Supporting Exception Handling for Futures in Java

Lingli Zhang Chandra Krintz Priya Nagpurkar
Computer Science Department
University of California, Santa Barbara
{lingli_z,ckrintz,priya}@cs.ucsb.edu

ABSTRACT

A future is a simple and elegant construct that programmers can use to identify potentially asynchronous computation and to introduce parallelism into serial programs. In its recent 5.0 release, Java provides an interface-based implementation of futures that enables users to encapsulate potentially asynchronous computation and to define their own execution engines for futures. In prior work, we have proposed an alternative model, called directive-based lazy futures (DBLFutures), to support futures in Java, that simplifies Java programmer effort and improves performance and scalability of future-based applications. In the DBLFuture model, programmers use a new directive, "*@future*", to specify potentially concurrent computations within a serial program. DBLFutures enable programmers to focus on the logic and correctness of a program in the serial version first, and then to introduce parallelism gradually and intuitively. Moreover, DBLFutures provide greater flexibility to the Java virtual machine for efficient future support.

In this paper, we investigate the exception handling aspect of futures in Java. In Java 5.0 Future APIs, exceptions of future execution are propagated to the point in the program at which future values are queried (used). We show that this exception handling model is not appropriate or desirable for DBLFutures. Instead, we propose an *as-if-serial* exception handling mechanism for DBLFutures in which the system delivers exceptions at the same point as they would be delivered if the program was executed sequentially. Our approach, we believe, provides programmers with intuitive exception handling behavior and control. We present the design and implementation of our approach within the DBLFuture framework in the Jikes Research Virtual Machine. Our results show that our implementation introduces negligible overhead for applications without exceptions, and guarantees serial semantics of exception handling for applications that throw exceptions.

PPPJ 2007, September 5–7, 2007, Lisboa, Portugal.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures, Control structures*

General Terms

Language, Performance, Reliability

Keywords

exception handling, concurrent programming, Java, future

1. INTRODUCTION

An exception handling mechanism is a language control structure that allows programmers to specify the behavior of the program when an exceptional (unusual) event occurs [9]. Exception handling is key for software fault tolerance and enables developers to produce reliable, robust software systems. Many languages support exception handling as an essential part of the language design, including CLU [27], Ada95 [18], C++ [34], Java [14], Eiffel [29], and others.

As multi-processor computer systems become increasingly popular, many parallel programming languages or constructs [8, 2, 23, 26] have been proposed to enable programmers to express potential parallelism in programs easily so that the extra computation resources could be exploited. It is important to extend the exception handling mechanism to the concurrent context for fault tolerance and error recovery. However, exception handling semantics in a concurrent system are much more complex than for a serial environment. Their implementation requires careful design and must be implemented efficiently.

In this work, we investigate how to support exception handling in the context of the *future* parallel programming construct in the Java programming language. A future is a simple and elegant construct that programmers can use to identify potentially asynchronous computation and to introduce parallelism into serial programs. It was first introduced in Multilisp [16], and has been supported by many languages including Java J2SE 5.0 [23] and X10 [8]. In our prior work [39], we propose a new implementation of futures in Java that we refer to as *Directive-based Lazy Futures*, and DBLFutures for short. In the DBLFuture model, programmers use a future directive, denoted as a new Java annotation, `@future`, to specify potentially concurrent computations within a serial program, and

leave the decisions of when and how to execute these computations to the Java virtual machine (JVM). The DBLFuture-aware JVM recognizes the future directive in the source and makes effective scheduling decision automatically and adaptively by exploiting its runtime services (recompilation, scheduling, allocation, performance monitoring) and its direct access to detailed, low-level, knowledge of system and program behavior that are not available at the library level.

A key design goal of DBLFutures and our work in general, is to enable programmers to develop and reason about serial programs first and then introduce parallelism gradually and intuitively. We take this approach to simplify the process of parallel programming to improve programmer productivity so that more applications can take advantage of the current and next generation of systems with multiple processing cores. In a DBLFuture program, if we elide the future annotations, the program is in its serial form. As a result, programmers write their program as if it were serial and then identify code regions that can be safely executed in parallel and capture the return value from calls to these functions using an annotated local variable. Our goal with this paper, thus, is to maintain these *as-if-serial* semantics and introduce a novel exception handling mechanism into DBLFutures.

In Java 5.0 Future APIs [23], exceptions from future execution are propagated to the point in the program at which future values are queried (used). Our *as-if-serial* exception handling mechanism delivers exceptions to the same point that they are delivered if the program is executed sequentially. In particular, an exception thrown and uncaught by a future thread will be delivered to the invocation point of the future call instead of the use point of the future value. Given this "as-if-serial" property, our approach provides programmers with more intuitive understanding of the exception handling behavior and control. We present the design and implementation of our exception handling mechanism within the DBLFuture framework on the Jikes Research Virtual Machine. Our results show that our implementation introduces negligible overhead for applications without exceptions, and guarantees serial semantics of exception handling for applications that throw exceptions.

2. BACKGROUND

We first overview the state of the art in Java Future support and implementation. We begin with the existing Future APIs in Java 5.0, and then give an overview of directive-based lazy futures, a technique that we have developed as part of prior work [39].

2.1 Java 5.0 Future APIs

Version 5.0 of the Java programming language introduces a future feature via a set of APIs in the `java.util.concurrent` package. The primary APIs include `Callable`, `Future`, and `Executor`. Figure 1 shows code snippets of these interfaces.

