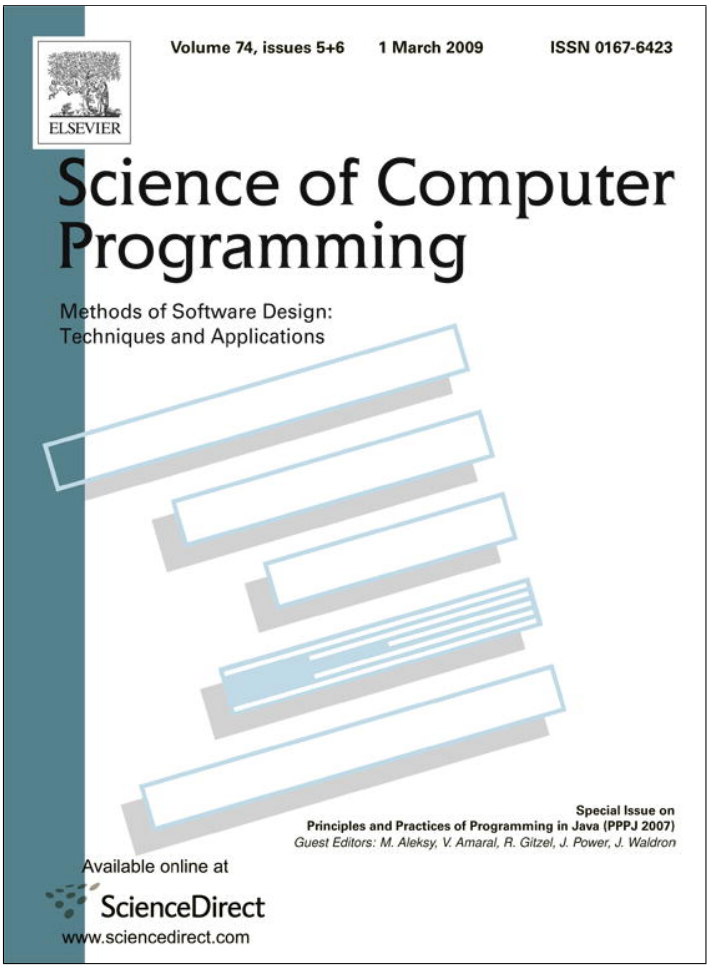

Science of Computer Programming 74 (2009) 314–332

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# As-if-serial exception handling semantics for Java futures

Lingli Zhang [a], Chandra Krintz [b,*]

[a] Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, United States

[b] Computer Science Department, University of California, Santa Barbara, CA 93106, United States

ARTICLE INFO

*Article history:*
Available online 8 February 2009

*Keywords:*
Java
Exception handling
Concurrent programming
Futures

ABSTRACT

Exception handling enables programmers to specify the behavior of a program when an exceptional event occurs at runtime. Exception handling, thus, facilitates software fault tolerance and the production of reliable and robust software systems. With the recent emergence of multi-processor systems and parallel programming constructs, techniques are needed that provide exception handling support in these environments that are intuitive and easy to use. Unfortunately, extant semantics of exception handling for concurrent settings is significantly more complex to reason about than their serial counterparts.

In this paper, we investigate a similarly intuitive semantics for exception handling for the future parallel programming construct in Java. Futures are used by programmers to identify potentially asynchronous computations and to introduce parallelism into sequential programs. The intent of futures is to provide some performance benefits through the use of method-level concurrency while maintaining *as-if-serial* semantics that novice programmers can easily understand — the semantics of a program with futures is the same as that for an equivalent serial version of the program. We extend this model to provide as-if-serial exception handling semantics. Using this model our runtime delivers exceptions to the same point it would deliver them if the program was executed sequentially. We present the design and implementation of our approach and evaluate its efficiency using an open source Java virtual machine.



## 1. Introduction

As multi-processor computer systems increase in popularity and availability, so too has the focus of the programming language community to provide support for easy extraction of parallelism from programs [8,9,2,23,26]. One language construct that has emerged from these efforts is the *future*. A future is a simple and elegant construct that programmers can use to identify potentially asynchronous computations and to introduce parallelism into serial programs. The future was first introduced by the progenitors of Multilisp [17], and is available in many modern languages including Java [23] and X10 [9].

The goal of the future is to provide developers, particularly novices, average programmers, and those most familiar with sequential program development, with a way to introduce concurrency into their serial programs incrementally, easily, and intuitively. In our recent work [40], we present the design and implementation of futures for the Java programming language that achieves this goal by providing programmers with a directive (''*@future*'') that they can use to annotate method calls that can safely be executed concurrently (i.e. those that have no side effects). Our system, called *Directive-Based Lazy Futures (DBLFutures)*, automates the spawning of such calls on separate threads so that programmers need not concern themselves

* Corresponding author.
*E-mail addresses:* Lingli.Zhang@microsoft.com (L. Zhang), ckrintz@cs.ucsb.edu (C. Krintz).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.01.006

with *when* these methods should be spawned concurrently. A DBLFutures JVM makes spawning decision adaptively by efficiently sampling both the underlying processor availability and the computation granularity of the call and only spawns potential future methods when doing so is likely to improve performance.

DBLFutures also simplifies future programming and facilitates intuitive parallel program understanding. First, a sequential version of a Java program is the same program textually as the parallel future version with annotations elided. Second, DBLFutures enforces *as-if-serial* semantics for futures. That is, the semantics of the parallel future version is the same as that of the sequential version. Our goal with this DBLFutures programming model is to ease the transition of sequential Java programmers to parallel programming by providing support for incrementally adding concurrency to Java programs.

In this paper, we investigate how to extend this as-if-serial approach to exception handling in Java programs that employ futures. An exception handling mechanism is a program control construct that enables programmers to specify the behavior of the program when an exceptional event occurs [10]. 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 [15], and Eiffel [29]. Unfortunately, many languages do not support exception handling in concurrent contexts or violate the as-if-serial property that futures provide — making exception handling verbose, difficult to use, and error prone for developers. The Java J2SE Future API [23] is one example of a future implementation that violates the as-if-serial property: exceptions from future method executions are propagated to the point in the program at which future values are queried (used), and thus require try-catch blocks to encapsulate the use of the result of a method call rather than the method call itself.

To provide as-if-serial semantics for exception handling for Java futures, we present an alternative approach to J2SE 5.0 futures. First, we employ the DBLFutures system to provide as-if-serial semantics of futures that is annotation-based and thus easier to use (with less programmer effort). DBLFutures also automates future scheduling and spawning and provides an efficient and scalable implementation of futures for Java. We extend DBLFutures with exception handling semantics that is similarly easy to use and reason about.

