. In the case of `f2`, the updates of `y` in lines 4, 5, and 6 all execute atomically. The updates caused by the returns of `B` and `C` are executed by JCilk's built-in inlets, and the update caused by `D`'s return is executed by `f2`'s primary locus of control.

Implicit atomicity places no constraints on the interactions between two JCilk threads in different method instances, however. It is the responsibility of the programmer to handle those interactions using synchronized methods, locks, nonblocking synchronization (which can be subtle to implement in Java due to its memory model — see, for example, [12,24,27,34]), and other such techniques. We do not attempt here to address these synchronization issues, which appear to be orthogonal to the control issues discussed in this paper.

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[2] 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. The catching and handling of a `CilkAbort` exception is not required, however, and the `CilkAbort` exception is implemented as an unchecked exception.

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

3 The JCilk language features

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

The `cilk try` statement

Figure 3, which shows an example of the use of `cilk try`, demonstrates how this linguistic construct interacts with the spawning of subcomputations. The parent method `f3` spawns the child `cilk` method `A` in line 4, but its primary locus of control continues within the parent, proceeding to spawn 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 the primary locus of control were

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

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

```
 1  cilk int f3() {
 2     int x, y;
 3     cilk try {
 4        x = spawn A();
 5     } catch(Exception e) {
 6        x = 0;
 7     }
 8     cilk try {
 9        y = spawn B();
10     } catch(Exception e) {
11        y = 0;
12     }
13     sync;
14     return x + y;
15  }
```

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

held up at the end of the `cilk try` block, writing a `catch` clause would preclude parallelism.

In the code from the figure, if one of the children throws an exception, it is caught by the corresponding `catch` clause. The `catch` clause may be executed 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, 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 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 almost ready to tackle the full semantics of `cilk try`, which includes the aborting of side computations when an exception is thrown, but we require one key concept 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 dynamically enclosing `catch` clause handles the exception.

JCilk faithfully extends these semantics, using the notion of "dynamically enclosing" to determine, in a manner consistent with Java's notion of "abrupt completion," which method instances should be aborted. (See the quotation in Section 1.) Specifically, when an exception is thrown, JCilk delivers a `CilkAbort` exception semisynchronously to the ***side computations*** of the exception. The side computations include any method that is also dynamically enclosed by the `catch` clause that handles the exception. The side computations also include the primary locus of control of the method containing that `cilk try` statement if that locus of control still resides in the `cilk try` statement. JCilk thus throws a `CilkAbort` exception at the point of the primary locus of control in that case. Moreover, the `catch` clause handling the `CilkAbort` thrown a to-be-aborted `cilk` block is not executed

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

until all its children have completed, allowing the side computation to be "unwound" in a structured way from the leaves up.

Figure 4 shows a `cilk try` statement. If method `A` throws an exception that is caught by the `catch` clause beginning in line 6, the side computation that is signaled to be aborted includes `B` and any of its descendants, if B 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 `catch` clause.

JCilk makes no guarantees that the `CilkAbort` is handled quickly after it signals an exception's side computation to abort. It simply offers a best-effort attempt to do so. Linguistically, if the side computations are executed speculatively, and the overall correctness of a programmer's code then does not depend on whether the "aborted" methods complete normally or abruptly.

The semantics of `cilk try`

After an exception is thrown, when and how is it handled? The exception-handling mechanism decomposes exception handling into six actions:

1. Select an exception to be handled by the nearest dynamically enclosing `catch` clause that handles the exception.
2. Signal the side computations to be aborted.
3. Wait until all dynamically enclosed spawned methods complete, either normally or abruptly by dint of Action 2.
4. Wait until the primary locus of control for the method exits the `cilk try` block, either normally or by dint of Action 2.
5. Run the catchlet associated with the selected exception.
6. If the `cilk try` contains a `finally` clause, run the associated finallet.

The exception-handling mechanism executes these actions 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 their `catch` clauses do not handle it. Action 2 is then initiated to signal the side computation to abort. The mechanism now waits in Actions 3 and 4 until the side computations terminate. At this point Action 5 safely executes the `catch` clause, which is followed by Action 6 to execute the `finally` clause, if it exists.

