Programming with Exceptions in JCilk [1]

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

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

Abstract

JCilk extends the serial subset of the Java language by importing the fork-join primitives `spawn` and `sync` from the Cilk multithreaded language, thereby providing call-return semantics for multithreaded subcomputations. In addition, JCilk transparently integrates Java's exception handling with multithreading by extending the semantics of Java's `try` and `catch` constructs, but without adding new keywords. 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 causes its sibling computations to abort, which yields a clean semantics in which the enclosing `cilk try` block need only handle a single exception.

The exception semantics of JCilk allow programs with speculative computations to be programmed easily. Speculation is essential in order to parallelize programs such as branch-and-bound or heuristic search. We show how JCilk's linguistic mechanisms can be used to program the "queens" puzzle and a parallel alpha-beta search.

We have implemented JCilk's semantic model in a prototype compiler and runtime system, called JCilk-1. The compiler implements continuations in a novel fashion by introducing `goto` statements into Java. The JCilk-1 runtime system shadows the dynamic hierarchy of `cilk try` blocks using a "try tree," allowing the system to chase down side computations that must be aborted. Performance studies indicate that JCilk's exception mechanism incurs minimal overhead, contributing at most a few percent on top of the cost for a `spawn`/`return`.

1 Introduction

JCilk is a Java-based multithreaded language for parallel programming that extends the semantics of Java (Gosling *et al.*, 2000) by introducing "Cilk-like" (SuperTech,

[1] This research was supported in part by the Singapore-MIT Alliance and by NSF Grant ACI-0324974. I-Ting Angelina Lee was supported in part by a Sun Microsystems Fellowship. John S. Danaher is now employed at Google, Inc.

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

Fig. 1. A simple JCilk program.

2001; Frigo *et al.*, 1998) 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 (Kernighan and Ritchie, 1988) context. These facilities are not available in Java's threading model (Gosling *et al.*, 2000, Ch. 11) or in the Posix thread specification (Institute of Electrical and Electronic Engineers, 1996) 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*** (Frigo *et al.*, 1998) 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.

The original Cilk language provides these simple semantics for `spawn` and `sync`, but it requires no semantics for exceptions, because spawned functions in Cilk can only return normally, just as C functions can only return normally. 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* (Gosling *et al.*, 2000, 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 potentially opens a Pandora's box of subsidiary linguistic problems to be resolved. Aborting might cause a computation to be interrupted asynchronously (Gosling *et al.*, 2000, 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 (Feldmann *et al.*, 1993; Kuszmaul, 1995; Dailey and Leiserson, 2002). The Cilk language provides the keywords `inlet` and `abort`, which allow speculative computations to be managed. JCilk's integration of `spawn` and `sync` with Java's exception-handling semantics obviates Cilk's `inlet` and `abort` keywords for programming speculative applications such as the so-called "queens" puzzle and 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's semantics and how Cilk-like multithreading is integrated with Java's existing exception semantics. 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 puzzle. 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 overviews the prototype JCilk-1 compiler and runtime system which implements the JCilk language semantics. Section 7 describes the implementation of exceptions in JCilk-1, which keeps track of the dynamic hierarchy of `cilk try` statements using a "try tree" data structure. Section 8 evaluates the performance of JCilk-1's exception mechanism and offers insight into how a highly tuned implementation would perform. Section 9 presents related work, and Section 10 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 first present the language's syntax. We go on to describe the notion of a program cursor, which is analogous to a program counter. We then discuss how JCilk's support for "implicit atomicity" simplifies reasoning about concurrency. Finally, we describe how users can safely clean up aborting methods using the built-in exception class `CilkAbort`.

Syntax

JCilk inherits its basic mechanisms for parallelism from Cilk. As mentioned in Section 1, the JCilk language extends Java by including three new keywords: `cilk`, `spawn`, and `sync`. The keyword `cilk` is a method modifier. In order to make parallelism manifest 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, and because we know of no applications that would benefit from this flexibility.

Program cursors

When a `cilk` method is spawned, a ***program cursor*** is created for the method instance, which is more-or-less equivalent to its program counter, but at the language level rather than machine level. When the method returns, its program cursor 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 program cursors that can execute `A` and `C` independently from their parent `f1`.

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

JCilk allows ***secondary*** program cursors 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 cursor. Incorporating a return value may be more involved than the case of a simple assignment, such as the ones shown in lines 2 and 4 in Figure 1. 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 cursor normally stays within the child, for circumstances such as those in lines 4 and 5, the child's cursor operates for a time in its parent `f2` to perform the update. JCilk encapsulates these secondary cursors 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

```
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   }
```

Fig. 2. Implicit atomicity simplifies reasoning about multiple JCilk threads operating within the same method.

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***[2] to be a maximal sequence of statements executed by a single program cursor that includes no parallel control. From a linguistic point of view, a JCilk thread executes no `spawn` or `sync` statements, nor exits from a `cilk` method or `cilk try`.

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

(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, including the entire call to `B`;
(3) from the point in line 4 where the `C` computation is actually spawned to the `sync` in line 6, including the entire call to `D`;
(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 program cursors.

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

[2] 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.

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 instance'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 program cursor.

Implicit atomicity places no constraints on the interactions between 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, Lea (1999); Pugh (2000); Gontmakher and Schuster (2000); Manson *et al.* (2005) — and other such techniques. This paper does not address these synchronization issues, which are orthogonal to our focus on the linguistic constructs for exceptions.

Because of the havoc that can be caused by aborting computations asynchronously, JCilk leverages the notion of implicit atomicity by ensuring that all aborts occur ***semisynchronously***; that is, when a method is aborted, all its program cursors reside at thread boundaries. 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` returns from `D` and continues on 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 program cursor reaches the next thread boundary. What it promises is only that when an abort occurs, the primary cursor resides at *some* thread boundary, and likewise for secondary cursors.

The `CilkAbort` exception

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

[3] In keeping with the usage in Gosling *et al.* (2000), 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   }
```

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

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 program cursor 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 illustrates the use of `cilk try` and 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 program cursor continues within the parent, proceeding to spawn another child `B` in line 9. As before, the primary cursor 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 cursor can continue on beyond the scope of the `cilk try` statements even though `A` and `B` may yet throw exceptions. If the primary cursor were 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 cursor 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 cursor 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 program cursors running on `f3`.

Similar to a catchlet, a ***finallet*** runs atomically with respect to other program cursors 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 (Gosling *et al.*, 2000, 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 all methods that are dynamically enclosed by the `catch` clause that handles the exception, which can include the primary program cursor of the method containing that `cilk try` statement if that cursor still resides in the `cilk try` statement. JCilk thus throws a `CilkAbort` exception at the point of the primary cursor in that case. Moreover, the `catch` clause handling the `CilkAbort` thrown to a to-be-aborted `cilk` block is not executed 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 program cursor 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.