Our *as-if-serial* exception handling mechanism delivers exceptions to the same point at which they are delivered if the program is executed sequentially. An exception thrown and uncaught by a future thread in a DBLFutures system, will be delivered to the invocation point of the future call instead of the use point of the future value. Our approach enables programmers to use the same exception handling syntax for a concurrent, future-based version, as they would for their serial equivalent. Our system guarantees serial semantics via novel ordering and revocation techniques within the DBLFutures JVM.

In addition, we investigate the implementation of as-if-serial *side effect* semantics for DBLFutures and the implementation of exception handling semantics of futures within this context. DBLFutures alone requires that programmers only annotate methods that have no side effects, i.e., that only communicate program state changes through their return values. To relieve the programmer of this burden, i.e., to enable the programmer to annotate any method as a future, we extend DBLFutures with the *safe future* [37] functionality. Safe futures guarantee that the observable program behavior (i.e. the side effects introduced by updates to shared data by the future method or its continuation) is the same as that for the serial execution.

In our design and implementation of Safe DBLFutures (SDBLFutures), we extend this functionality with novel techniques that simplify and extend prior work on safe futures and significantly improve safe future performance through intrinsic JVM support. Our extensions also facilitate the use of arbitrary levels of future nesting and provide exception handling support for safe futures. We find that integration of as-if-serial exception handling support and side effect semantics is straightforward since the two have similar functionality and requirements.

We implement DBLFutures and SDBLFutures in the Jikes Research Virtual Machine (JikesRVM) from IBM Research and evaluate its overhead using a number of community benchmarks. Our results show that our implementation introduces negligible overhead for applications without exceptions and guarantees serial semantics for exception handling for applications that throw exceptions.

In the sections that follow, we first overview the state of the art in the design and implementation of the future construct for the Java programming language. We describe the J2SE 5.0 library-based approach to futures as well as the DBLFutures systems which we extend in this work. In Section 3, we present our as-if-serial exception handling mechanism. We detail its implementation in Section 4. We then detail our as-if-serial side effect semantics for exception handling (Section 5) and evaluate the overhead and scalability of the system (Section 6). Finally, we present related work (Section 7), and conclude (Section 8).

## 2. Background

We first overview the state of the art in Java Future support and implementation. We begin with a description of the Future APIs in Java J2SE 5.0, and then present directive-based lazy futures (DBLFutures) [40].

### 2.1. Java J2SE 5.0 Future APIs

Java J2SE 5.0 support futures in Java via a set of APIs in the `java.util.concurrent` package. The primary APIs include `Callable`, `Future`, and `Executor`. Fig. 1 shows code snippets of these interfaces.

Using the J2SE 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

```
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;
}
```

**Fig. 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){
      ...
    }
  }
  ...
}
```

**Fig. 2.** The Fibonacci program using J2SE 5.0 Future API.

the computed result later via its `get()` method. The current thread executes the code immediately 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 J2SE 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. Fig. 2 shows a simplified program for computing the Fibonacci sequence (Fib) using the J2SE 5.0 Future interfaces.

There are several drawbacks of the J2SE 5.0 Future 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 memory consumption and thus needless 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 over a sufficiently long running computation. Such decisions must consider the granularity of computation and the underlying resource availability. However, in the J2SE 5.0 Future model, the scheduling components (Executors) are implemented at the library level, i.e., outside and independent of the runtime. As a result, these components do not have access to the accurate information about computation granularity or resource availability that is necessary to make effective scheduling decisions. Poor scheduling decisions can severely degrade performance and scalability, especially for applications with fine-grained parallelism.

### 2.2. Overview of DBLFutures

To reduce programmer effort and to improve performance and scalability of futures-based applications in Java, we present a new implementation of futures in Java, called *Directive-based Lazy Futures* (DBLFutures) in [40]. In the DBLFutures model, programmers use a future directive, denoted as a new Java annotation, *@future*, to specify safe, 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 an effective scheduling decision automatically and adaptively by exploiting its runtime services (recompilation, scheduling,

```
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;
  }
  ...
}
```

**Fig. 3.** The Fibonacci program using DBLFutures.

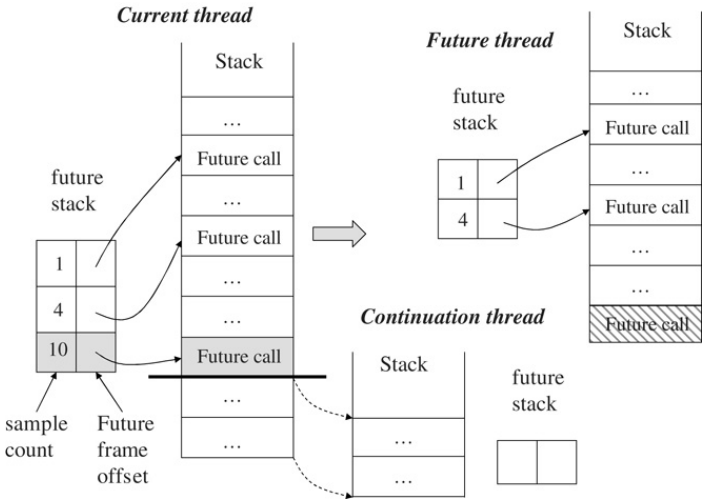


**Fig. 4.** DBLFutures creates futures lazily and spawns the continuation only when it identifies that doing so is profitable. Compiler support enables stack splitting of Java runtime stack threads, continuation spawning at the correct point in the instruction stream, and managing the return value of the future method efficiently.

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.

Programmers use the future directive to annotate local variables that are place-holders of results that are returned by calls to potentially concurrent functions. If a function call stores its return value into an annotated local variable, the DBLFuture system identifies the call as an invocation of a future.

Fig. 3 shows a version of *Fib* that uses this model. One way that DBLFutures simplifies concurrent programming is by requiring significantly fewer lines of code. The use of DBLFutures also enables programmers to develop a parallel version of a program by first developing the serial version and then adding future annotations onto a subset of local variable declarations. Programmers using DBLFutures 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 programmers who are new to concurrent programming to take advantage of available processing cores efficiently and with little effort.

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 DBLFutures as an extension to the Java Virtual Machine (JVM). We provide a complete description of the DBLFutures implementation in [40] and [42]. In summary, our DBLFutures 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., the JVM executes it sequentially (in-line), on the current thread stack. The system spawns the continuation of the future using a new thread only when it decides that it is beneficial to do so. Note, that in this implementation, the scope of a future call is the method in which it is implemented. Moreover, the continuation is the execution of the current method following the future method call up to point at which the return value from the future method is used. The laziness of DBLFutures is key to efficient and scalable execution of fine-grained futures.
- **Low-overhead performance monitoring**. Our system leverages the sampling system that is common to JVMs for support of adaptive compiler 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 facilitates effective scheduling decisions by a DBLFutures JVM.
- **Efficient stack splitting for the implementation of future spawning**. Fig. 4 overviews how a DBLFutures JVM spawns threads for continuations of future method calls. Our system maintains a shadow stack that holds metadata (sample count and reference to return value location) for each sequentially executed future on the stack. The dark line identifies

the split point on the stack. The future splitter creates a new thread (or takes one from a thread pool) for the continuation of the future call, copies the stack frames below the future frame, which corresponds to the continuation, restores the execution context from the stack frames, and resumes the continuation thread at the return address of the spawned future call. Note that we choose to create a new thread for the continuation instead of the spawned future, so that we need not set up the execution contexts for both threads. The spawned future call becomes the bottom frame of the current thread. The system deletes the future stack entry so that it is no longer treated as a potential future call.

- **Volunteer stack splitting**. As opposed to the commonly used work-stealing approach [30,13], a thread in our DBLFutures 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 back-edges), 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 [11].
- **Compatibility with the Java thread system**. Instead of using the popular worker-crew approach in which 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 J2SE 5.0 Future 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 DBLFutures implementation is 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

Exception handling in Java enables robust and reliable program execution and control. In this work, we investigate 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 [40], we focused on the efficient implementation of DBLFutures without regard for exception handling. We provide an initial investigation into the study of extending this system with as-if-serial exception handling semantics in [41]. We extend this work with additional details herein and with an investigation of its support within a DBLFutures system that provides as-if-serial side effect semantics (Section 5). In this section, we describe how we can support exception handling in the DBLFutures system.

One way to support exception handling for futures is to propagate exceptions to the *use point* of the future's return value, as is done in the J2SE 5.0 Future model. Using the Java 5.0 Future APIs, the `get()` method of the `Future` interface can throw an exception with type `ExecutionException` (Fig. 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* [15, Sec. 11.2] [23]. Therefore, the calls to `Future.get()` are required by the Java language specification to be enclosed by a try-catch block (unless the caller throws this exception). Without this encapsulation, the compiler raises a compile-time error at the point of the call. Fig. 2 includes the necessary try-catch block in the example.

We can apply a similar approach to support exceptions in the DBLFutures system. For the future thread, in case of exceptions, instead of storing the returned value into the `Future` object that the DBLFutures 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 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 DBLFutures 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