Using the Java 5.0 Future APIs, programmers encapsulate a potentially parallel computation in a `Callable` object and submit it to an `Executor` for execution. The Executor returns a `Future` object that the current thread can use to query the computed result later via its `get()` method. The current thread immediately executes the code right after the submitted computation (i.e., the continuation) until it invokes the `get()` method of the `Future` object, at which point it blocks until the submitted computation finishes and the result is ready. The Java 5.0 library provides several implementations of `Executor` with various scheduling strategies. Programmers can also implement their own customized Executors that meet their special scheduling requirements. We refer to this programming model as *J5Future* in this paper. Figure 2 shows a simplified program for computing the Fibonacci number (Fib) using the Java 5.0 Future interfaces.

```
public interface Callable<T>{
  T call() throws Exception;
}

public interface Future<T>{
  ...
  T get() throws InterruptedException,
           ExecutionException;
}

public interface ExecutorService extends Executor{
  ...
  <T> Future<T> submit(Callable<T> task)
    throws RejectedExecutionException,
           NullPointerException;
}
```

Figure 1: The `java.util.concurrent` futures API

```
public class Fib implements Callable<Integer>
{
  ExecutorService executor = ...;
  private int n;

  public Integer call() {
    if (n < 3) return n;
    Future<Integer> f = executor.submit(new Fib(n-1));
    int x = (new Fib(n-2)).call();
    try{
      return x + f.get();
    }catch (ExecutionException ex){
      ...
    }
  }
  ...
}
```

Figure 2: The Fibonacci program using Java 5.0 futures API

There are several drawbacks of the J5Future programming model. First, given that the model is based on interfaces, it is non-trivial to convert serial versions of programs to parallel versions since programmers must reorganize the programs to match the provided interfaces, e.g., wrapping potentially asynchronous computations into objects. Secondly, the multiple levels of encapsulation of this model results in significant, but unnecessary, memory consumption which can degrade performance significantly due to the extra memory management overhead. Finally, to achieve high-performance and scalability, it is vital for a future implementation to make effective scheduling decisions, e.g., to spawn futures only when the overhead of parallel execution can be amortized by doing so. Such decisions must consider both the granularity of computation and the underlying resource availability. However, in the J5Future model, the scheduling components (Executors) are implemented at the library level, i.e., outside and independent of the runtime. As a result, these components are unable to acquire accurate information about either computation granularity or underlying resource availability that is necessary to make good scheduling decisions. Poor scheduling decisions can severely degrade performance and scalability, especially for applications with fine-grained parallelism.

2.2 Overview of DBLFutures

To simplify programmer effort and to improve performance and scalability of future-based applications in Java, in prior work [39], we propose a new implementation of futures in Java that we refer to as *Directive-based Lazy Futures* (DBLFutures).

The syntax of DBLFutures is very simple. We introduce a future directive, denoted as a new Java annotation, `@future`, for Java source code. Programmers use this directive to annotate local variables that are placeholders of results that are returned by calls to potentially concurrent functions. If a function call stores its return value to an annotated local variable, the DBLFuture system identifies the call as an invocation of a future.

Figure 3 shows the much simpler version of *Fib* that uses this model. Note that using DBLFutures, *the parallel version of a program is the same syntactically as its serial version except for future annotations* on a subset of local variable declarations (the goal of our work). Programmers using DBLFutures can focus their efforts on the logic and correctness of a program in the serial version first, and then introduce parallelism to the program gradually and intuitively. Note that such a methodology is not appropriate for all concurrent programs or for expert parallel programmers. It is, however, a methodology that enables novices to take advantage of available processing cores efficiently and with little effort. In this work, we target at the programs that developers produce using this methodology.

```
public class Fib
{
  public int fib(int n) {
    if (n < 3) return n;
    @future int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
  }
  ...
}
```

Figure 3: The Fibonacci program using DBLFutures

To exploit the services and knowledge of the execution environment that are not available at the library level, and to relieve programmers of the burden of future scheduling, we implemented DBLFuture as an extension to the Java Virtual Machine (JVM). We provide a complete description of the DBLFuture implementation in [39] and [40]. In summary, our DBLFuture system facilitates:

- **Lazy spawning**. Similar to Mohr et al. [30], our system always treats a future function call the same as a normal call initially, i.e., executes it on the current thread's stack. The continuation of the future will be spawned to a new thread only if the system decides that it's beneficial to do so based on runtime information. The laziness is the key to efficiently support fine-grained futures. Note, however, that the scope of a future call is the method in which it is implemented. That is, the continuation is the execution of the current method following the future call up to the return of the method.

- **Low-overhead monitoring system**. Our system leverages the sampling system that is common to JVMs for support of adaptive optimization, to extract accurate and low-level program (e.g. long running methods) and system information (e.g. number of available processors) with low overhead. Such information is the key for effective scheduling decisions.

- **Volunteer stack splitting**. As opposed to the commonly used work-stealing approach [30, 12], a thread in our DBLFuture system voluntarily splits its stack and spawns its continuation using a new thread. The system performs such splits at thread-switch points (method entries and loop backedges), when the monitoring system identifies an unspawned future call as long-running ("hot" in adaptive-optimization terms). The current thread (to which we refer to as the future thread) continues to execute the future and its method invocations as before; the system creates a new continuation thread, copies all the frames prior to the future frame from the future thread's stack to the stack of the new thread. The system then initiates concurrent execution of the new thread, as if the future call had returned. The system also changes the return address of the future call to a special stub, which stores the return value appropriately for use (and potential synchronization) when the future call returns on the future thread. With the volunteer stack splitting mechanism, we avoid the synchronization overhead incurred by work-stealing, which can be significant in a Java system [10].

- **Compatibility with the Java thread system**. Instead of using the popular worker-crew approach, i.e., a fixed number of special workers execute queued futures one by one, our system executes futures directly on the current Java execution stack and relies on stack splitting to generate extra parallelism. Each volunteer stack split results in two regular Java threads, the future and the continuation, both of which are then handed to the internal JVM threading system for efficient scheduling. This feature makes futures compatible with Java threads and other parallel constructs in our system.