```
cilk int f4() {
   int x, y, z;
   cilk try {
      x = spawn A();
      y = spawn B();
   } catch(Exception e) {
      x = y = 0;
      handle(e);
   }
   z = spawn C();
   sync;
   return x + y + z;
}
```

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

Although JCilk makes no guarantee that the abort executes quickly after an exception's side computation is signaled to abort, it makes a best-effort attempt to do so. If the side computations are executed speculatively, the overall correctness of a programmer's code should not depend on whether the "aborted" methods complete normally or abruptly, and if abruptly, quickly or slowly.

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 method's primary program cursor 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.

```
 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   }
```

Fig. 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.

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

The decision to allow the primary program cursor 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 (Gosling *et al.*, 2000, Ch. 11), not to establish a precondition for subsequent execution. Moreover, JCilk does provide a way 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 program cursors within loops

When a primary program cursor 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 program cursor 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 cursor may change the value of `i` before a given child completes, thereby causing the secondary cursor created when the child returns to use the wrong value when it executes the print statement in line 8 in the `finally` clause.

```
 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   }
```

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

This situation is called a ***data race*** (or, a ***general race***, as defined by Netzer and Miller (1992)), which occurs when two threads operating in parallel both access a variable and one modifies it. In this case, `f5`'s primary cursor increments the value of `i` in line 2 in parallel with the secondary cursor 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 cursor, 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 the loop variable `icopy` in line 4 of Figure 6 and replacing the reading of `i` in line 8 of Figure 5 with the reading of `icopy` in line 9 of Figure 6. Every time `f6` iterates its loop, a new copy of the 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*** (Aho *et al.*, 1986, 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.

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

```
cilk void f7() {
   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;
}
```

Fig. 7. Catching `CilkAbort`.

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 puzzle

This section illustrates how a parallel solution to the so-called "queens" puzzle can be programmed using the JCilk extensions to Java. 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 8 shows the JCilk code. 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. When one branch discovers such a configuration, the others abort. JCilk's exception mechanism makes this strategy easy to implement.

Although speculation can enhance parallelism, it can be ineffective, because the program incurs more work. For speculation to be effective, the chances should be good that the speculative computation will need to be performed. The queens pro-

```
public class Queens {
   private int n;
   ⋮
   private cilk void q(int[] cfg, int row) throws Result {
      boolean flag = true;
      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);
            if(flag) {
               sync;
               flag = false;
            }
         }
      }
      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;

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

Fig. 8. The queens puzzle coded in JCilk. 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`) notifies the `main` method when a result is found.

gram uses the heuristic that if the first child of a node in the search tree does not contain a safe configuration, then neither do its siblings. Thus, it spawns off the first child serially, and only when that child returns (unsuccessfully) does it spawn off the remaining children in parallel.

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 the `q` method to search for a safe configuration. The `q` method takes in two arguments: the current configuration `cfg` of queens on the board, and the current row `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, a child `q` method is spawned in line 13 to perform the subsearch with the new queen placed in the position (`row`, `col`).

After the first child of the current node is spawned, the `q` method executes a `sync` in line 15, suspending the method until the first child returns. By setting the boolean `flag` to `false`, subsequent children are spawned without an immediate `sync`, thereby allowing them to run in parallel.

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) in line 6 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 upward until it is caught in line 28 of the `main` method, all outstanding `q` methods abort automatically. To ensure that all side computations have terminated and the `catch` clause has been executed, the `main` method executes a `sync` statement in line 31 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 program, 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 (Knuth and Moore, 1975; Winston, 1992) is often used when programming two-player games such as chess or checkers. It is basically a "mini-max" (Russell and Norvig, 2003) search algorithm applied with "alpha-beta pruning" (Russell and Norvig, 2003), 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, Russell and Norvig (2003, Ch. 6) and Winston (1992, Ch. 6)), we assume a basic familiarity with this search strategy. The idea of the algorithm is that if White can make a move in a position so good that Black would 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 in what is termed 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 game tree in which children are ordered optimally at every none, either all the 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*** (Feldmann *et al.*, 1993): if the first child searched fails to generate a cutoff, the algorithm speculates 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 algorithm prunes the rest of the children and 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 superfluous.

Figure 9 shows a JCilk implementation of this parallel search algorithm using the ***negamax*** strategy (Knuth and Moore, 1975), 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 3, which returns `true` if this node is a leaf of the game tree: the depth has been 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 a "static evaluation" or "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 move-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 values for `alpha` and `beta` are initialized to the minimum and maximum values that can be represented with an `int` type, respectively. 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 29. This part of the code is similar to line 13 in the queens code from Figure 8.

Second, the code exploits JCilk's support for the lexical-scope rule. Specifically, the `finally` clause (lines 32–40) is contained within a loop, and it refers to the loop local variable `score2`. Since `score2` is declared within the loop (in line 24), the

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

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

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

Fig. 9. A parallel alpha-beta search coded in JCilk.

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 32–40), an assignment to the local variable `alpha` is made in line 35. Even though `alpha` is written simultaneously by multiple secondary program cursors (executing `finally` clauses from different iterations), JCilk's guarantee of implicit atomicity causes 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 support 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 (Dailey and Leiserson, 2002), this implementation is arguably cleaner and simpler.

6 The JCilk-1 prototype implementation

We have implemented the JCilk semantics in a prototype system called JCilk-1. Although an ideal implementation of JCilk might incorporate a JCilk virtual machine analogous to a Java Virtual Machine (JVM) (Lindholm and Yellin, 2000) and a compiler that translates directly to bytecode, the JCilk-1 strategy required much less work. JCilk-1 consists of two components — a runtime system and a compiler — both which heavily leverage the existing Java infrastructure, albeit at the cost of some overheads that would not be incurred by an ideal implementation. JCilk-1's runtime system is implemented in Java and is modeled after the Cilk runtime system (SuperTech, 2001; Frigo *et al.*, 1998) which incorporates a randomized work-stealing scheduler. The JCilk-1 compiler compiles JCilk source code into Java bytecode with library calls to the runtime system. The bytecode along with the runtime libraries can be executed on any standard JVM. This section overviews the structure of the JCilk-1 runtime system and compiler.

The JCilk-1 runtime system

JCilk-1's runtime system schedules threads dynamically according to available processor resources using a Cilk-like work-stealing scheduling algorithm (Blumofe and Leiserson, 1999; Frigo *et al.*, 1998). A collection of Java threads, called ***workers***, schedule and execute the JCilk threads. Each worker maintains a ***ready deque*** (doubly-ended queue) of ***activation frames*** each containing the variables associated with the corresponding method instances that are ready to execute. Each deque has two ends, a ***head*** and a ***tail***, from which frames can be added or removed. A worker operates locally on the tail of its own deque, treating it much as Java treats its call stack, pushing and popping spawned frames. When a worker runs out of work, it becomes a ***thief*** and attempts to steal a frame from another worker, called its ***victim***, at random. The thief steals the frame from the head of the victim's deque, that is, the opposite end from which the victim is working. The stolen frame is always the oldest frame in the victim's deque. If the victim has no work to be stolen, the thief simply chooses another victim at random and repeats the process.