```
public int f1() {
  @future int x=0;
  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

**Fig. 5.** Examples for two different approaches to exception handling for DBLFutures.

```
1    public int f2() {
2      @future int x=0;
3      int w=0, y=0, z=0;
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   }
```

**Fig. 6.** A simple DBLFutures program with exceptions.

serial semantics when we propagate exceptions to the use-point of the future's return value. We believe that by violating the as-if-serial model, we make programming with 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 Fig. 5 (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's return value, programmers must write the function as we show in Fig. 5(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 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 Fig. 6 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 delivery. True as-if-serial semantics requires that the global side effects of parallel execution of a DBLFutures program be 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 delivery. We describe how to employ software transactions to enable as-if-serial side effect semantics

```
public int f() {
  @future int x=0, y=0;
  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;
}
```

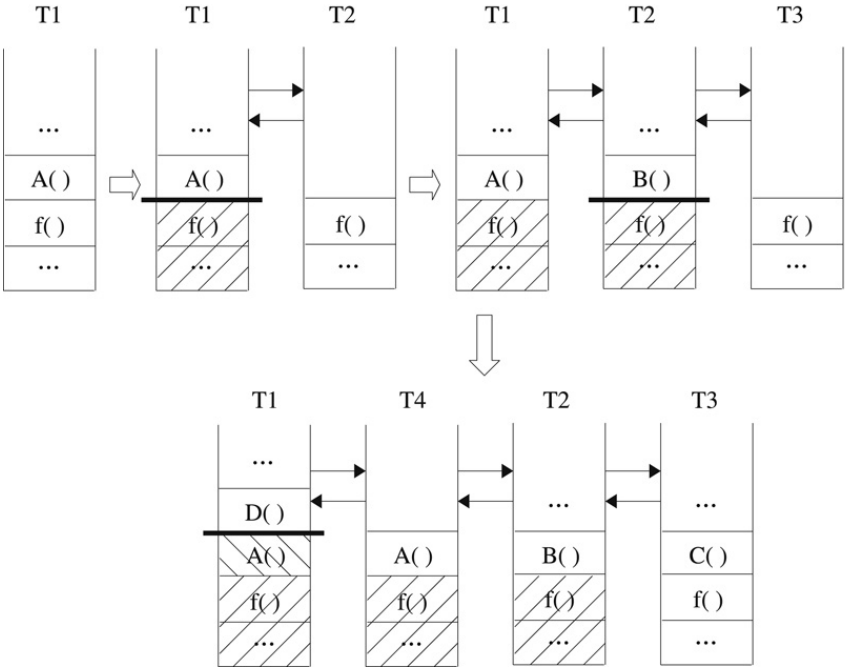


**Fig. 7.** Example of establishing a total ordering across threads.

in Section 5. Prior to this however, we present the implementation of as-if-serial exception handling without this support — that is, when we assume that the programmer only annotates variables that store the return value of side-effect-free futures.

## 4. DBLFutures implementation of exception handling semantics

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 that execute concurrently. We define this total order as the order in which the threads would terminate if the program was executed using a single processor (serially). Such ordering does not prohibit threads from executing concurrently. It simply enables our system to control the order in which threads complete, i.e. return their results (commit). As mentioned previously, we assume that futures are side-effect-free in this section. 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 predecessor of the newly created continuation thread to be the future thread, since sequentially, the future will execute and complete prior to execution of the continuation. If the future thread has a successor, we insert the new continuation thread into the linked list between the future thread and its successor.

Fig. 7 exemplifies 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 methodology of our system. Later, the system decides to split T1's stack and to spawn a new thread T2 to execute `A()`'s continuation in parallel with `A()`.

At this point, we link T1 and T2 together. Then, after T2 executes the second future function call (`B()`) for a duration long enough to trigger splitting, the system performs splitting again. At this point, the system creates thread T3 to execute `B()`'s continuation, and links T3 to T2 (as T2's successor).

One interesting case is when there is a future function call within `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 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 choice that we encounter is which thread context should handle the exception. For example, in Fig. 7, if `A()` throws an exception with type `Exception1` after the first split event, we can handle 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. The context of T1 ends just prior to the return point of `A()`, i.e., the point at which the value returned by the future is stored into the local variable.

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 request that T2 pause its current execution at some point to execute the handler. This asynchronous exception delivery mechanism is very complex to implement correctly.