- **Low memory consumption**. In the J5Future model, the system must always generate wrapper objects (`Future`, `Callable`, etc.) even if the future is not spawned to another thread since the object creation is hard-coded in the program. In contrast, the simple and annotation-based specification of DBLFutures provide greater flexibility to the JVM. In our system, the JVM treats annotated local variables as normal local variables until the corresponding future is split, at which point a `Future` object is created and replaces the original local variable adaptively. We thus, avoid object creation which translates into significant performance gains.

In summary, our DBLFuture implementation is very efficient and scalable since it exploits the powerful adaptivity of the JVM that is enabled by its runtime services (recompilation, scheduling, allocation, performance monitoring) and its direct access to detailed, low-level, knowledge of system and program behavior.

3. AS-IF-SERIAL EXCEPTION HANDLING

A key feature of the Java programming language is its exception handling mechanism that enables robust and reliable program execution and control. In this work, we consider how to implement this feature in coordination with Java futures. Our goal is to identify a design that is both compatible with the original language

design and that preserves our as-if-serial program implementation methodology. In our prior work [39], we focused on the efficient implementation aspect of DBLFutures without regard for exception handling. In this section, we describe how we can support exception handling in the DBLFuture system.

One way to support exception handling for futures is to propagate exceptions to the *use point* of future return values, as is done in the J5Future model. Using the Java 5.0 Future APIs, the `get()` method of the `Future` interface can throw an exception with type `ExecutionException` (Figure 1). If an exception is thrown and not caught during the execution of the submitted future, the Executor intercepts the thrown exception, wraps the exception in an `ExecutionException` object, and saves it within the `Future` object. When the continuation queries the returned value of the submitted future via the `get()` method of the `Future` object, the method throws an exception with type `ExcecutionException`. The continuation can then inspect the actual exception using the `Throwable.getCause()` method. Note that the class `ExecutionException` is defined as a *checked exception* [14, Sec. 11.2] [23]. Therefore, the calls to `Future.get()` are required by the Java language specification to be enclosed by a a try-catch block (unless the caller throws this exception). Without this encapsulation, the compiler raises a compiler-time error at the point of the call. Figure 2 includes the necessary try-catch block in the example.

We can apply a similar approach to support exceptions in the DBLFuture system. For the future thread, in case of exceptions, instead of storing returned value into the `Future` object that the DBLFuture system creates during stack splitting, and then terminating, we can save the thrown and uncaught exception object in the `Future` object, and then terminate the thread. The continuation thread can then extract the saved the exception at the use points of the return value (the use of the annotated variable after the future call). That is, we can propagate exceptions from the future thread to the continuation thread via the `Future` object.

One problem with this approach is that it compromises one of the most important advantages of the DBLFuture model, i.e., that programmers code and reason about the logic and correctness of applications in the serial version first, and then introduce parallelism incrementally by adding future annotations. In particular, we are introducing inconsistencies with the serial semantics when we propagate exceptions to the use-point of the future return value. We believe that by violating the as-if-serial model, we make programming futures less intuitive.

For example, we can write a simple function `f1()` that returns the sum of return values of `A()` and `B()`. The invocation of `A()` may throw an exception, in which case, we use a default value for the function. In addition, `A()` and `B()` can execute concurrently. In Figure 4 (a), we show the corresponding serial version for this function, in which the try-catch clause wraps the point where the exception *may* be thrown. Using the aforementioned future exception-handling approach in which the exceptions are received at the point of the first use of the future return value, programmers must write the function as we show in Figure 4(b). In this case, the try-catch clause wraps the use point of return value of the future. If we elide the future annotation from this program (which produces a correct serial version using DBLFutures without exception handling support), the resulting version is not a correct serial version

```
public int f1() {
  @future int x;
  try{
    x = A();
  }catch (Exception e){
    x = default;
  }
  int y = B();
  return x + y;
}
```

(a)

```
public int f1() {
  @future int x;
  x = A();
  int y = B();
  try {
    return x + y;
  }catch (Exception e){
    return default + y;
  }
}
```

(b)

Figure 4: Examples for two different approaches to exception handling for DBLFutures

```
1    public int f2() {
2      @future int x;
3      int w, y, z;
4      try{
5        w = A();
6        x = B();     // a future function call
7        y = C();
8      }catch (Exception1 e){
9        x = V1;
10     }catch (Exception2 e){
11       y = V2;
12     }
13     z = D();
14     return w + x + y + z;
15   }
```

Figure 5: A simple DBLFuture program with exceptions

of the program due to the exception handling.

To address this limitation, we propose *as-if-serial* exception semantics for DBLFutures. That is, we propose to implement exception handling in the same way as is done for serial Java programs. In particular, we deliver any uncaught exception thrown by a future function call to its caller at the invocation point of the future call. Moreover, we continue program execution as if the future call has never executed in parallel to its continuation.

We use the example in Figure 5 to illustrate our approach. We assume that the computation granularity of `B()` is large enough to warrant its parallel execution with its continuation. There are a number of ways in which execution can progress:

case 1: `A()`, `B()`, `C()`, and `D()` all finish normally, and the return value of `f2()` is `A()+B()+C()+D()`.

case 2: `A()` and `D()` finish normally, but the execution of `B()` throws an exception of type `Exception1`. In this case, we propagate the uncaught exception to the invocation point of `B()` in `f2()` at line 6, and the execution continues in `f2()` as if `B()` is invoked locally, i.e., the effect of line 5 is preserved, the control is handed to the exception handler at line 8, and the execution of line 7 is ignored regardless whether `C()` finishes normally or abruptly. Finally the execution is resumed at line 13. The return value of `f2()` is `A()+V1+0+D()`.

case 3: `A()`, `B()`, and `D()` all finish normally, but the execution of `C()` throws an exception in type `Exception2`. In this case, the uncaught exception of `C()` will not be delivered to `f2()` until `B()` finishes its execution and the system stores its return value in `x`. Following this, the system hands control to the exception handler at line 10. Finally, the system resumes execution at line 13. The return value of `f2()` is `A()+B()+V2+D()`.