As an example of how work-stealing operates, suppose that a method A spawns a method B. The worker executing A immediately begins work on B, leaving A for later resumption. If a thief steals A, it resumes the execution from where the original (victim) worker left off. Later, when B attempts to return control to its parent A, it instead notifies A's current worker that its subcomputation has completed. It passes B's result, which can be either a return value or a thrown exception, back to A's current worker.

If the method was spawned as part of an assignment operation and has returned a value, then that value must eventually be stored into the appropriate variable. This action is accomplished by an inlet created by the compiler for this purpose. The inlet takes the return value as an argument and assigns it either to a field in some object or to an entry in the method's frame on the ready deque. An inlet created from a more complex assignment operation (such as `x += spawn A()`) might perform a small operation (in this case, an addition) before storing the result. The inlet executes atomically with respect to other threads executing on the method where the return value is stored. As we shall see in Section 7, the JCilk-1 compiler also creates catchlets and finallets from the code in `catch` and `finally` clauses, respectively, as part of JCilk's exception-handling mechanism.

The JCilk-1 compiler

The driving philosophy in the design of the JCilk-1 compiler has been that a user should pay the overhead of running parallel code only when using JCilk's parallel extensions. A JCilk program should be compiled so that blocks containing only regular Java code run without any compiler-induced slowdown. The JCilk-1 compiler borrows heavily from the two-clone compilation strategy (Frigo *et al.*, 1998) used in Cilk to minimize the work overhead.

JCilk-1's compiler capitalizes on an important property of its work-stealing algorithm. The frame at the head of a worker's deque is a special frame called a ***closure***. A closure's bookkeeping is somewhat more complicated than that for ordinary frames, because a closure may have several children executing on different workers. In contrast, all other frames on the deque are much simpler, in part because each has at most one child, which executes on the same processor. When the JCilk-1 compiler compiles a method, it produces two separate ***clones*** of the method as output. The ***slow clone*** handles all the vagaries of bookkeeping for closures, whereas the ***fast clone*** is optimized for the common case of an ordinary frame having only a single child executing on the same processor.

We would have preferred that the JCilk-1 compiler mimic the Cilk strategy (as described in Frigo *et al.* (1998)) by performing a JCilk-to-Java translation, translating only JCilk keywords while leaving regular Java code intact. The generated Java postsource would have the same general structure as the original JCilk program, but the JCilk keywords `cilk`, `spawn`, and `sync` would be expanded into the Java statements necessary to actually accomplish their functionality. Unfortunately, this strategy does not work.

To see why, remember that the JCilk keywords define the boundaries of JCilk threads, as described in Section 2. These boundaries are points where the method instance's frame can potentially migrate from one worker to another. That is, two JCilk threads separated by a boundary might execute on two different workers. When migration occurs during a steal, the thief needs access to the most recent state of local variables and must reset its primary program cursor to the point where the victim left off. The mechanism to allow this resumption is called a ***continuation***. The Cilk system, which is implemented in C, supports a continuation mechanism using `goto` statements.

Adopting this approach for JCilk-1 is problematic, however, since the Java language has no `goto` statement. To support a continuation mechanism without slowing down pure Java code, we created an intermediate language called GoJava which is a minimal extension of Java to allow `goto` statements in limited and specific circumstances. Since Java bytecode already contains jump instructions, compiling a GoJava program into ordinary Java bytecode required minimal changes to a Java compiler.

JCilk-1's compiler compiles a JCilk program using a two-stage compilation process. The first stage is a source-to-source translation from JCilk to GoJava. This stage expands all JCilk keywords into their effects and leaves regular Java code unaffected. This translation from JCilk source to GoJava postsource is performed using Polyglot (Nystrom *et al.*, 2003), a compiler toolkit designed specifically for implementing Java language extensions. The second stage of the compilation process translates the GoJava postsource to Java bytecode using Jgo, a compiler for GoJava which we created by minimally modifying GCJ, the Gnu Compiler for Java (GNU, 2004). This second stage adds no additional overhead as compared to using GCJ directly, maintaining the property that pure Java code suffers no slowdown.

Figure 10 shows the GoJava postsource from the first compilation stage when run on the `q` method of the JCilk Queens code from Figure 8. Notice that the `spawn`

```
private void q(Worker worker, CilkFrame frame)
throws Result {
   Queens_nqueens_frame f = (Queens_nqueens_frame)frame;
   switch(f._pc) {
   case 1:
      goto _cilk_sync1;
   case 2:
      goto _cilk_sync2;
   }
   ⋮
   for(f._col = 0; f._col < n; f._col++) {
      ⋮
      if(safe(f._row, f._col, f._ncfg)) {
         f._pc = 1;
         try {
            q_fast(worker, nconfig, f._row+1);
         } catch(Result e) {
            if(worker.popFrameCheckExc(e)) {
               return;
            } else {
               throw e;
            }
         } catch (RuntimeException e) {
            ⋮
         }
         if(worker.popFrameCheck(null)) {
            return;
         }
         _cilk_sync1: ;
         if(f._flag) {
            f._pc = 2;
            if(!worker.sync()) {
               return;
            }
            _cilk_sync2: ;
            f._flag = false;
         }
      }
   }
   f._pc = 3;
   if(!worker.sync()) {
      return;
   }
   _cilk_sync3: ;
   return;
}
```

Fig. 10. The GoJava output of the JCilk compiler's first stage when run on the `q` method of the JCilk Queens code from Figure 8. This code is for the slow clone in the two-clone compilation strategy (Frigo *et al.*, 1998), and it is only called when a stolen (migrated) computation is resumed.

statement has been replaced with lines 12–26. Before the call, the current primary program cursor (line 12) is stored into the frame. After the call completes (either with an exception or a normal return value), a call to the worker (lines 16 and 23) checks to see whether a steal has occurred and immediately returns from the method if it has. A thief that steals this method reads the `pc` field in the frame and jumps from the top of the method (line 6) to the appropriate continuation point, here at line 26.

7 JCilk-1's implementation of exceptions

The implementation of JCilk's exception-handling mechanism is based on a data structure, called a "try tree," which shadows the dynamic hierarchy of `cilk try` statements. This section shows how JCilk-1 uses try trees to choose an exception to handle, to signal aborts to side computations, and to signal an abort at the catching method's program cursor when necessary. We also describe how catchlets are used to execute `catch` clauses atomically.

The try tree

Due to the potential parallelism in a `cilk` block, JCilk-1 must be ready to handle whatever exceptions may arise out of the parallel computations spawned from within the same `cilk` block. The system must select only one of possibly many concurrent exceptions to be handled. Moreover, it must determine which side computations should be aborted and signal them, which can be complicated.

As an example, consider the methods in Figure 11. The method `threeWay` contains three spawns: method `A` is not enclosed by any `cilk try` statement, method `B` is enclosed by one `cilk try` statement, and method `C` is enclosed by two nested `cilk try` statements. Depending on what kind of exception the call to `C()` throws, different sets of spawned methods might receive the abort signal. For example, if `C` throws a `RuntimeException`, then `B` could be aborted (assuming it was still running), but `A` would continue normally.