Fortunately, since our system operates on the Java stack directly, always executes the future function call on the current thread's 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), to make the stack reachable by T1 again. With this approach, our system handles the exception in T1's context normally using the existing synchronous exception delivery mechanism of the JVM.

This observation significantly simplifies our implementation and is key to the efficiency of our system. In this scenario, we abort T2 and all threads that originate from T2 as if they were never generated. If any of these threads have thrown an uncaught exception, we simply ignore the exception.

### 4.3. Enforcing total order on thread termination

In Section 4.1, we discuss how 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 execute concurrently, our system only allows them to commit (return their result) in serial-execution order*.

First, we add a field, called `commitStatus`, to the internal thread representation of the virtual machine. This field has three possible values: `UNNOTIFIED`, `READY`, and `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.

Fig. 8 shows the pseudocode of the algorithm that we use at point at which the program stores the future return value. 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 place-holder to which both the future and continuation thread have access.

This function is invoked by a future thread after it completes 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 clean up 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, wakes up any thread waiting for the value (which may or may not be its immediate successor), and then terminates itself.

The algorithm we perform at the first use of the return value from a future (Fig. 9) 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 until 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 any) to abort also, and then cleans up its resources and terminates.

```
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();
    clean up and terminate currentThread;
  } else {
    currentThread.futureNext.commitStatus = READY;
    f.setValue(value);
    f.notifyReady();
    terminate currentThread;
  }
}
```

**Fig. 8.** 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;
    }
    clean up and terminate currentThread;
  } else {
    return f.getValue();
  }
}
```

**Fig. 9.** Algorithm for the future return value use point.

The algorithm for the exception delivery point is somewhat more complex. Fig. 10 shows the pseudocode of the existing exception delivery process in our JVM augmented with our support of 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 we fail to find a handler in the current frame, we unwind the stack by one frame and repeat this process until we find the handler. If we do not find the handler, we report the exception to the system and terminate the current thread.

To support as-if-serial exception semantics in DBLFutures, 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 this is the case, the thread checks whether it should abort, i.e., its predecessor has aborted. If the predecessor has aborted, the current thread also aborts and instead of delivering the exception, it notifies its successor (if there is any) that it should also abort, cleans up, and then terminates itself. Note that the system only does this checking for threads with 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 to exception delivery is to stack unwinding. The current thread checks whether the current frame belongs to a future function call that has a spawned continuation. In this case, we must roll back the splitting decision, and reset the caller frame of the current frame 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 still must wait for the committing notification from its predecessor (if there is any). In the case for which it is aborted, it cleans up and terminates, otherwise, it reverses the 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 regardless since the value is not ready yet. Therefore, our algorithm does not prevent threads from executing in parallel in any order.