We made the decision in JCilk that if multiple concurrent exceptions are thrown to the same `cilk` block, only one is selected to be handled. In particular, if one of these exceptions is a `CilkAbort` exception, the `CilkAbort` exception is selected to be handled.

```
 1  cilk int f5() {
 2     for(int i=0; i<10; i++) {
 3        int a = 0;
 4        cilk try {
 5           a = spawn A(i);
 6        } finally {
 7           System.out.println("In iteration "
 8                 + i + " A returns " + a);
 9        }
10     }
11     sync;
12  }
```

Figure 5: A loop containing a `cilk try` illustrating a race condition between the update of `i` in line 2 and the read of `i` in line 8.

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.

Secondary loci of control within loops

When a primary locus of control exits a `cilk try` block in a loop before its `catch` clause or `finally` clause is run and proceeds to another iteration of a loop, a secondary locus of control eventually executes the `catch` or `finally` clause. As in the Cilk language, this situation requires the programmer to reason carefully about the code.

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

This situation is called a ***data race*** (or, a ***general race***, as defined by Netzer and Miller in [30]), which occurs when two threads operating in parallel both access a variable and one modifies it. In this case, `f5`'s primary locus of control increments the value of `i` in line 2 in parallel with the secondary locus of control executing the `finally` block which reads `i` in line 8. Whereas JCilk's support for implicit atomicity guarantees that the `finally` block executes atomically with respect to `f5`'s primary locus of control, it does not guarantee that data races do not occur. In this case, the data race makes the code incorrect.

The race condition in the code from Figure 5 can be fixed by declaring a new loop local variable `icopy`, as shown in Figure 6. The only differences between code in Figure 5 and Figure 6 are the additional declaration of a loop variable, `icopy` in line 4 and the reading of `i` replaced with the reading of `icopy` in line 9. Every time `f6` iterates its loop, a new copy of variable `icopy` is created and initialized with the current value of `i`. When the `finally` clause executes on behalf of an iteration `i`, the `finally` clause reads and prints the corresponding value of `icopy` as determined by a ***lexical-scope rule*** [4, Sec. 7.4]. The JCilk compiler and runtime system provide an efficient implementation of the lexical-scope rule which avoids creating many extraneous versions of loop variables.

```
 1  cilk int f6() {
 2     for(int i=0; i<10; i++) {
 3        int a = 0;
 4        int icopy = i;
 5        cilk try {
 6           a = spawn A(icopy);
 7        } finally {
 8           System.out.println("In iteration "
 9                 + icopy + " A returns " + a);
10        }
11     }
12     sync;
13  }
```

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

```
 1  cilk void f7() {
 2     cilk try {
 3        spawn A()
 4     } catch(CilkAbort e) {
 5        cleanupA();
 6     }
 7     cilk try {
 8        spawn B()
 9     } catch(CilkAbort e) {
10        cleanupB();
11     }
12     cilk try {
13        spawn C()
14     } catch(CilkAbort e) {
15        cleanupC();
16     }
17     sync;
18  }
```

Figure 7: Catching `CilkAbort`.

Handling aborts

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

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

The code in Figure 7 shows how `CilkAbort` exceptions can be caught. If any of `A`, `B`, or `C` throws an exception that is not handled within `f7` while the others are still executing, then those others are aborted. Any spawned methods that abort have their corresponding cleanup method called.

4 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

```
 1  public class Queens {
 2     private int n;
       .
       .
       .
 3     private cilk void q(int[] cfg, int row)
             throws Result {
 4        if(row == n) {
 5           throw new Result(cfg);
 6        }

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

11           if(safe(row, col, ncfg)) {
12              spawn q(ncfg, row+1);
13           }
14        }
15        sync;
16     }

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

21        cilk try {
22           spawn (new Queens(n)).q(cfg, 0);
23        } catch(Result e) {
24           ans = (int[]) e.getValue();
25        }
26        sync;

27        if(ans != null) {
28           System.out.print("Solution: ");
29           for(int i = 0; i < n; i++) {
30              System.out.print(ans[i] + " ");
31           }
32           System.out.print("\n");
33        } else {
34           System.out.println("No solutions.");
35        }
36     }
37  }
```

Figure 8: 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 extends class `Exception`) is used to notify the `main` method when a result is found.

n-by-n chessboard such that no queen attacks another, that is, no two queens occupy the same row, column, or diagonal. Figure 8 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 a highly parallel program.