In order to determine which spawned child methods should be aborted, the worker must track the location in the parent method where they were originally spawned. This information is maintained using a data structure called a ***try tree***. In the same way that the ready deque mirrors the Java call stack, the try tree mirrors the dynamic hierarchy of nested `cilk try` statements. Each worker "owns" one try tree.

Because of the way that work-stealing scheduler operates (described in Section 6), the closure is the only frame within the deque that might have children running on other workers. Thus, it is sufficient to maintain the try tree only in closures. In

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

Fig. 11. A method containing nested `cilk` blocks, each containing a `spawn` statement.

addition, maintaining the try tree in the closure does not add significant overhead, because the vast majority of the work is done deeper in the ready deque.

The try tree for a worker tracks in which `cilk` block the top-level method (represented by the closure) spawned off children currently executing on other workers. Each internal node represents a `cilk` block, which can be either a `cilk` method body or a `cilk try` block. A leaf represents either the primary program cursor on the try tree's worker or a spawned call currently executing on a different worker. A node's parent in the try tree represents the `cilk` block most directly containing the node. Thus, each leaf's parent corresponds to the `cilk` block from which the method was spawned.

One of the nodes in the try tree is designated as the ***cursor node***. The cursor node tracks the `cilk` block containing the worker's current primary program cursor. The cursor node can be either a leaf or an internal node. When the cursor node is a leaf, it means that a child method is spawned but is being executed on the same worker.

Maintaining the try tree is straightforward. Whenever the top-level method enters a `cilk try` statement, a new node representing the `cilk try` block is created as a child of the cursor node, and the cursor node is moved down to the new node. Whenever the top-level method leaves a `cilk try` statement normally (not as a result of a thrown exception), the cursor node moves up a level. Whenever work is stolen, the try tree is migrated over to the thief along with the closure, and a new node is created as a child of the current cursor node, representing a spawn call now executing on the victim. Whenever a spawn call executing on a different worker completes (either normally or abruptly), the leaf representing the spawn call is then removed from the try tree.

The abort signal

Before a side computation can be aborted, it first must be signaled to abort. The abort signal is delivered to a worker via a flag that the worker checks only at thread boundaries: when a spawned method returns, at a `sync` statement, or when a `cilk try` statement completes. Each time a spawned method returns, if the abort flag is set to indicate the worker has received a signal to abort, then whatever value (if any) was being returned by the spawned method is discarded. In its place, a (new) `CilkAbort` exception is thrown.

Although there are no return values to replace in cases where abort happens after a `sync` statement or when a `cilk try` completes, a (new) `CilkAbort` is still thrown. This strategy ensures that every `cilk` block acts as if it has received the signal, even though the actual signal was only sent to each worker once. The amount of work done per worker to signal the exception is thus dependent only on the size of its try tree, and is independent of the depth of each worker's ready deque.

Aborting side computations

The try tree guides the aborting of side computations. When an exception is caught and propagated back to a top-level method in a ready deque, the exception is logically thrown by the child method corresponding to a leaf in that method's try tree. The leaf either represents a spawned child executing on another worker which terminated by throwing an exception, or it represents the primary program cursor on the try tree's worker and the throwing method is executing on the same worker. A lookup table produced by the compiler tells the worker how many levels up the try tree the exception is caught. If the exception will not be caught in this method, the exception propagates all the way up to the root of the try tree. This process thus determines the "catching node" of the exception in this method.

Once the catching node has been identified, we know which side computations to abort, namely, those methods that are also dynamically enclosed by the `cilk try` statement containing the catching node that handles the exception. These dynamically enclosed methods are represented by leaves in the catching node's subtree. For efficiency reasons, and because we know that all work contained in that subtree should be aborted, we signal the abort to all of those children simultaneously and asynchronously—that is, we signal them all as soon as possible, without waiting for any acknowledgment from any of the affected workers. The signaling worker also propagates the abort signal recursively through the affected children's try trees to all of their children, and their children's children, and so on. This strategy of propagation creates no semantic problems, because all of these methods operate logically in parallel, and they should all be signaled to abort eventually. Consequently, they might as well be signaled immediately. Even though the abort signal is sent to all descendants simultaneously and asynchronously, the `CilkAbort` exception is thrown semisynchronously (as JCilk's semantics guarantee), because each worker checks its abort flag only at thread boundaries.

Even after a child method has been signaled to abort, it may still return an ordinary return value or a non-`CilkAbort` exception. This case happens when the child method doesn't have a chance to check for the abort flag before it completes. Since these signaled methods should have been aborted, any value or non-`CilkAbort` exception that they returned are discarded and replaced with `CilkAbort` exceptions.

Among the side computations that must be signaled to abort is the method containing the `cilk try` statement itself if the primary program cursor still resides in the `cilk try` statement. This case requires a careful implementation, because the method containing the `cilk try` is running on the same worker that initiates the propagation of abort signals.

We use a two-step process to signal the abort. First, we spin off the closure in the worker's ready deque as if it were being stolen. The new child frame left behind is now a new closure which can be aborted in the same way that each of the other spawned children are aborted. This action leaves the parent frame (the original closure) in an unacceptable state, however: the next statement the parent executes is the statement following the spawn in the method we just spun off, which likely is still inside the aborted block. To rectify this situation, we move the try-tree cursor node and the frame's primary program cursor both to a point immediately after the catching `try` statement if it is a `cilk try` statement. If the catching node is a `cilk` method block, we instead advance the primary cursor to the end of the method. Since the method must still wait for all side computations to complete, it acts as if the execution had encountered a `sync` statement.

Executing catchlets and inlets

After all side computations have aborted, the corresponding catching block of the `catch` clause handling the exception needs to be executed as a catchlet. If the `cilk try` statement contains any `finally` block, the corresponding finallet needs to be executed as well. The semisynchronous nature of inlets, catchlets, and finallets is enforced by only executing them at `spawn` and `sync` statements using a mechanism similar to the one that checks for an abort signal. If any exception is thrown, it is propagated up the try tree, and the appropriate children are signaled to abort. The leaves representing the throwing child method and the aborted child methods are removed from the tree once they terminate. When all of a node's children in the tree have been removed, its catchlet and finallet (if any) execute as inlets and then that node is removed from the tree as well.

Ignoring exceptions

Concurrent exceptions can potentially occur when methods are executed in parallel. These methods are represented by different leaves in the try tree, possibly residing in different levels. Since each `cilk try` statement handles at most one exception, at most one exception should be selected at each node in the try tree. As we climb up the tree from a leaf that has thrown an exception up to its catching node, we examine each node we pass through. If the current node has not yet selected an exception, we store the new exception in the node and continue up the tree. On the other hand, if the node has already selected an exception earlier, we discard the new exception, preventing it from propagating further, and we continue up the tree with the earlier exception. When we reach the catching node, we perform the same check, again either selecting the new exception or discarding it. In general, the later exception is discarded if the current node has already picked another exception. The only time a later exception takes precedence is when it is a `CilkAbort` exception.