```
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;
        }
        clean up 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){
        clean up 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;
}
```

**Fig. 10.** Algorithm for the exception delivery point.

## 5. Safe future support in DBLFutures

In the previous sections, we assume that the programmer determines which future methods are safe to execute concurrently, i.e., identifies methods for which there are no side effects (for which the only change in program state as a result of method execution that is visible outside of the method, is the method's return value, if any). In this section, we consider how to support as-if-serial exception handling for a DBLFutures JVM that guarantees correct (as-if-serial) execution of concurrent methods with side effects, i.e., *safe futures*, automatically. We first overview prior work on safe futures, an extension of DBLFutures that facilitates similar safety, and how as-if-serial is supported in such a system.

### 5.1. Overview of safe futures

Safe futures for Java [37] is an extension to J2SE 5.0 futures (although it does not, in general, require this programming model). Safe futures guarantee that the observable program behavior (i.e. the side effects introduced by updates to shared data by the future method or its continuation) is the same as that for the serial execution. This semantics is stronger than those for lock-based synchronization and transactional memory since they guarantee the update order of shared data (program state). Although this guarantee may limit parallelism, it provides significant programmer productivity benefits: the concurrent version is guaranteed to be correct once the programmer completes a working serial version, without requiring that the programmer debug a concurrent version.

The safe future implementation extends the J2SE 5.0 library implementation with a `SafeFuture` class that implements the J2SE 5.0 `Future` interface. To spawn a computation as a future, programmers first wrap the computation in a class that implements the `Callable` interface. At the spawning point, programmers create a `SafeFuture` object that takes the

`Callable` object as a parameter, and then call the `frun()` method of the `SafeFuture` object. Upon the invocation of the `frun()` method, the system spawns a new thread to evaluate the computation enclosed by the `SafeFuture` object. At the same time, the current thread immediately continues to execute the code immediately after the call site of the `frun()` method (i.e., the continuation), until it attempts to use the value computed by the future, when the `get()` method is invoked. The current thread blocks until the value is ready.

To preserve the as-if-serial side effect semantics, the safe future system divides the entire program execution into a sequence of *execution contexts*. Each context encapsulates a fragment of the computation that is executed by a single thread. There is a primordial context for the code that invokes the future, a future context for the future computation, and a continuation context for the continuation code. Safe futures spawn a new thread for the future context and use the thread of the primordial context (which it suspends until the future and continuation contexts complete), for the continuation context. These execution contexts are totally ordered by the system based on their logical serial execution order.

The safe future system couples each future-continuation pair, and defines two data dependency violations for each pair:

- Forward dependency violation: The continuation context does not observe the effects of an operation performed by its future context.
- Backward dependency violation: The future context does observe the effect of an operation performed by its continuation.

The system tracks every read or write to shared data between the pair of contexts via a compiler-inserted barrier that records both activities. The system checks for forward dependency violations using two bitmaps per context, one for reads and one for writes, to shared data. When a continuation context completes, the system checks the read bitmap against the write bitmap of its future context once it completes. If there is no violation, the continuation commits and ends. Otherwise, the runtime revokes the changes that the continuation has made and restarts the continuation from the beginning.

The system performs revocation via copy-on-write semantics for all writes to shared data and prevents backward dependencies using versioning of shared data. When the continuation writes to shared data, the system makes a copy of it for subsequent instructions to access. This ensures that the future does not access any shared data modified by the continuation. A context validates and commits its changes only once all contexts in its logical past have committed. If a context is revoked by the system, all contexts in its logical future that are awaiting execution are aborted (or revoked if executing, when finished).

### 5.2. Implementing safe future semantics in the DBLFutures system

The extant approach to safe futures in Java imposes a number of limitations on the JVM, and program performance. First, it employs the library approach to futures since it builds upon the J2SE 5.0 future implementation, which, as we have discussed previously, introduces significant overhead and tedious program implementation. Second, there is no support for future nesting — the safe future system assumes a linear future creation pattern, i.e., only the primordial/continuation context can create a future. This is a significant limitation since it precludes future method calls in future methods and thus, divide-and-conquer, recursive, and other parallel algorithms. Third, context management in the safe future system is implemented without compiler support (and optimization). The extant approach uses bytecode rewriting and manages revocation and re-execution through Java's exception handling mechanism. Such an approach imposes high-overhead (memory, bytecode processing, compilation), is error prone, not general, and severely limits compiler optimization and support of futures.

These restrictions (no nesting or compiler support) force the implementation of safe futures to be ad-hoc and redundant, thus imposing high overhead. For example, when a context commits, the system must compare its read and write bitmaps against all of its logical predecessors even when it has already compared predecessors against predecessors.

To address these limitations, we have designed and implemented *Safe DBLFutures (SDBLFutures)*: safe future support within the DBLFutures system. SDBLFutures extends and is significantly different from prior work on safe futures [37]. Our extensions include dependency violation tracking and prevention, execution contexts, data-access barriers, read/write bit-maps, version list maintenance from the safe future implementation, and support for exception handling semantics. Moreover, SDBLFutures provide as-if-serial side effect semantics for any level of future nesting, thereby supporting a much larger range of concurrent programs, including divide-and-conquer for fine-grained, method-level parallelism. By modifying a Java virtual machine to provide safe future support significantly simplifies the implementation of as-if-serial side effect semantics. We avoid all bytecode rewriting by associating execution context creation with stack splitting. We avoid redundant local state saving and restoring by accessing context state directly from Java thread stacks. Finally, we simplify context revocation and provide an efficient implementation for exception handling for futures.

Key to enabling arbitrary future nesting, is a novel hierarchical approach to context identifier assignment. Context identifiers (IDs) must preserve the logical order of contexts to guarantee the total serial order among, potentially concurrently, executing threads. Our context IDs are four-byte binary numbers. We use the two least-significant bits to indicate whether the value is an ID or a pointer to a more complex ID-structure (our JVM aligns objects so the last two bits are free for us to use). We use the remaining 30 bits of the value in pairs, called nesting levels, from left to right, to assign new context IDs. Each level has one of three binary values: 00, 01, and 10, which corresponds to the primordial context, the future context, and the continuation context of that nesting level, respectively. We unset all unused levels. If a program requires

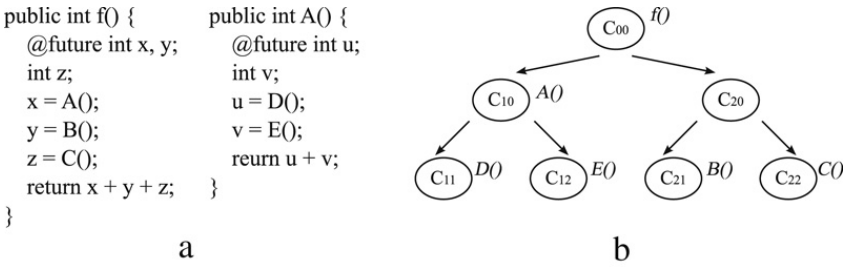


**Fig. 11.** Tree structure of execution contexts.

more than 15 levels, we encode an address in place of the ID that refers to an ID-structure that encapsulates additional layers. Except for the initial primordial (starting program) context whose ID is 0x00000001, all contexts are either a future context or a continuation context relative to a spawning point in the program. The same future or continuation context can be the primordial context of the next level, when there is nesting.

Using bits from high to low significance enables all sub-contexts spawned by a future context to have context IDs with smaller values than those of all sub-contexts spawned by the continuation context. This property is preserved for the entire computation tree using this model. Therefore, the values of context IDs are consistent with the logical order of execution contexts, which facilitates simple and fast version control.

We employ a new context organization for SDBLFutures to facilitate efficient access to contexts. This organization uses a tree structure that maintains the parent-child relationships of the nesting levels, that avoids false conflicts between contexts, and that intelligently accumulates shared state information for fast conflict checking (addressing all of the limitations of the current safe future approach).

Fig. 11 depicts a simple example of future code in Java using our context tree organization. Each node in the tree is marked with our context IDs in base-4 (with intermediate 0's removed). In this structure, the primordial context is the parent of the future and continuation contexts. The structure handles primordial context suspension in a straightforward way. The system suspends the context at the point of the future spawn and resumes it after the future and continuation complete. Thus, when the system commits a primordial context, then it has committed the entire subtree of computation below the context.

Upon a context commit, our system merges the metadata for all shared data from a child context into the parent. This merging significantly simplifies the conflict detection process and requires many fewer checks than in the original safe future implementation (which uses a linked list of contexts). Specifically, the SDBLFutures system merges (takes the intersection of) the read/write bitmaps of the child and parent contexts. In addition, for an object version created by the child context, if the parent context also has a private version for the same object, we replace that version with the child's version; otherwise, we tag the child's version with the parent's context ID and record it as an object version created by the parent context. Note that a continuation context initiates a commit only if its corresponding future context commits successfully. Therefore, our system requires no synchronization for merging.

With such information aggregation and layout, we need only check for contexts against the root of a subtree as opposed to all nodes in a subtree, when checking conflicts for contexts that occur logically after the root. In addition, our approach requires no synchronization for any tree update since all are now thread-local.

Moreover, the laziness of context creation (future spawning) in SDBLFutures avoids unnecessary context management overhead. This laziness means that the spawn point of a future will be later in time than the function entry point of the future. Any shared data access (shared with the continuation) that occurs in the future prior to spawning is guaranteed to be safe since the system executes the future prior to the continuation and sequentially. Thus, our learning delay also has the potential for avoiding conflicts and better enables commits in a way not possible in prior work (which triggers all futures eagerly).

The SDBLFutures system also handles context revocation naturally and in a much simpler way since it has direct access to the runtime stack frames of Java threads. In our model, the system stores the local state of the continuation on the future's stack even after splitting. Since we have access to both stacks, we need not duplicate work to save or restore these states. To revoke the continuation, we need only to reset the split flag of the caller frame of the future call. This causes the future call to return as a normal call, and the current thread continues to execute the code immediately following the future call — which implements the revocation of the continuation context.

### *5.3. SDBLFutures implementation of exception handling semantics*

As-if-serial exception handling semantics preserves the exception handling behavior of a serially executed application, when we execute the application with concurrent threads. To enable this, we preserve all side effects caused by computations that occur earlier in the logical (serial) order than the exception being handled and discard all side effects that occur in the logical future. We can guarantee this semantics using the as-if-serial side effect support provided by the SDBLFutures system.

As we detailed previously, we maintain a total ordering on thread termination across threads that originate from the same future spawning point and execute concurrently. In SDBLFutures, maintaining this total thread order is not necessary