Note that our current as-if-serial exception semantics for DBLFutures is as-if-serial in terms of the control flow of exception delivering. True as-if-serial semantics requires that the global side effects of parallel execution of a DBLFuture program is consistent with that of the serial execution. For example, in case 2 of the above example, any global side effects of `C()` must also be undone to restore the state to be the same as if `C()` is never executed (since semantically `C()'s` execution is ignored due to the exception thrown by `B()`). However, this side effect problem is orthogonal to the control problem of exception delivering that we address in this paper. We plan to use techniques such as transactional memory [36] to enable true as-if-serial semantics for DBLFutures as part of our future work.

4. IMPLEMENTATION

To implement exception handling for DBLFutures, we extend a DBLFuture-aware Java Virtual Machine implementation that is based on the Jikes Research Virtual Machine (JikesRVM) [22]. In this section, we detail this implementation.

4.1 Total ordering of threads

To enable as-if-serial exception handling semantics, we must track and maintain a total order on thread termination across threads that originate from the same context and execute concurrently. We define this total order as the order in which the threads would terminate if the program was executed serially. We detail how we make use of this ordering in Section 4.3.

To maintain this total order during execution, we add two new references, `futurePrev` and `futureNext`, to the virtual machine thread representation with which we link related threads in an acyclic, doubly linked list. We establish thread order at future splitting points, since future-related threads are only generated at these points. Upon a split event, we set the future thread as the predecessor of the newly created, continuation, thread since this is how the the threads are executed in the serial execution. If the future thread already has a successor, we add the new continuation thread between the future thread and its successor in the linked list.

Figure 6 gives an example of this process. Stacks in this figure grow upwards. Originally, thread T1 is executing `f()`. The future function call `A()` is initially executed on the T1's stack according to the lazy spawning principle of our system. Later, the system decides to split T1's stack and spawns a new thread T2 to execute `A()`'s continuation in parallel to `A()`. At this point, we link T1 and T2 together. Then, after T2 executes the second future function call, `B()`, long enough to trigger splitting, the system again decides to split the execution. At this point, the system creates thread T3 to execute `B()`'s continuation, and links T3 to T2 (as T2's successor).

An interesting case is if there is a future function call in `A()` (`D()` in our example) that has a computation granularity that is large enough to trigger splitting again. In this case, T1's stack is split again, the system creates a new thread, T4, to execute `D()`'s continuation. Note that we must update T2's predecessor to be T4 since, if executed sequentially, the rest of `A()` after the invocation point of `D()` is executed before `B()`.

The black lines in the figure denote the split points on the stack for each step. The shadowed area of the stack denotes the stack frames that are copied to the continuation thread. These frames

```
public int f() {
  @future int x, y;
  int z;
  try{
    x = A(); //split point 1
    y = B(); //split point 2
  }catch(Exception1 e){
    ...
  }
  z = C();
  return x + y + z;
}
```

```
public int A()
    throws Exception1{
  @future int u;
  int v;
  u = D(); //split point 3
  v = E();
  return u + v;
}
```

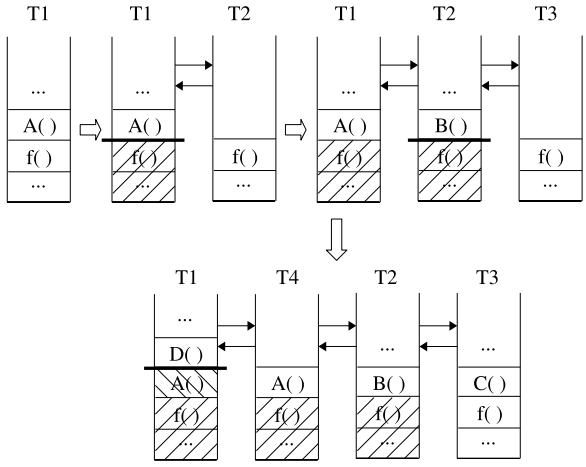

Figure 6: Example of establishing total ordering of threads.

are not reachable by the original future thread once the split occurs since the future thread terminates once it completes the future function call and saves the return value.

4.2 Choosing a Thread to Handle the Exception

One important implementation design decision is the choice of thread context in which we should handle the exception. For example, in Figure 6, if `A()` throws an exception with type `Exception1` after the first split event, we have the choice of handling the exception in T1 or T2.

Intuitively, we should choose T2 as the handling thread since it seems from the source code that after splitting, everything after the invocation point of `A()` is handed to T2 for execution, including the exception handler. T1 only has context up to the return point of `A()`, when it will store the future value and then terminate itself.

The problem is that the exception delivery mechanism in our JVM is synchronous, i.e., whenever an exception is thrown, the system searches for a handler on the current thread's stack based on the PC (program counter) of the throwing point. T2 does not have the throwing context, and will only synchronize with T1 when it uses the value of x. Thus, we must communicate the throwing context on T1 to T2 and inform T2 to pause its current execution at some point to execute the handler. This asynchronous exception delivering mechanism can be very complex to implement.