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 its `q` method to search for a safe configuration. Method `q` takes in two arguments: `cfg`, which is the current configuration of queens on the board, and `row`, which is 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 ordinary Java method `safe`, whose definition we omit for brevity, 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 with the new queen placed in the position (`row`, `col`).

The newly spawned subsearch can run in parallel with all other subsearches spawned so far. The parallel search continues until it finds a configuration in which every row contains a queen. At this point `cfg` contains a legal placement of all n queens. The successful `q` method throws the user-defined exception `Result` (whose definition we also omit for brevity) to signal that it has found a solution. The `Result` exception is used to communicate between the `q` and `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 by the catching clause. In this example, since the `Result` exception propagates all the way up to the `main` method, all outstanding `q` methods will be aborted automatically. To ensure that all side computations have terminated and the `catch` clause has been executed, the `main` method executes a `sync` statement before it prints out the solution.

5 Parallel alpha-beta search

This section explores the coding of a parallel alpha-beta search in JCilk, which highlights JCilk's semantics in more depth. Like the queens algorithm, our alpha-beta code exploits JCilk's exception-handling mechanism to abort speculative computations that are found to be unnecessary. In addition, this JCilk program provides an example that exploits the implicit lexical-scope rule to ensure correct execution.

Alpha-beta search [21,41] is often used when programming two-player games such as chess or checkers. It is basically a "minimax" [36] search algorithm applied with "alpha-beta pruning" [36], a technique for pruning out irrelevant parts of the game tree so that more ply of depth can be searched within a given time bound. Since the search algorithm is described in virtually every introduction to adversarial search (see, for example, [36, Ch. 6] and [41, Ch. 6]), we assume the basic familiarity with the search strategy in the paper. The basic idea of the algorithm is that, if White can make a move in a position so good that Black will not make the move leading to that position, then there is no point in searching White's other moves from that position. Therefore, those additional moves can be pruned. This situation is referred as a ***beta cutoff***.

The basic alpha-beta search algorithm is inherently serial, because the information from searching one child of a node in the game tree is used to prune subsequent children. It is difficult to use information gained from searching one child to prune another if one wishes to search all children in parallel.

One key observation helps to parallelize alpha-beta search: in an optimal game tree, either all children of a node are searched (the node is ***maximal***), or only one child needs to be searched to generate a cutoff (the node is ***singular***). This observation suggests a parallel search strategy called ***young brothers wait*** [9]: if the first child searched fails to generate a cutoff, speculate that the node is maximal, and thus searching the rest of the children in parallel wastes no

work. To implement this strategy, the parallel alpha-beta algorithm first searches what it considers to be the best child. If the score returned by the best child generates a cutoff, the rest of the children are pruned, and the search returns immediately. Otherwise, the algorithm speculates that the node is maximal, and spawns searches of all the remaining children in parallel. If one of the children returns a score that generates a beta cutoff, however, the other children are aborted, since their work has been rendered unnecessary.

Figure 9 shows a JCilk implementation of this parallel search algorithm using the ***negamax*** strategy [21], where scores are always viewed from the perspective of the side to move in the game tree. In this strategy, when subsequent moves are searched, the `alpha` and `beta` roles are reversed and the scores returned are negated. The `search` method is called with the current board configuration, the depth to search, and the `alpha` and `beta` values that bound the search of the current node. When invoked, the code first checks for the base case by calling the method `isDone` in line 4, which basically returns `true` if this node is at the leaf of the game tree (meaning the depth is reached), the board configuration is a draw, or one side has lost. (The definition for `isDone` is omitted for simplicity.) If `isDone` returns `true`, the algorithm evaluates and returns the score of the current board configuration. Otherwise, it generates a list `successors` of legal moves that can be made from the current board configuration. This `successors` list contains the moves in best-first order as determined by more-ordering heuristics.