```
1    void deliverException(Exception e) {
2      while (there are more frames on stack){
3        if (the current frame has a split future) {
4          // a frame for a continuation context
5          currentContext = currentThread.executionContext;
6          try to globally commit currentContext;
7          if (currentContext is aborted
                 || currentContext will be revoked){
8            clean up currentContext;
9            terminate currentThread;
10         }
11       }
12       search for a handler for e in the compiled
                method on the current stack;
13       if (found a handler) {
14         jump to the handler and resume execution there;
15         // not reachable
16       }
17       if (the current frame is for a future function call
18            && its continuation has been spawned) {
19         currentContext = currentThread.executionContext;
20         try to globally commit currentContext;
21         if (currentContext is aborted) {
22           clean up currentContext;
23           terminate currentThread;
24         }else{
25           abort continuationContext;
26           reset the caller frame to non-split status;
27         }
28       }
29       unwind the stack frame;
30     }
31     // No appropriate catch block found
32     report the exception and terminate;
33   }
```

**Fig. 12.** Algorithm for the exception delivering point within the SDBLFutures system.

since we can derive the required ordering information of threads from their associated execution contexts, which are totally ordered based on their logical serial execution order. Therefore, all extra data structures that we have introduced previously including new fields of the internal thread objects (`futurePrev`, `futureNext`, and `commitStatus`) are not needed if the JVM provides safe future support.

Without safe future support, we augment the *futureStore* and *futureLoad* algorithms in the DBLFutures system. This additional logic enables the current thread that is waiting for the previous thread in the total ordering to finish and clean up (if aborted) before performing the real actions of both functions (see Figs. 8 and 9). This augmentation is necessary to preserve the total ordering on termination of threads.

However, in SDBLFutures, this logic is already part of the algorithms of *futureStore* and *futureLoad* to preserve the as-if-serial side effect semantics, except that we use execution contexts instead of threads. This means that the extra work in *futureStore* and *futureLoad* that was required to support as-if-serial exception handling *now comes for free in the SDBLFutures system*. Moreover, the clean-up-on-abort that we perform in SDBLFutures includes removal of all private object versions that the aborted context created. Since the side effects of an execution context are kept as private object versions of that context that are not visible until the context commits, the clean up process reverts all side effects of the computation associated with the aborted context completely. Such a process is not possible without as-if-serial side effect support.

The only additional algorithm required for support of as-if-serial exception handling in the SDBLFutures system is the exception delivery algorithm. This algorithm is similar to that of the delivery process in Fig. 10. However, we implement total ordering using execution contexts instead of threads. We present this new exception delivery algorithm in Fig. 12.

Compared to the normal exception delivery algorithm in the unmodified virtual machine, this exception delivery algorithm has two extra parts. The first part is executed before searching for a handler in the compiled method of the current frame (line 3–11 in Fig. 12). This part ensures that an exception thrown by a continuation context, but that is not handled within the continuation context before it unwinds to the splitting frame, will not be handled unless the current context commits successfully.

Successfully committing the current context indicates that all side effects of the concurrently executed contexts up to this point are guaranteed to be the same as if the program is executed sequentially. Therefore, we can proceed to search for a handler in the current method as we do for serial execution. If the current context is aborted or revoked, which indicates that the current exception would not have existed if the program was executed sequentially, the current context is cleaned

**Table 1**
Benchmark characteristics.

| Bench marks | Inputs size | future# ($10^6$) | Ex. time (s) |
|---|---|---|---|
| FFT | $2^{18}$ | 0.26 | 8.27 |
| Raytracer | balls.nff | 0.27 | 20.41 |
| AdapInt | 0–250000 | 5.78 | 28.84 |
| Quicksort | $2^{24}$ | 8.38 | 8.77 |
| Knapsack | 24 | 8.47 | 11.88 |
| Fib | 40 | 102.33 | 29.09 |

up and the current exception is ignored. Note that a continuation context usually ends and commits at the usage point of the future value, but in case of exceptions, it ends and commits at the exception throwing point.

The second extra part in this algorithm (line 17–28 in Fig. 12) occurs when we are unable to find a handler in the current frame before we unwind the stack to the next frame. Similar to the first part, this part ensures that an exception thrown by a future context, but that is not handled locally even when the system unwinds the stack the stack frame of the future call, will not be handled unless the future context commits successfully. When a context aborts, we clean up the context and ignore the exception (as in the first part). If the future context successfully commits, to handle the thrown exception on the current stack as if the future call is a normal call as we describe in Section 4.2, the system resets the split flag of the caller frame to revert the stack split. The system also aborts the continuation context, which recursively aborts all contexts in the logical future of the current context. This process also reverts all side effects caused by these contexts since they would not have occurred had we executed the program sequentially to this point. Finally, the system unwinds the stack and repeats the process for the next stack frame.

In summary, supporting as-if-serial exception handling and preserving as-if-serial side effect semantics have many similar requirements and can share many common implementations. Therefore, integrating the support of as-if-serial exception handling into the SDBLFutures system is simple and straightforward. Moreover, with the underlying support for preserving as-if-serial side effects in the SDBLFutures system, the complete as-if-serial exception handling semantics, which also defines the program behavior in the presence of exceptions, is now available.

## 6. Performance evaluation

We implement DBLFutures, the safe future extensions (SDBLFutures), and as-if-serial exception handling support for both, in the popular, open-source Jikes Research Virtual Machine (JikesRVM) [22] (x86 version 2.4.6) from IBM Research. We conduct our experiments on a dedicated 4-processor box (Intel Pentium 3(Xeon) xSeries 1.6 GHz, 8 GB RAM, Linux 2.6.9) with hyper-threading enabled. We report results for 1, 2, 4, and 8 processors — 8 enabled virtually via hyper-threading. We execute each experiment 10 times and present performance data for the best-performing. We report data for the adaptively optimizing JVM configuration compiler [3] (with pseudo-adaptation [4] to reduce non-determinism).

The benchmarks that we investigate are from the community benchmark suite provided by the Satin system [36]. 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.

We present the benchmarks and their performance characteristics in Table 1. Column 2 in the table is the input size that we employ and column three is the total number of potential futures in each benchmark. In the final two columns of the table, we show the execution time in seconds for the serial (single processor) version using the J2SE 5.0 implementation. For this version, we invoke the `call()` method of each `Callable` object directly instead of submitting it to an Executor for execution.