Fortunately, since our system operates on the Java stack directly and always executes the future function call on the current thread's

```
void futureStore(T value) {
  if (currentThread.futurePrev != null) {
    while (currentThread.commitStatus == UNNOTIFIED){
      wait;
    }
  } else {
    currentThread.commitStatus = READY;
  }
  Future f = getFutureObject();
  if (currentThread.commitStatus == ABORTED){
    currentThread.futureNext.commitStatus = ABORTED;
    f.notifyAbort();
    cleanup and terminate currentThread;
  } else {
    currentThread.futureNext.commitStatus = READY;
    f.setValue(value);
    f.notifyReady();
    terminate currentThread;
  }
}
```

Figure 7: Algorithm for the future value storing point

```
T futureLoad() {
  Future f = getFutureObject();
  while (!f.isReady() &&
          !currentThread.commitStatus == ABORTED){
    wait;
  }
  if (currentThread.commitStatus == ABORTED){
    if (currentThread.futureNext != null) {
      currentThread.futureNext.commitStatus
                                      = ABORTED;
    }
    cleanup and terminate currentThread;
  } else {
    return f.getValue();
  }
}
```

Figure 8: Algorithm for the future return value use point

stack, and spawns the continuation, we have a much simpler implementation option. Note that the shadowed area on T1's stack after the first split event is logically not reachable by T1. Physically, however, these frames are still on T1's stack. As a result, we can simply *undo* the splitting as if the splitting never happened via clearing the split flag of the first shadowed stack frame (the caller of A() before splitting), which makes the stack reachable by T1 again. Then, the exception can be handled on T1's context normally using the existing synchronous exception delivering mechanism of the JVM.

This observation significantly simplifies our implementation. Now, T2 and all threads that originate from T2 can be aborted as if they were never generated. If some of these threads have thrown an exception that is not caught within its own context, the thrown exception can also be ignored.

4.3 Enforcing Total Order on Thread Termination

In section 4.1, we discuss the way to establish a total order across related future threads. In this section, we describe how we use this ordering to preserve as-if-serial exception semantics for DBLFutures. Note that these related threads can execute concurrently, we simply require that their termination (commit) be ordered.

First, we add a field, called `commitStatus`, to the internal thread representation of the virtual machine. This field has three possible values: `UNNOTIFIED`, `READY`, `ABORTED`. `UNNOTIFIED` is the default and initial value of this field. A thread checks its `commitStatus` at three points: (i) the future return value store point, (ii) the first future return value use point, and (iii) the exception delivery point.

Figure 7 shows the pseudocode of the algorithm that we use at the future return value store point. The pre-condition of this function is that the continuation of the current future function call is spawned on another thread, and thus, a `Future` object is already created as the placeholder that both the future and continuation thread have access to.

This function is invoked by a future thread after it finishes the future function call normally, i.e., without any exceptions. First, if the current thread has a predecessor, it waits until its predecessor finishes either normally or abruptly, at which point, the commitStatus of the current thread is changed from UNNOTIFIED to either READY or ABORTED by its predecessor. If the commitStatus is ABORTED, the current thread notifies its successor to abort. In addition, the current thread notifies the thread that is waiting for the future value to abort. The current thread then performs any necessary cleanup and terminates itself. Note that a split future thread always has a successor. If the commitStatus of the current thread is set to READY, it stores the future value in the `Future` object, and wakes up any thread waiting for the value (which may or may not be its immediate successor), and then terminates itself.

The algorithm for the future return value use point (Figure 8) is similar. This function is invoked by a thread when it attempts to use the return value of a future function call that is executed in parallel. The current thread will wait until either the future value is ready or it is informed by the system to abort. In the former case, this function simply returns the available future value. In the latter case, the current thread first informs its successor (if there is any) to abort also, and then cleans up and terminates itself.

The algorithm for the exception delivering point is somewhat more complicated. Figure 9 shows the pseudocode of the existing exception delivering process in our JVM augmented with our support to as-if-serial semantics. We omit some unrelated details for clarity. The function is a large loop that searches for an appropriate handler block on each stack frame, from the newest (most recent) to the oldest. If no handler is found on the current frame, the stack is unwound by one frame. Finally, if the function finds no handler on the entire stack, it reports the exception to the system, and terminates the current thread.

To support as-if-serial exception semantics, we make two modifications to this process. First, at the beginning of each iteration, the current thread checks whether the current stack frame is for a spawned continuation that has a split future. If so, it checks whether the current thread has already been aborted by its predecessor. In this case, instead of delivering the exception, it notifies its successor (if there is any) to abort, cleans up, and then terminates itself. Note that the system only does this checking for a spawned continuation frame. If a handler is found before reaching such a spawned continuation frame, the exception will be delivered as usual since in that case, the exception is within the current thread's local context.

The second modification is prior stack unwinding. The current thread checks if the current frame belongs to a future function call that has a spawned continuation. In this case, we must rollback the splitting decision, and reset the caller frame of the current frame

Bench-marks	#proc=1				#proc=2				#proc=4			
	Base	EH	Diff	T	Base	EH	Diff	T	Base	EH	Diff	T
AdapInt	29.36 (0.09)	27.96 (0.18)	-4.8%	-31.79	15.02 (0.25)	15.40 (0.81)	2.5%	1.97	8.47 (1.01)	8.67 (1.35)	2.4%	0.53
FFT	7.89 (0.03)	7.78 (0.03)	-1.5%	-11.49	4.92 (0.08)	5.03 (0.10)	2.2%	3.78	4.24 (0.09)	4.18 (0.10)	-1.6%	-2.33
Fib	16.47 (0.13)	17.04 (0.06)	3.5%	17.81	8.34 (0.09)	8.48 (0.06)	1.7%	5.94	4.26 (0.02)	4.33 (0.04)	1.6%	6.47
Knapsack	11.27 (0.04)	10.79 (0.03)	-4.3%	-41.78	6.36 (0.16)	6.35 (0.14)	-0.2%	-0.22	4.40 (0.19)	4.40 (0.15)	0.1%	0.07
QuickSort	8.11 (0.04)	8.01 (0.03)	-1.3%	-9.20	4.31 (0.08)	4.28 (0.04)	-0.5%	-1.07	2.52 (0.03)	2.54 (0.03)	0.9%	2.34
Raytracer	21.22 (0.09)	20.91 (0.07)	-1.4%	-12.12	11.18 (0.10)	11.28 (0.14)	0.9%	2.56	6.26 (0.07)	6.33 (0.07)	1.1%	3.27