The search begins with the first move stored in the `successors` list, which ostensibly corresponds to the best child. When this child returns with a score, `alpha` is updated, and the condition for a beta cutoff is checked. If the score generates a beta cutoff (meaning this node is singular), the score for this node (which is stored in `beta` in this case) is returned. If the score does not generate a beta cutoff, the algorithm then proceeds to spawn the rest of the children in parallel, with the remaining moves stored in the `successors` list. As each of these children returns, the `alpha` value is again updated and the condition for a beta cutoff is checked. If any of these children happens to generate a beta cutoff, a user-defined exception `Result` (whose definition is omitted) is thrown, causing all children spawned in parallel by this node to be aborted. The `Result` object contains a single field to store the score of the node so that the score can be communicated back to its parent.

The `search` method is first invoked by the `rootSearch` method, which initiates the searches from the root node. The definition of the `rootSearch` method is omitted because it is similar to the definition of the `search` method. The only differences are that no checks for beta cutoffs are performed, because no beta cutoff can occur at the root of the game tree, and the value for `beta` is initialized to the maximum value that can be represented with an `int` type. One could merge `rootSearch` and `search` into a single method with a flag indicating whether the current node is the root node, but we chose to separate them into distinct methods for simplicity.

The code for the `search` method shown in Figure 9 capitalizes on three JCilk language features:

- implicit abort semantics,
- the lexical-scope rule,
- implicit atomicity.

We now examine how `search` makes use of each of these features.

First, the `search` method exploits JCilk's implicit abort semantics to abort extraneous computations spawned in line 30. This part of the code is similar to line 12 in the queens code from Figure 8.

Second, the code exploits JCilk's support for the lexical-scope rule. Specifically, the `finally` clause (lines 33–41) is contained within a loop, and it refers to the loop local variable `score2`.

```
 1  private cilk int search(Board board,
                            int depth,
                            int alpha,
                            int beta)
 2  throws Result {
 3     int score1;

 4     if(isDone(board, depth)) {
 5        return eval(board);
 6     }
 7     List successors = board.legalMoves();
 8     List move = (List) successors.pop_front();
 9     Board nextBoard = (Board) board.copy();
10     nextBoard.move(move);
11     cilk try {
12        score1 = spawn search(nextBoard,
                               depth+1,
                               -beta,
                               -alpha);
13     } catch(Result e) {
14        score1 = e.getValue();
15     }
16     sync;
17     score1 = -score1;
18     if(score1 > alpha) {
19        alpha = score1;
20        if(score1 >= beta) {
21           return beta;
22        }
23     }

24     while(mayPlay (successors)) {
25        int score2 = -Integer.MAX_VALUE;
26        move = (List) successors.pop_front();
27        nextBoard = (Board) board.copy();
28        nextBoard.move(move);
29        cilk try {
30           score2 = spawn search(nextBoard,
                                   depth+1,
                                   -beta,
                                   -alpha);
31        } catch(Result e) {
32           score2 = e.getValue();
33        } finally {
34           score2 = -score2;
35           if(score2 > alpha) {
36              alpha = score2;
37              if(score2 >= beta) {
38                 throw new Result(beta);
39              }
40           }
41        }
42     }
43     sync;

44     return alpha;
45  }
```

Figure 9: The core of a parallel alpha-beta search.

Since `score2` is declared within the loop (in line 25), the lexical-scope rule applies. When each `finally` clause refers to `score2`, it resolves to the version corresponding to the iteration to which the `finally` belongs lexically. This "correct" resolution of `score2` is crucial to the correctness of the alpha-beta code.

Third, the code exploits JCilk's guarantee of implicit atomicity. In particular, in the same `finally` clause (lines 33–41), an assignment to the local variable `alpha` is made in line 36. Even though `alpha` is written simultaneously by multiple secondary loci of control (executing `finally` clauses from different iterations), no data races exist. JCilk's guarantee of implicit atomicity allows all the instantiations of the `finally` clause to execute atomically with respect to one another. Since the order of their execution does not matter, the code is correct.