Handling an `Error` exception

JCilk's implementation of exceptions generally prevents the worker thread from ever catching any exceptions thrown in user code. While this behavior is desired for typical cases (a user's deliberately thrown exception should not affect a worker), we must handle a thrown `Error` exception differently. All Java exception objects describe a way in which a computation has failed. The `Error` exception is no exception. Consequently, it is propagated back up to parent methods just as any exception would be. Because the `Error` probably also describes a fatal condition which the worker itself needs to know about, however, it is also rethrown at the worker level. The `Error` then propagates all the way up through the worker thread, ultimately terminating the worker itself.

8 Performance

This section describes empirical studies that seek to understand whether JCilk's linguistic primitives for exceptions can be efficiently supported. We first evaluate overheads in our prototype JCilk-1 implementation without the exception mechanism so that we can establish a baseline against which to benchmark. We establish that JCilk-1 obtains near-perfect linear speedup on up to 16 processors for simple benchmarks, and that the overhead of `spawn` is about 30 times the cost of an ordinary Java method call. We argue that suitable optimizations could bring the `spawn` overhead down by a factor of 10 in a highly tuned implementation. We demonstrate that `cilk try` imposes only a small overhead in our prototype and that even in an highly tuned implementation, the overhead would be minimal. Finally, we present evidence that the time to abort a computation is reasonable and that it grows only slowly with the number of processors.

Experimental setup

All our measurements were taken on a Sun Fire 6800 with 16 1.2-gigahertz UltraSPARC-III processors, each with 2 gigabytes of main memory, running Solaris 9 OS (64-bit sparcv9 kernel modules). We compiled our runtime system and executed the compiled JCilk program with Java 2 Platform Standard Edition (J2SE 5.0) released by Sun Microsystems (2004). Because the JVM does not distribute newly created threads across the processors quickly, after starting up the JCilk-1 runtime system, we let the worker threads run idly for roughly a minute before starting the benchmark, so that they are properly distributed across the machine before measurements begin. In addition, we run the benchmarks twice in each execution, and take the timing measurement in the second run, in order to warm up the system and possibly take advantage of optimizations done in JVM.

Our studies use four benchmarks:

- `Queens` — the queens puzzle from Section 4 given in Figure 8.
- `CountQueens` — similar to `Queens` from Section 4, but counts the total number of safe configurations of n queens on an n-by-n chessboard, instead of just finding a single solution.
- `Fib` — computes the nth Fibonacci number using an exponential-time recursive algorithm.
- `FibTry` — semantically the same as `Fib`, but with the spawns nested within three nested `cilk try` blocks.

System performance

The “counting-queens” puzzle serves as a good benchmark to establish a baseline for overall system performance. Unlike the queens puzzle presented in Section 4, `CountQueens` does not involve speculative computing or implicit abort. Consequently, the program executes a uniform amount of work across each run given the same number n of queens to place on the n-by-n board. Our implementation of the counting-queens puzzle simply walks through all possible board configurations and increments a counter whenever a safe configuration is found. Even though the program does not utilize JCilk’s exception mechanism, the try-tree data structure is maintained throughout the execution.

Figure 12 tabulates JCilk-1’s performance when running `CountQueens` with input size $n = 15$, averaging over 20 runs. The program obtains almost perfect linear speedup up to 16 processors.[4] The last column in the figure shows the speedup

[4] The occasional superlinear speedup is presumably due to the JVM’s inconsistent perfor-

# proc P	*exec time* T_P	*speedup* T_1/T_P	*% efficiency* $(T_1/T_P)/P$
1	441.6 s	1.00	100%
2	220.1 s	2.01	100%
3	145.0 s	3.05	102%
4	109.0 s	4.05	101%
5	87.6 s	5.04	101%
6	74.2 s	5.95	99%
7	62.8 s	7.03	100%
8	55.2 s	8.00	100%
9	48.2 s	9.16	102%
10	44.3 s	9.98	100%
11	40.0 s	11.04	100%
12	36.6 s	12.08	101%
13	34.4 s	12.84	99%
14	32.2 s	13.73	98%
15	30.0 s	14.70	98%
16	28.9 s	15.29	96%

Fig. 12. The execution time in seconds of `CountQueens` with input parameter $n = 15$ when running on the JCilk-1 runtime system. The first column shows the number P of processors used during execution. The second column shows the average execution time T_P of the program on P processors. The third column gives the speedup T_1/T_P, and the last column normalizes the speedup as a fraction of the theoretical maximum-possible speedup.

as a fraction $(T_1/T_P)/P$ of perfect linear speedup. Since the program's serial elision executes in roughly $T_{\text{serial}} = 400$ seconds, the work overhead of the counting queens puzzle is $T_1/T_{\text{serial}} - 1 \approx 10\%$. (This value could be reduced by simply "coarsening" the recursion to avoid spawning near the leaves of the spawn tree, thereby lengthening the average thread length.)

Overhead for `spawn`/`return`

Our second experiment measures the work overheads inherent in a `spawn` statement and its corresponding `return` statement in our JCilk-1 prototype implementation. This study allows us to estimate what the overhead would be in a highly tuned production implementation, thereby enabling us to gauge the overhead of JCilk exception semantics more realistically. To estimate the `spawn`/`return` overhead of our prototype implementation, we benchmarked the Fibonacci program `Fib` on one processor against the execution time of its corresponding Java serial elision.

The JCilk program for the `Fib` code uses an exponential time algorithm: with input parameter `n`, it recursively spawns itself with `n-1` and `n-2` and then sums the values returned by the two spawned methods. A `sync` statement before the summing

mance model.

variant	*spawn time*	*spawn overhead*	*cumulative saving*	*incremental saving*
`Fib0`	191.37 ns	31.1	0.0%	0.0%
`Fib1`	71.31 ns	11.6	62.7%	62.7%
`Fib2`	54.08 ns	8.8	71.7%	9.0%
`Fib3`	48.17 ns	7.8	74.8%	3.1%
`Fib4`	17.45 ns	2.8	90.9%	16.1%
`Fib5`	6.41 ns	1.0	96.7%	5.8%

Fig. 13. Breakdown of overheads for `Fib` running on one processor averaged over 20 runs. The first column labels the benchmark. The second column shows the time per `spawn` call in nanoseconds. The third column shows the `spawn`/`return` overhead compared to an ordinary Java method call, which we measured as about 6.2 nanoseconds. The last column shows the percentage savings compared to the unmodified `Fib0` code.

of the return values ensures that the spawned methods have terminated properly. The `Fib` code makes a good benchmark for estimating the `spawn`/`return` overhead, because it spawns recursively and performs no real computation besides one addition. The `sync` statement does not contribute to the work overhead, because it compiles into an empty statement in the fast clone. Therefore, the work overhead benchmarked by `Fib` mainly comes from its `spawn` and corresponding `return` statements.

Our methodology for obtaining a breakdown of overheads was as follows. First, we timed the execution of the generated GoJava postsource that resulted from compiling the `Fib` code. Then, we took away a portion of the GoJava code responsible for a particular overhead and timed the execution of the resulting program. We attributed the decrease in execution time to the removed GoJava code. We repeated this process, removing one additional overhead, and then a third, and the last, until only a “bare-bones” version of `Fib` remained.