Table 1: Overhead and scalability of the as-if-serial exception handling for DBLFutures

```
void deliverException(Exception e) {
  while (there are more frames on stack){
    if (the current frame has a split future) {
      while (currentThread.commitStatus == UNNOTIFIED){
        wait;
      }
      if (currentThread.commitStatus == ABORTED){
        if (currentThread.futureNext != null) {
          currentThread.futureNext.commitStatus = ABORTED;
        }
        cleanup and terminate currentThread;
      }
    }
    search for a handler for e in the compiled method
    on the current stack;
    if (found a handler) {
      jump to the handler and resume execution there;
      // not reachable
    }
    if (the current frame is for a future function call
          && its continuation has been spawned) {
      if (currentThread.futurePrev != null) {
        while (currentThread.commitStatus == UNNOTIFIED){
          wait;
        }
      } else {
        currentThread.commitStatus = READY;
      }
      currentThread.futureNext.commitStatus = ABORTED;
      Future f = getFutureObject();
      f.notifyAbort();
      if (currentThread.commitStatus == ABORTED){
        cleanup and terminate currentThread;
      }else{
        reset the caller frame to non-split status;
      }
    }
    unwind the stack frame;
  }
  // No appropriate catch block found
  report the exception and terminate;
}
```

Figure 9: Algorithm for the exception delivering point

to be the next frame on the local stack. This enables the system to handle the exception on the current thread's context (where the exception is thrown) as if no splitting occurred. In addition, the thread notifies its successor and any thread that is waiting for the future value to abort since the future call finishes with an exception. The thread must still needs wait for the committing notification from its predecessor (if there is any). In case for which it is aborted, it cleans up and terminates, otherwise, it reverses splitting decision and unwinds the stack.

Note that our algorithm only enforces the total termination order when a thread finishes its computation and is about to terminate, or when a thread attempts to use a value that is asynchronously computed by another thread, at which point it will be blocked anyway if the value is not ready yet. Therefore, our algorithm does not prevent threads from executing in parallel in any order, and thus, does not sacrifice the parallelism in programs.

5. PERFORMANCE EVALUATION

Although the as-if-serial exception handling semantics are very attractive for programmer productivity since it significantly simplifies the task of writing and reasoning about DBLFuture programs with exceptions, it is important that it does not introduce significant overhead. In particular, it should not slow down applications for programs that throw no exceptions. If it does so, it compromises the original intension of the DBLFuture programming model which is to introduce parallelism easily, and to achieve better performance when there are available computational resources. In this section, we provide an empirical performance evaluation of our implementation to evaluate its overhead.

Our implementation is based on the previous DBLFuture system that is an extension to the popular, open-source Jikes Research Virtual Machine (JikesRVM) [22] (x86 version 2.4.6) from IBM Research. The test machine we use is a 4-processor box (Intel Pentium 3(Xeon) xSeries 1.6GHz, 8GB RAM, Linux 2.6.9). We only report data for the adaptively optimizing JVM configuration compiler [3] (with pseudo-adaptation (PA) [4] to reduce non-determinism) since results for the non-optimizing compiler are similar.

The benchmarks that we investigate are from the benchmark suite in the Satin system [35]. Each implements varying degrees of fine-grained parallelism. At one extreme is *Fib* which computes very little but creates a very large number of potentially concurrent methods. At the other extreme is *FFT* and *Raytracer* which implement few potentially concurrent methods, each with large computation granularity. Moreover, no future threads in these benchmarks finished exceptionally. We execute each experiment 20 times and present the average performance data in Table 1.

Table 1 has three sections, each for results with 1, 2, and 4 processors, respectively. The first column of each section is the mean execution time (in seconds) for each benchmark in the DBLFuture system without exception handling support (denoted as *Base* in the table). We show the standard deviation across runs in the parentheses. The second column of each section is the mean execution time (in seconds) and standard deviation (in parentheses) in the DBLFuture system with the as-if-serial exception handling support (denoted as *EH* in the table). The third column is the percent degradation (or improvement) of the DBLFuture system with exception handling support.

To ensure that these results are statistically meaningful, we conduct the independent t-test [13] on each set of data, and present the corresponding t values in the last column of each section. For experiments with sample size 20, the t value must larger than 2.093 or

smaller than -2.093 to make the difference between Base and EH statistically significant with 95% confidence. We highlight those overhead numbers that are statistically significant in the table.

This table shows that our implementation of the as-if-serial exception handling support for DBLFutures introduces only negligible overhead for some benchmarks. The maximum percent degradation is 3.5%, which occurs for `Fib` when one processor is used. Most of the overhead numbers are less than 2%.

These results may seem counter-intuitive since we enforce a total termination order across threads to support the as-if-serial exception semantics. However, our algorithm only does so (via synchronization of threads) at points at which a thread either operates on a future value (stores or uses) or delivers an exception. Thus, our algorithm delays termination of the thread, but does not prevent it executing its computation in parallel to other threads. For a thread that attempts to use a future value, if the value is not ready, this thread will be blocked anyway. Therefore, our requirement that threads check for an aborted flag comes for free.

Moreover, half of the performance results show that our EH extensions actually improve performance (all negative numbers). This phenomenon is common in the 1-processor case especially. It is difficult for us to pinpoint the reasons for the improved performance phenomenon due to the complexity of JVMs and the nondeterminism inherent in multi-threaded applications. We suspect that our system slows down thread creation to track total ordering and by doing so, it reduces both thread switching frequency and the resource contention to improve performance.

In terms of scalability, our results do not show a relative increase in overhead when we introduce more processors. Although we only experiment with up to 4 processors, given the nature of our implementation, we believe that the overhead will continue to be low given additional processors.

In summary, our system guarantees the as-if-serial exception handling semantics for future-based applications that throw exceptions. Moreover, our implementation of these semantics introduce little overhead for applications without exceptions.