This parallel alpha-beta search demonstrates the expressiveness of JCilk's language features and their semantics. Without the sup-

port of any one of these three features, the parallel alpha-beta search could not be programmed so easily. Compared to a parallel alpha-beta search coded in Cilk [7], this implementation is arguably cleaner and simpler.

6 Related work

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

Most parallel languages do not provide an exception-handling mechanism. For example, none of the parallel functional languages VAL [1], SISAL [11], Id [31], parallel Haskell [3, 32], MultiLisp [15], and NESL [5] and none of the parallel imperative languages Fortran 90 [2], High Performance Fortran [35] [29], Declarative Ada [39, 40], C* [16], Dataparallel C [17], Split-C [6], and Cilk [37] contain exception-handling mechanisms. The reason for this omission is simple: these languages were derived from serial languages that lacked such linguistics.[3]

Some 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 [33], which provides a set of compiler directives and library functions compatible with C++; and Java Fork/Join Framework [23], 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.

Tazuneki and Yoshida [38] and Issarny [19] have investigated the semantics of concurrent exception-handling, taking different approaches from our work. In particular, these researchers pursue new linguistic mechanisms for concurrent exceptions, rather than extending them faithfully from a serial base language as does JCilk. The treatment of multiple exceptions thrown simultaneously is another point of divergence.

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

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

The recent version of the Java Language, known as Tiger or Java 1.5 during development and now called Java 5.0 [28], provides call-return semantics for threads similar on the surface to JCilk. In particular, Java 1.5 provides a protocol that is similar to that of JCilk. Although Java 5.0 (like everything else in Java) uses an object-based semantics for multithreading, rather than JCilk's choice of a linguistic semantics, it does move in the direction of providing more linguistic support for multithreading. In particular, Java 5.0 introduces the `Executor` interface, which provides a mechanism to decouple the scheduling from execution. It also introduces the `Callable` interface, which, like the earlier `Runnable` interface, encapsulates a method which can be run at a later time (and potentially on a different thread). Unlike `Runnable`, `Callable` allows its encapsulated method to return a value or throw an exception. When a `Callable` is submitted to an `Executor`, it returns a `Future` object. The `get` method of that object waits for the `Callable` to complete, and then it returns the value that the `Callable`'s method returned. If that method throws an exception, then `Future.get` throws an `ExecutionException` containing the original exception as its cause. (The `Future` object also provides a nonblocking `isDone` method to see if the `Callable` is already done.)

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

7 Conclusions

CLU [26] was the first language to cleanly define the semantics for an exception-handling mechanism, but only in a serial context. Although much effort has been spent on developing tools, software, and languages to aid in the writing of multithreaded programs, comparatively little research explores how exception mechanisms should be extended to a concurrent context. The JCilk language explores how concurrency can be made semantically consistent with the exception mechanisms of modern serial computing. Our research leaves us optimistic that the sometimes-arcane world of parallel computing and the day-to-day world of commodity computing may eventually be united.

Acknowledgments

Special thanks to Scott Ananian of MIT CSAIL for his copious comments on the paper and helpful discussions. Thanks to Kunal Agrawal, Jeremy Fineman, Viktor Kuncak, Bradley Kuszmaul, Martin Rinard, Gideon Stupp, and Jim Sukha also of MIT CSAIL and Wong Weng Fai of National University of Singapore for helpful discussions.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, R. J. Anderson, and R. 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] Institute of Electrical and Electronic Engineers. Information technology —Portable Operating System Interface (POSIX) —Part 1: System application program interface (API) [C language]. IEEE Std 1003.1, 1996 Edition.

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

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

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

[22] B. C. Kuszmaul. The StarTech massively parallel chess program. *The Journal of the International Computer Chess Association*, 18(1):3–20, Mar. 1995.

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

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

[25] I.-T. A. Lee. The JCilk multithreaded language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Aug. 2005.

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

[27] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, Jan 2005.

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

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

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

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

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

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

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

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

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

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

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

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

[40] J. Thornley. Declarative Ada: parallel datafbw 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.

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