Specifically, we benchmarked the following variants on `Fib`:

- `Fib0` — the GoJava postsource produced by compiling the original JCilk `Fib` program.
- `Fib1` — produced from the `Fib0` postsource by removing the synchronization overhead for the work-stealing protocol.
- `Fib2` — produced from `Fib1` by removing the compiler-generated Java `try` statements for intercepting potential exceptions thrown by spawned calls.
- `Fib3` — produced from `Fib2` by removing the state saving of a method before a `spawn`.
- `Fib4` — produced from `Fib3` by removing the compiler-generated calls to the runtime system to push and pop spawned activation frames.
- `Fib5` — produced from `Fib4` by eliminating the code that allocates and frees memory for spawned activation frames.

Figure 13 shows a breakdown of JCilk-1’s overheads for `Fib` on the Sun Fire 6800. To sanity-check these numbers, we individually removed one particular overhead (synchronization, generated `try` statements, and state saving) from `Fib0` without removing the other two. Then, we summed up the times required by the individual overheads together with the execution time of `Fib3` and compared the resulting time to the execution time of the unmodified `Fib0` code. The two times differed by less than 10%.

Next, we verified that the difference between `Fib4` and `Fib5` corresponds to the overhead saved from removing the frame allocations. Since we could not completely remove the code for allocating activation frames without removing code for pushing and popping the frames, we opted to allocate one static frame which is reused throughout execution. Even though this modification is not quite the same as removing the frame allocations completely, it roughly simulates the same effect by preventing new allocations. We compared the time saved by using a single static frame with the time difference between `Fib4` and `Fib5`. The two times differed by approximately 10%.

Finally, we compared the execution times between `Fib3` and `Fib4`, attributing the decrease in time to the removal of the calls to the runtime system for pushing and popping activation frames. We could not easily verify this overhead calculation as we did before, because the synchronization code is initiated by push and pop calls to the runtime system. Since our measurements of the other overheads substantiate our methodology, however, we feel confident in the numbers. Moreover, the `Fib5` code has a work overhead of 1.04 times that of its serial elision, which seems consistent.

As shown in Figure 13, the `spawn`/`return` overhead in our prototype JCilk-1 implementation is just over 30 times that of the serial elision. The overhead for memory management is the second lowest of the five measured overheads. It would have been far higher had we not circumvented Java's memory manager by implementing our own type-specific memory pool to recycle activation frames throughout the execution.

Compared to our JCilk-1 implementation, a production-quality implementation of JCilk would significantly reduce all of these overheads. An ideal implementation would build the JCilk primitives directly into a JVM, instead of building on top of a JVM, as we mentioned in Section 6. A large part of these overheads stem from duplicating work already done by the JVM. Specifically, JCilk-1's ready deque shadows the JVM stack but allows work-stealing to occur. If the JCilk runtime system were built directly into a JVM, the overhead for creating new frames when a `spawn` is executed would be nearly identical to an ordinary Java method call. In addition, the redundant state-saving done by JCilk-1 before a `spawn` could be eliminated entirely. Synchronization for the work-stealing protocol would be cheaper as well, because we could synchronize directly through memory, rather than using Java's heavy-weight synchronization variables. We would also be able to remove all the `try`-and-`catch` wrappers for checking and intercepting unexpected exceptions at every level of `spawn` calls, and instead, set an internal vari-

Construct	*time*	`cilk try` *%*
`cilk try`	2.34 ns	100%
Java `try`	0.39 ns	600%
JCilk-1 `spawn`	191.37 ns	1%
Highly tuned `spawn`	12.33 ns	19%
Java method call	6.16 ns	38%

Fig. 14. The overhead of a `cilk try` statement. The first column indicates the linguistic construct. The second column indicates the time overhead for using the construct. The third column normalizes the overhead for `cilk try` by dividing 2.34 ns by the overhead of the construct. Each measurement was obtained by averaging over 20 runs.

able to indicate which top frame an unexpected exception should be delivered to, should it occur. Although it is unrealistic to expect that we could eliminate all the overheads, it seems reasonable that a highly tuned JCilk implementation could reduce the `spawn`/`return` overhead to perhaps twice the cost of a Java method call. Experience from Cilk-5 (Frigo *et al.*, 1998) indicates that even without a direct implementation, a `spawn`/`return` overhead of 2–6 times the cost of a Java method call is achievable.

Overhead for `cilk try`

Our third experiment studied the work overhead in JCilk-1 associated with `cilk try` statements. When running on one processor, a `cilk try` statement imposes no overhead compared with a Java `try`, because `cilk try` simply compiles into an ordinary Java `try`. Running on multiple processors incurs additional overhead for `cilk try` when work is stolen (see Section 7). This overhead includes the overhead for using the ordinary Java `try` construct and the overhead for maintaining the try tree data structure in the runtime system. If the work per processor of the computation dominates its critical-path length, however, this overhead is provably small (Frigo *et al.*, 1998; Blumofe and Leiserson, 1999).

To estimate the overheads in JCilk-1 associated with the `cilk try` statements, we compared the execution times of `Fib` and `FibTry`, which does the same computation as `Fib` except with 10 additional but unnecessary nested `cilk try` statements enclosing the two `spawn` calls. We added 10 nested `cilk try` statements in `FibTry` instead of just one, because the `cilk try` overhead is relatively small compared with other overheads in JCilk, and one `cilk try` statement imposes insufficient overhead to affect the execution time significantly.

Figure 14 shows the overheads for `cilk try` and other constructs, as well as the ratio of the `cilk try` overhead to the overhead of each of the constructs. These measurements were obtained as follows. We executed `Fib` and `FibTry` on 16 processors, calculated the total work (i.e., the sum of execution times spent on all 16 processors) for both benchmarks, and attributed the increase in total work between `Fib` and `FibTry` to the additional `cilk try` statements. The overhead measurement for `cilk try` in Figure 14 was obtained by dividing the increased work by the total number of `cilk try` statements executed. We averaged the values thus obtained over 20 runs. Similarly, we obtained the ordinary Java `try` overhead by comparing the serial elisions of `Fib` and `FibTry` and then normalizing the difference appropriately.

Is the overhead of a `cilk try` statement high or low? Since a Java `try` statement is about 17% of the speed of a `cilk try` statement, the overhead of `cilk try` would appear to be considerable. Whenever a programmer uses `cilk try`, however, the `cilk try` block likely contains a `spawn` statement, and compared to a `spawn`/`return` in the current JCilk-1 implementation, the overhead of `cilk try` is negligible: only 1.2%. As we have argued, however, JCilk-1's `spawn` overhead can be improved dramatically. To make a fair comparison, the overhead of `cilk try` should be compared to a more-realistic value for `spawn`/`return` in a highly tuned, production-quality implementation. To this end, assume that the `spawn`/`return` overhead can be reduced to twice the cost of an ordinary Java method call. Under this assumption, a `spawn` would cost 12.33 ns instead of the almost 200 ns as in JCilk-1. The cost of `cilk try` would be 19% of this highly tuned `spawn`/`return`, and about 38% of a Java method call itself. Since the real work of programs generally dominates `spawn`/`return`, we can expect that the `cilk try` overhead would rarely be seen in realistic applications.