6. RELATED WORK

Many early languages that support futures (e.g. [16, 6]) do not provide concurrent exception handling mechanisms among the tasks involved. This is because these languages do not have built-in exception handling mechanisms, even for the serial case. This is also the case for many other parallel languages that originate from serial languages without exception handling support, such as Fortran 90 [11], Split-C [24], Cilk [5], etc.

For concurrent programming languages that do support exception handling, most of them focus on the exception handling mechanism within thread boundaries, but have none or limited support for concurrent exception handling. For example, for normal Java [14] threads, exceptions that are not handled locally by a thread will not be automatically propagated to other threads, instead, they are silently dropped "on-the-floor". The C++ extension Mentat [15] does not address the exception handling problem at all. In OpenMP [31], a thrown exception inside a parallel region must be caught by the same thread that threw the exception and the execution must be resumed within the same parallel region.

Most of more recent languages that adopt futures (e.g. [23, 8, 1]) do provide concurrent exception handling for futures to some extent. For example, in Java, while future values are queried via invoking `Future.get()`, an `ExecutionException` is thrown to the caller if the future computation terminates abruptly[23]. Similar exception propagation strategy is used by the Java Fork/Join Framework [25], which supports the divide-and-conquer parallel programming style in Java. In Fortress [1], the `spawn` statement is conceptually a future construct. The parent thread queries the value returned by the spawned thread via invoking its `val()` method. When a spawned thread completes exceptionally, the exception is deferred. Any invocation of `val()` then throws the deferred exception. This is similar to the J5Future model (Figure 1).

X10 [8] proposes a *rooted exception* model, that is, if activity A is the `root-of` activity B and A is suspended at a statement awaiting the termination of B, exceptions thrown in B are propagated to A at that statement while B terminates. Currently, only the `finish` statement marks code regions as a root activity. We expect that future versions of the language may soon introduce more such statements, including the `force()` method, which extracts the value of a future computation.

The primary difference between our as-if-serial exception handling model for futures and the above approaches is the point at which exceptions are propagated. In these languages, exceptions raised in the future computation that cannot be handled locally are propagated to the thread that spawns the computation when it attempts to synchronize with the spawned thread, such as using the returned value. While in our model, asynchronous exceptions are propagated to the invocation point of the future function call as if the call is executed locally. In this sense, the exception handling mechanism for the Java Remote Method Invocation model [21] is closer to our approach since the exception context where remote execution exceptions are propagated back to the caller thread is the invocation point of the remote method. However, an RMI is usually blocking while a future call is asynchronous.

JCilk [26, 10] is the one most related to our work. JCilk is a Java-based multithreaded language that enables a "Cilk-like" parallel programming model in Java. It strives to provide a faithful extension of the semantics of Java's serial exception mechanism, that is, if we elide JCilk primitives from a JCilk program, the result program is a working serial Java program. In JCilk, an exception thrown and uncaught in a spawned thread is propagated to the invocation context in the parent thread, which is same as our model.

However, there are several major differences between these two. First, JCilk does not enforce ordering among spawned threads before the same `sync` statement. If multiple spawned threads throw exceptions simultaneously, the runtime randomly picks one to handle, and aborts all other threads in the same context. In our model, even when there are several futures spawned in the same try-catch context, there is always a total ordering among them, and our system selects and handles exceptions in their serial order. In this sense, JCilk does not maintain serial semantics to the same degree as our model does. Secondly, JCilk requires a `spawn` statement surrounded by a special `cilk try` if exceptions are possible. In our DBLFuture model, normal Java `try` clause is sufficient. Finally, since JCilk is implemented at library level, it requires very complicated source level transformation, code generation, and runtime data structures to support concurrent exception correctly (e.g., `catchlet`, `finallet`, `try tree`, etc.), whereas our imple-

mentation is much simpler thanks to the direct access to Java call stacks and the stack splitting technique.

There are only a few concurrent object-oriented languages that have built-in concurrent exception handling support, e.g., DOOCE [17], Arche [19, 20], etc. DOOCE addresses the problem of handling multiple exceptions thrown concurrently in the same `try` block by extending the `catch` statement to take multiple parameters. Also, multiple `catch` blocks are allowed to associated with one `try` block. In case of exceptions, all `catch` blocks that match thrown exceptions, individually or partially, will be executed. In addition, DOOCE supports two kinds of model for the timing of acceptance and the action of exception handling: (1) waiting for all subtasks to complete, either normally or abruptly, before starting handling exceptions (using the normal `try` clause); (2) if any of the participated objects throws an exception, the exception is propagated to other objects immediately via a *notification message* (using the `try_noti` clause). In addition to the common termination model ([33], i.e., execution is resumed after the `try-catch` clause), DOOCE supports resumption via the `resume` or `retry` statement in the `catch` block, which resumes execution at the exception throwing point or the start of the `try` block.

Arche proposes a cooperation model for exception handling. In this model, there are two kinds of exceptions: *global* and *concerted*. If a process terminates exceptionally, it signals a global exception, which is propagated to other processes that communicate synchronously with it. For multiple concurrent exceptions, Arche allows programmers to define a customized *resolution function* that takes all exceptions as input parameters and returns a *concerted* exception that can be handled in the context of the calling object.

Other prior works (e.g. [32, 28, 7, 33, 38]) have focused on general models for exception handling in distributed systems. These models usually assume that processes participating in a parallel computation are organized coordinately in a structure, such as a *conversation* [32] or an *atomic action* [28]. Processes can enter such a structure asynchronously, but have to exit the structure synchronously. In case that one process throws an exception, all other processes will be informed and an appropriate handler is invoked for all participants. With regards to the problem of handling concurrently signalled exceptions, a technique, called *exception resolution* [7] is used. Multiple exceptions are resolved to a single one based on different resolution strategies, such as the exception resolution tree [7], the exception resolution graph [37], or user defined resolution functions [19].