### 6.1. DBLFutures performance

We first overview the performance benefits that we achieve using the DBLFuture approach to Java futures. Table 2 shows the speedup of DBLFutures over the state of the art library based approach. Our compiler support is implemented in the JikesRVM optimizing compiler which optimizes hot methods dynamically. We experiment with 5 recursive benchmarks and futurize all of the functions that implement the algorithms. The number of dynamic futures in these programs range from 0.26 to 8.47 million futures. We investigate the use of different numbers of 1.6 GHz processors for these programs. Our results show that DBLFutures is 1.3 to 1.6 times faster (without Fib since it is an extreme case) than the current state of the art. The benefits come from more intelligent spawning decisions and from significantly less memory management.

For Safe DBLFutures (SDBLFutures), our system introduces overhead for state tracking (not shown here) that is similar to that of the original safe future system [37]. However, since we introduce this support within DBLFutures, and the original safe future system builds upon library-based, J2SE 5.0 Future for its implementation, the gains shown are also what we achieve when we compare SDBLFutures against the original implementation of safe futures.

Table 3 presents our evaluation of the overhead and scalability of DBLFutures. $T_i$ is the execution time of the program with $i$ processors. $T_s$ for the execution time of the corresponding serial version (column 2 in seconds). In this study, we use

**Table 2**
DBLFutures performance: speedup of the directive-based approach over the library approach.

| Bench marks | Processor Count | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| AdapInt | 1.23 x | 1.18 x | 1.26 x | 1.47 x |
| FFT | 1.08 x | 1.12 x | 1.01 x | 1.00 x |
| Fib | 4.46 x | 6.64 x | 12.42 x | 18.17 x |
| Knapsack | 1.31 x | 1.57 x | 1.76 x | 1.86 x |
| QuickSort | 1.87 x | 2.10 x | 2.27 x | 2.72 x |
| Raytracer | 1.01 x | 1.01 x | 1.00 x | 1.01 x |
| Avg | 1.83 x | 2.27 x | 3.29 x | 4.37 x |
| Avg(w/o Fib) | 1.30 x | 1.40 x | 1.46 x | 1.61 x |

**Table 3**
DBLFutures performance: the overhead and scalability of DBLFutures.

| Benchmarks | $T_s/T_1$ | $T_1/T_2$ | $T_1/T_4$ | $T_1/T_8$ |
|---|---|---|---|---|
| AdapInt | 0.93 x | 1.73 x | 3.43 x | 5.24 x |
| FFT | 0.99 x | 1.60 x | 1.99 x | 1.88 x |
| Fib | 0.34 x | 1.98 x | 3.94 x | 4.02 x |
| Knapsack | 0.96 x | 1.84 x | 2.76 x | 2.58 x |
| QuickSort | 0.88 x | 1.90 x | 3.01 x | 3.44 x |
| Raytracer | 0.99 x | 1.90 x | 3.22 x | 3.84 x |
| Avg | 0.85 x | 1.83 x | 3.06 x | 3.50 x |
| Avg (w/o Fib) | 0.95 x | 1.79 x | 2.88 x | 3.40 x |

a serial version that employs our annotation-based system without splitting — i.e. it does not employ the J2SE 5.0 library implementation (due to its high overhead). It therefore is a much faster baseline that reveals the overhead that DBLFutures imposes. $T_s/T_1$ is our overhead metric.

The difference between $T_1$ (single processor) and $T_s$ reveals three sources of overhead: (1) the bookkeeping employed to maintain the shadow future stack, (2) the activities of the future profiler, controller, and compiler, and (3) the conditional processing required by DBLFutures for storing and first use of the value returned by a potential future call. The JVMs perform no splitting in either case. DBLFutures, although unoptimized introduces little overhead and extracts performance from parallelism to enable the speedups shown.

### 6.2. Performance of as-if-serial exception handling support

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 DBLFutures 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 intent of our DBLFutures 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 evaluation of our implementation to evaluate its overhead. We present our results in the tables in Fig. 13. We measure the overhead of exception handling support in the DBLFutures system.

Fig. 13 has three subtables, for 1, 2, and 4 processors, respectively. The second column of each subtable is the mean execution time (in seconds) for each benchmark in the DBLFutures system without exception handling support (denoted as *Base* in the table). We show the standard deviation across runs in the parentheses. The third column is the mean execution time (in seconds) and standard deviation (in parentheses) in the DBLFutures system with the as-if-serial exception handling support (denoted as *EH* in the table). The fourth 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 [14] 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 be larger than 2.093 or smaller than -2.093 to make the difference between Base and EH statistically significant with 95% confidence. We highlight all 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. The maximum percent degradation is 3.5%, which occurs for `Fib` when one processor is used. The overhead is in general, less than 2%.

These results may seem counterintuitive 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

| Benchs | Base | EH | Diff | T |
|---|---|---|---|---|
| AdapInt | 29.36 (0.09) | 27.96 (0.18) | -4.8% | -31.79 |
| FFT | 7.89 (0.03) | 7.78 (0.03) | -1.5% | -11.49 |
| Fib | 16.47 (0.13) | 17.04 (0.06) | 3.5% | 17.81 |
| Knapsack | 11.27 (0.04) | 10.79 (0.03) | -4.3% | -41.78 |
| QuickSort | 8.11 (0.04) | 8.01 (0.03) | -1.3% | -9.20 |
| Raytracer | 21.22 (0.09) | 20.91 (0.07) | -1.4% | -12.12 |

(a) With 1 processor.

| Benchs | Base | EH | Diff | T |
|---|---|---|---|---|
| AdapInt | 15.02 (0.25) | 15.40 (0.81) | 2.5% | 1.97 |
| FFT | 4.92 (0.08) | 5.03 (0.10) | 2.2% | 3.78 |
| Fib | 8.34 (0.09) | 8.48 (0.06) | 1.7% | 5.94 |
| Knapsack | 6.36 (0.16) | 6.35 (0.14) | -0.2% | -0.22 |
| QuickSort | 4.31 (0.08) | 4.28 (0.04) | -0.5% | -1.07 |
| Raytracer | 11.18 (0.10) | 11.28 (0.14) | 0.9% | 2.56 |

(b) With 2 processors.

| Benchs | Base | EH | Diff | T |
|---|---|---|---|---|
| AdapInt | 8.47 (1.01) | 8.67 (1.35) | 2.4% | 0.53 |
| FFT | 4.24 (0.09) | 4.18 (0.10) | -1.6% | -2.33 |
| Fib | 4.26 (0.02) | 4.33 (0.04) | 1.6% | 6.47 |
| Knapsack | 4.40 (0.19) | 4.40 (0.15) | 0.1% | 0.07 |
| QuickSort | 2.52 (0.03) | 2.54 (0.03) | 0.9% | 2.34 |
| Raytracer | 6.26 (0.07) | 6.33 (0.07) | 1.1% | 3.27 |

(c) With 4 processors.

**Fig. 13.** Overhead and scalability of the as-if-serial exception handling for DBLFutures. The *Base* and *EH* column list the mean execution time (in seconds) and standard deviation (in parentheses) in the DBLFutures system without and with the as-if-serial exception handling support. The *Diff* column is the difference between *Base* and *EH* (in percent). The last column is the T statistic that we compute using data in the first three columns. We highlight the difference numbers that are statistically significant with 95% confidence.

the thread, but does not prevent it from 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 non-determinism 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 show that as the number of processors increases our overhead does not increase. 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 for larger machines.

## 7. Related work

Many early languages that support futures (e.g. [17,6]) 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 [12], Split-C [24], Cilk [5], etc.

For concurrent programming languages that do support exception handling, most provide exception handling within thread boundaries, but have limited or no support for concurrent exception handling. For example, for Java [15] threads, exceptions that are not handled locally by a thread will not be automatically propagated to other threads, instead, they are silently dropped. The C++ extension Mentat [16] does not address how to provide exception handling. 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 recent languages that support the future construct (e.g. [23,9,1]) provide concurrent exception handling for futures to some extent. For example, in Java 5.0, when a program queries a return value from a future via the `Future.get()` method, 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 facilitates the divide-and-conquer parallel programming methodology for Java programs. 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 receives an exception, the exception is deferred. Any invocation of `val()` then throws the deferred exception. This is similar to the J2SE 5.0 Future model (Fig. 1).