Abort time

Our final experiment examines the time it takes to abort a computation. Figure 15 shows results of our study which measured the abort time in milliseconds when running `Queens` with input size $n = 28$. We measured the abort time as follows. The timing starts at the point immediately before the throw of the `Result` exception in line 6 of Figure 8 that triggers the implicit abort of the appropriate side computations. The timing stops when the exception is caught in line 28 of Figure 8 and all appropriate side computations have terminated.

The clustering of data points in Figure 15 indicates that the abort process typically completes in under 10 milliseconds and grows slowly as the number of processors increases. We compare this value with its corresponding Java elision with the same input size, $n = 28$. The exception propagation time (from the point of throw to the point of catch) in the Java elision is approximately 0.2 milliseconds. The abort time on a single processor is roughly 0.4–0.5 milliseconds, which is about 2–3 times the cost of the Java elision. The abort time for multiple processors falls in the range between 5 and 10 milliseconds, growing slowly as the number of processors increases. Since a speculative computation such as `Queens` can run for an arbitrarily long time if it is not aborted, we view 10 milliseconds as acceptable performance.

In order to understand the abort overhead, we inserted code into the Java elision so

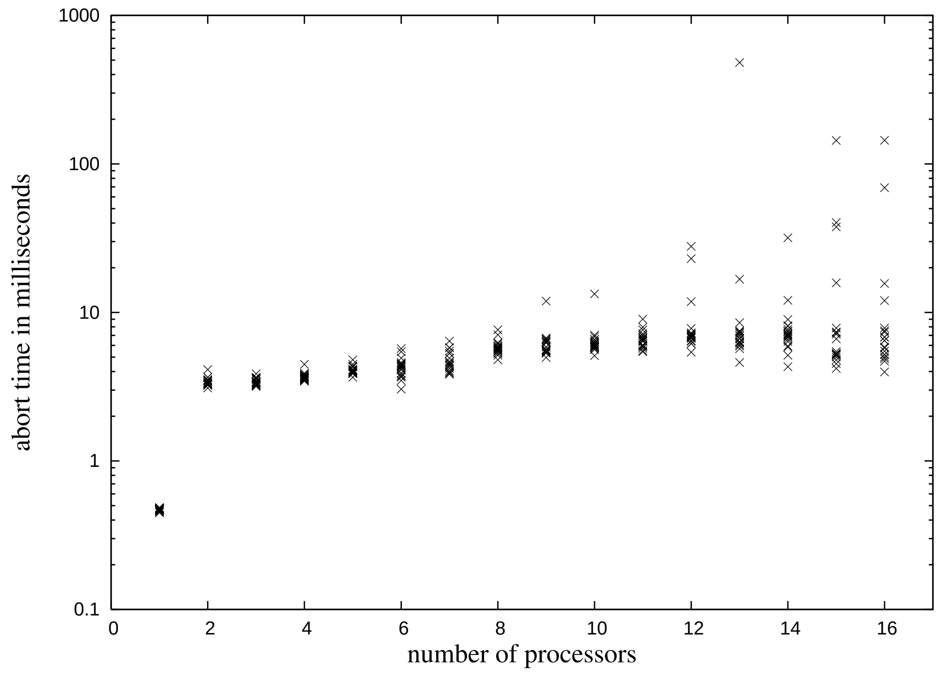

Fig. 15. The time to abort the `Queens` computation with input parameter $n = 28$ as a function of number of processors. For each number of processors, the program was run 20 times and the individual times are plotted.

that the exception is caught and rethrown at each level of recursion, thereby simulating behavior similar to an abort process. We discovered that Java's performance varied greatly depending on the contents of the `catch` clause, because Java employs "just-in-time" (JIT) compilation technology when it encounters code that is frequently executed. For instance, by inserting a loop incrementing some variable for 1000 times into the catch handler (call this Case 1), the exception propagation time costs less than 2 milliseconds. By inserting a function call in the catch handler (call this Case 2), where the function does the same loop with increment operations, the exception propagation time increases up to ≈ 7.5 milliseconds. On the other hand, if the code has been "warmed up" by executing it once in advance before taking measurements, Case 1 still has ≈ 2 milliseconds of exception propagation time, while Case 2 now has only ≈ 0.12 milliseconds propagation time. After warming up, the code from Case 2 is optimized, but not the code from Case 1.

These studies lead us to conjecture that a large part of the abort overhead comes from propagating exceptions between stack frames. The calls to the JCilk-1 runtime library are apparently too complex to allow Java's JIT technology to optimize them. We speculate that a more-direct implementation of JCilk in the JVM would yield an abort time of 0.2 milliseconds or better. A better implementation of Java's exception mechanism would likely improve the abort time even further.

The data set contains several outliers, which tend to increase in frequency as the number of processors grows. We suspect that these outliers are caused by other processes in the operating system or in the JVM during program execution. Since the abort time is extremely short, it is sensitive to any interference. Should a worker involved in an abort be context-switched out when an abort occurs, its temporary inability to participate in the abort process could significantly delay the abort.

Since the abort signal is detected only at thread boundaries, the length of the abort process is correlated to the average thread length in the JCilk program. If the performance of aborting is critical in an application with long threads, JCilk's `yield` statement[5] provides a way for the user to manually shorten threads. A `yield` in a piece of code indicates that a thread boundary should be inserted at that point. If a program contains a long piece of pure Java code, breaking it into smaller threads using `yield` allows the JCilk-1 system to check for abort more frequently.

In summary, although JCilk-1 is only a prototype implementation, our performance studies indicate that it performs well enough for many applications and that further engineering should allow it to perform even better. In particular, JCilk-1's exception mechanism, which involves the implicit aborting of side computations, is competitive with Java's unoptimized native mechanism, and it could be improved to run as fast as Java's optimized mechanism.

9 Related work

This section places JCilk and its exception-handling semantics into the context of other research in parallel programming languages. A key difference between JCilk and most 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 (Ackerman and Dennis, 1979), SISAL (Gaudiot *et al.*, 1997), Id (Nikhil, 1991), parallel Haskell (Nikhil *et al.*, 1995; Aditya *et al.*, 1995), MultiLisp (Halstead, 1985), and NESL (Blelloch, 1993) and none of the parallel imperative languages Fortran 90 (Adams *et al.*, 1992), High Performance Fortran (Richardson, 1996) (Merlin and Chapman, 1997), Declarative Ada (Thornley, 1993, 1995), C* (Hatcher *et al.*, 1991a), Dataparallel C (Hatcher *et al.*, 1991b), Split-C (Culler *et al.*, 1993), and Cilk (SuperTech, 2001) contain exception-handling mechanisms. The reason for this omission is simple: these languages were derived from serial languages that lacked such linguistics.[6]

[5] The `yield` statement is currently not implemented in JCilk-1.