Our exception handling mechanism for DBLFutures is different from other work in concurrent exception handling in that the intention of preserving serial semantics grants our model special properties that simplify the implementation significantly. For example, the exception resolution strategy of our model is very simple: pick the one that should occur first in the serial semantics. Also, although our model organizes involved threads in a structured way (a double linked list), one thread does not need to synchronize with all other threads in the group before exiting like the way conversation and atomic action work. Instead, threads in our system only communicate with their predecessors and successors, and exit according to a total order defined by the serial semantics of the program.

Safe Java futures are described in [36]. Their system uses object versioning and task revocation to enforce the semantic transparency of futures automatically so that programmers are freed from reasoning about the side-effects of future executions and ensuring correctness. This transaction style support is complementary to our as-if-serial exception handling model, and we plan to integrate it into our system as part of future work. Note that the authors of this work do mention that an uncaught exception thrown by the future call will be delivered to the caller at the point of invocation of the `run` method, which is similar to our as-if-serial model. However it is unclear as to how (or if) they implemented this since the authors provide no details on their design and implementation.

7. CONCLUSIONS

In this paper, we propose an *as-if-serial* exception handling mechanism for the DBLFutures. DBLFuture is a simple parallel programming extension of Java that enables programmers to use futures in Java [39]. Our as-if-serial exception handling mechanism delivers exceptions at the same point as they are delivered if the program is executed sequentially. In particular, an exception thrown and uncaught by a future thread will be delivered to the invocation point of the future call. In contrast, in the Java 5.0 implementation of futures exceptions of future execution are propagated to the point in the program at which future values are queried (used).

We show that the as-if-serial exception handling mechanism integrates easily into the DBLFuture system and preserves serial semantics so that programmers can intuitively understand the exception handling behavior and control in their parallel Java programs. With DBLFutures and as-if-serial exception handling, programmers can focus on the logic and correctness of a program in the serial version, including its exceptional behavior, and then introduce parallelism gradually and intuitively. We present the design and implementation of our exception handling mechanisms based on the DBLFuture framework in the Jikes Research Virtual Machine. Our results show that our implementation introduces negligible overhead for applications without exceptions, and guarantees serial semantics of exception handling for applications that throw exceptions.

Acknowledgments

We thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by Microsoft and NSF grants CCF-0444412 and CNS-0546737.

8. REFERENCES

[1] E. Allan, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification version 0.785. Technical report, Sun Microsystems, 2005.

[2] E. Allan, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification version 0.954. Technical report, Sun Microsystems, 2006.

[3] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

[4] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *International symposium on Memory management*, pages 143–151, 2004.

[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM Symposium on Principles and practice of parallel programming*, pages 207–216, 1995.

[6] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 95–113, 1990.

[7] R. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, 1986.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM conference on Object oriented programming systems languages and applications*, pages 519–538, 2005.

[9] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, 31(6):531–540, 1982.

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

[11] T. Ellis, I. Phillips, and T. Lahey. *Fortran 90 Programming*. Addison Wesley, 1st edition, 1994.

[12] M. Frigo, C. Leiserson, and K. Randall. The implementation of the cilk-5 multithreaded language. In *ACM conference on Programming language design and implementation*, pages 212–223, 1998.

[13] G. Hill. ACM Alg. 395: Student's T-Distribution. *Communications of the ACM*, 13(10):617–619, 1970.

[14] J. Gosling, B. Joy, G. Steel, and G. Bracha. *The Java Language Specification Second Edition*. Addison Wesley, second edition, 2000.

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

[16] R. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[17] Shin ichi Tazuneki and Takaichi Yoshida. Concurrent exception handling in a distributed object-oriented computing environment. In *International Conference on Parallel and Distributed Systems: Workshops*, page 75, Washington, DC, USA, 2000. IEEE Computer Society.

[18] Intermetrics, editor. *Information Technology - Programming Languages - Ada*. ISO/IEC 8652:1995(E), 1995.

[19] V. Issarny. An exception handling mechanism for parallel object-oriented programming: Towards reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–39, 1993.

[20] Valérie Issarny. An exception handling model for parallel programming and its verification. In *Conference on Software for citical systems*, pages 92–100, 1991.

[21] Java Remote Method Invocation Specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/.

[22] IBM Jikes Research Virtual Machine (RVM). http://www-124.ibm.com/developerworks/oss/jikesrvm.

[23] JSR166: Concurrency utilities. http://java.sun.com/j2se/1.5.0/docs/guide/concurrency.

[24] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *ACM/IEEE conference on Supercomputing*, pages 262–273, 1993.

[25] D. Lea. A java fork/join framework. In *ACM conference on Java Grande*, pages 36–43, 2000.

[26] I. Lee. The JCilk multithreaded language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, August 2005.

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

[28] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *ACM conference on Language design for reliable software*, pages 128–137, 1977.

[29] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.

[30] E. Mohr, D. A. Kranz, and Jr. R. H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, 1991.

[31] OpenMP specifications. http://www.openmp.org/specs.

[32] B. Randell. System structure for software fault tolerance. In *International conference on Reliable software*, pages 437–449, 1975.

[33] A. Romanovsky, J. Xu, and B. Randell. Exception handling and resolution in distributed object-oriented systems. In *International Conference on Distributed Computing Systems (ICDCS '96)*, page 545, Washington, DC, USA, 1996. IEEE Computer Society.

[34] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1991.

[35] R. van Nieuwpoort, J. Maassen, T. Kielmann, and H. Bal. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[36] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *ACM conference on Object oriented programming systems languages and applications*, pages 439–453, 2005.

[37] J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *International Conference on Distributed Computing Systems*, page 12, Washington, DC, USA, 1998. IEEE Computer Society.

[38] J. Xu, A. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.

[39] L. Zhang, C. Krintz, and P. Nagpurkar. Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2007.

[40] L. Zhang, C. Krintz, and S. Soman. Efficient Support of Fine-grained Futures in Java. In *International Conference on Parallel and Distributed Computing Systems (PDCS)*, 2006.