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

Some parallel languages do provide exception support, because they are built upon languages that support exception handling under serial semantics. These languages include Mentat (Grimshaw, 1993), which is based on C++; OpenMP (OpenMP, 2002), which provides a set of compiler directives and library functions compatible with C++; and Java Fork/Join Framework (Lea, 2000), 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 (Tazuneki and Yoshida, 2000) and Issarny (Issarny, 1991) 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 (McLaughlin and Flanagan, 2004), provides call-return semantics for threads similar on the surface to JCilk. In particular, Java 5.0 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 5.0 is that JCilk's parallel semantics for exceptions faithfully extend Java's serial semantics. Although Java 5.0'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.

10 Conclusion

CLU (Liskov and Snyder, 1979) was the first language to cleanly define and implement 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.

Because of the semantic richness of exception linguistics, both in serial and parallel programming, we believe that exceptions should be supported efficiently. Some programmers view exceptions as occurring relatively infrequently, and hence implementations of exception mechanisms, including many current JVM implementations, tend to be slow. Today's JVM's tend to use the "handler table" method (Atkinson *et al.*, 1978) used by CLU, which assumes that exceptions occur rarely. The alternative "branch table" method (Atkinson *et al.*, 1978) provides much faster exception handling, but the designers of CLU rejected this implementation because it increases the cost of a normal return. Back in the 1970's, when CLU was designed, this overhead was perhaps substantial, but today the cost of the extra register operations on overall runtime should be negligible. We believe that JVM implementers should reconsider using branch tables or related linkage mechanisms to make exceptional returns cost much the same as an ordinary method returns. There should be no performance penalty for programming elegantly with exceptions, either for serial computing or parallel computing.

Acknowledgments

Many thanks to Scott Ananian and Bradley Kuszmaul of MIT CSAIL for their copious comments. Thanks also to Bradley Kuszmaul of MIT CSAIL and Christine Flood and Dave Dice of Sun Microsystems Research for their help in understanding and debugging performance problems. Kunal Agrawal, Jeremy Fineman, Viktor Kuncak, Martin Rinard, Gideon Stupp, and Jim Sukha of MIT CSAIL and Wong Weng Fai of National University of Singapore engaged us in many helpful discussions. We gratefully acknowledge the generosity of the University of Rochester for access to their Sun Fire 6800 for our empirical studies.

References

Ackerman, W., Dennis, J. B., 1979. VAL — A value oriented algorithmic language. Tech. Rep. TR-218, Massachusetts Institute of Technology Laboratory for Computer Science.

Adams, J., Brainerd, W., Martin, J., Smith, B., Wagener, J., 1992. Fortran 90 Handbook. McGraw-Hill.

Aditya, S., Arvind, Maessen, J.-W., Augustsson, L., Nikhil, R. S., June 1995. Semantics of pH: A parallel dialect of Haskell. In: Hudak, P. (Ed.), Proc. Haskell Workshop, La Jolla, CA USA. pp. 35–49.
URL `citeseer.ist.psu.edu/aditya95semantics.html`

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

Atkinson, R. R., Liskov, B. H., Scheifler, R. W., 1978. Aspects of implementing CLU. In: ACM 78: Proceedings of the 1978 Annual Conference. ACM Press, New York, NY, USA, pp. 123–129.

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

Blumofe, R. D., Leiserson, C. E., Sep. 1999. Scheduling multithreaded computations by work stealing. Journal of the ACM 46 (5), 720–748.

Culler, D. E., Arpaci-Dusseau, A. C., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., von Eicken, T., Yelick, K. A., 1993. Parallel programming in Split-C. In: Supercomputing ’93. IEEE Computer Society, pp. 262–273.

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

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

Frigo, M., Leiserson, C. E., Randall, K. H., 1998. The implementation of the Cilk-5 multithreaded language. In: PLDI ’98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. ACM Press, New York, NY, USA, pp. 212–223.

Gaudiot, J.-L., DeBoni, T., Feo, J., BöHm, W., Najjar, W., Miller, P., 1997. The Sisal model of functional programming and its implementation. In: PAS ’97: Proceedings of the 2nd

AIZU International Symposium on Parallel Algorithms / Architecture Synthesis. IEEE Computer Society, p. 112.

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

Gontmakher, A., Schuster, A., 2000. Java consistency: Nonoperational characterizations for Java memory behavior. ACM Trans. Comput. Syst. 18 (4), 333–386.

Gosling, J., Joy, B., Steele, G., Bracha, G., 2000. The Java Language Specification, 2nd Edition. Addison-Wesley, Boston, Massachusetts.

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

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

Hatcher, P. J., Lapadula, A. J., Jones, R. R., Quinn, M. J., Anderson, R. J., 1991a. 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. ACM Press, pp. 73–82.

Hatcher, P. J., Quinn, M. J., Lapadula, A. J., Anderson, R. J., Jones, R. R., 1991b. Dataparallel C: A SIMD programming language for multicomputers. In: Sixth Distributed Memory Computing Conference. IEEE Computer Society, pp. 91–98.

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

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

Kernighan, B. W., Ritchie, D. M., 1988. The C Programming Language, 2nd Edition. Prentice Hall, Inc.

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

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

Lea, D., 1999. Concurrent Programming in Java: Design Principles and Patterns, 2nd Edition. Addison-Wesley, Boston, Massachusetts.

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

Lindholm, T., Yellin, F., 2000. The Java Virtual Machine Specification, 2nd Edition. Addison-Wesley, Boston, Massachusetts.

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

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

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

Merlin, J., Chapman, B., 1997. High Performance Fortran. URL `citeseer.ist.psu.edu/merlin97high.html`

Netzer, R. H. B., Miller, B. P., March 1992. What are race conditions? ACM Letters on

Programming Languages and Systems 1 (1), 74–88.

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

Nikhil, R. S., Arvind, Hicks, J. E., Aditya, S., Augustsson, L., Maessen, J.-W., Zhou, Y., 1995. pH Language Reference Manual, Version 1.0. Computation Structures Group, Massachusetts Institute of Technology Laboratory for Computer Science, technical Memo CSG-Memo-369.
URL `citeseer.ist.psu.edu/nikhil95ph.html`

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

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

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

Richardson, H., 1996. High Performance Fortran: history, overview and current developments.
URL `citeseer.ist.psu.edu/richardson96high.html`

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

Sun Microsystems, 2004. Java 2 platform standard edition 5.0.
URL `http://java.sun.com/j2se/1.5.0/`

SuperTech, Nov. 2001. Cilk 5.3.2 Reference Manual. Supercomputing Technologies Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139.
URL `http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf`

Tazuneki, S., Yoshida, T., 2000. 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. IEEE Computer Society, Washington, DC, USA, p. 75.

Thornley, J., Apr. 1993. The Programming Language Declarative Ada Reference Manual. Computer Science Department, California Institute of Technology.
URL `http://caltechcstr.library.caltech.edu/211/`

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

Winston, P. H., 1992. Artificial Intelligence, 3rd Edition. Addison-Wesley, Reading, Massachusetts.