X10 [9] proposes a *rooted exception* model: If activity A is the `root-of` activity B, and A is suspended at a statement awaiting the termination of B, then exceptions thrown in B are propagated to A at that statement when B terminates. Currently, only the `finish` statement marks code regions as a root activity. We expect that future versions of the language will introduce additional, similar statements, including `force()`, 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 upon use of the value that the future returns. Our model propagates asynchronous exceptions to the invocation point of the future function call as if the call is executed sequentially. In this sense, the exception handling mechanism for the Java Remote Method Invocation model [21] is similar to our approach since it propagates remote execution exceptions back to the context of the caller thread at the remote method call invocation site. However, an RMI calls typically block while future calls are non-blocking.

JCilk [26,11] is 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 resulting 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 the same as our model.

There are several major differences between JCilk and our work. 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. Second, JCilk requires a `spawn` statement surrounded by a special `cilk try` construct if exceptions are possible. In our model, only the traditional 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.). Our system is significantly simpler that JCilk since we modify the Java Virtual Machine and thus, have direct access to Java call stacks and perform stack splitting.

A small number of non-Java, concurrent object-oriented languages have built-in concurrent exception handling support, e.g., DOOCE [35] and Arche [20,19]. 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, DOOCE allows multiple `catch` blocks to be associated with one `try` block. In case of exceptions, all `catch` blocks that match thrown exceptions, individually or partially, are executed by the DOOCE runtime. In addition, DOOCE supports two different models 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 at 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 the system propagates to other processes with which it communicates synchronously. For multiple concurrent exceptions, Arche enables 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. [32,28,7,33,39]) focus on general models for exception handling in distributed systems. These models usually assume that processes participating in a parallel computation are organized in a coordinated fashion in a structure, such as a *conversation* [32] or an *atomic action* [28]. Processes can enter such a structure asynchronously but must exit the structure synchronously. When one process throws an exception, the runtime informs all other processes, each of which in turn invokes an appropriate handler. When multiple exceptions occur concurrently, the runtime employs *exception resolution* [7] in which it coalesces multiple exceptions into a single one based on different resolution strategies. Example strategies include the use of an exception resolution tree [7], an exception resolution graph [38], or user defined resolution functions [20].

Our exception handling model for DBLFutures is different from this work in that the we preserve serial semantics grants. By doing so, we are able to greatly simplify our model and its implementation. For example, the exception resolution strategy of our model selects the exception that should occur first given serial semantics. In addition, although our model organizes 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, as is required by conversations and atomic actions. 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.

Finally, an initial version of this work is described in [41]. In this paper, we extend the description and provide additional details on our approach that we were unable to in this initial study due to space constraints. Moreover, in this paper, we investigate how to provide as-if-serial exception handling semantics in a DBLFutures system that supports as-if-serial side effect semantics. Such side effect semantics for futures was first proposed for Java in [37], as *Safe Futures*, as an extension to the J2SE 5.0 Future library-based support.

Safe Futures for Java uses object versioning and task revocation to enforce the semantic transparency of futures automatically. By doing so, Safe Futures frees programmers from considering the side effects of future executions (ensuring correctness). This transaction style support is complementary to our as-if-serial exception handling model. We note that the authors of this prior work 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) this has been implemented this since the authors provide no details on their design and implementation of such a mechanism.

## 8. Conclusions

In this paper, we propose an *as-if-serial* exception handling mechanism for DBLFutures. DBLFutures is a simple parallel programming extension of Java that enables programmers to use futures in Java [40]. Our as-if-serial exception handling mechanism delivers exceptions to the same point as they are delivered when 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 DBLFutures system as well as into a DBLFutures system that provides side effect guarantees, 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 detail the implementation of DBLFutures with as-if-serial side effects and a number of novel extensions that we provide for their efficient implementation. This system to which we refer to as Safe DBLFutures (SDBLFutures) employ a technique similar to that of software transactions within a Java Virtual Machine (JVM) to guarantee correct execution (relative to the serial execution of the program) of future methods with side effects. We integrate this semantics with that for as-if-serial exception handling to simplify the implementation of the latter and to complete our as-if-serial JVM support.

We implement and evaluate exception and side effect semantics for futures in Java through an extension to our DBLFutures framework for the Jikes Research Virtual Machine from IBM Research. Our results for the implementation of our exception handling semantics show that our system introduces negligible overhead for applications without exceptions, and guarantees serial semantics of exception handling for applications that throw exceptions. Our evaluation of the Safe DBLFutures systems (SDBLFutures) indicates that with JVM modification, SDBLFutures eliminates the significant memory management overhead of the Safe Future system from prior work (which employs a library-based approach to future implementation). The overhead that SDBLFutures imposes to guarantee correctness via software transactions is the same as that of the Safe Future system.

## Acknowledgments

We sincerely thank the reviewers of this article for providing use with very detailed and thoughtful suggestions for its improvement. This work was funded in part by Microsoft and NSF grants CCF-0444412 and CNS-0546737.

## References

[1] E. Allan, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele, S. Tobin-Hochstadt, The Fortress language specification version 0.785, Tech. Rep., Sun Microsystems, 2005. URL http://research.sun.com/projects/plrg/.
[2] E. Allan, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele, S. Tobin-Hochstadt, The Fortress language specification version 0.954, Tech. Rep., Sun Microsystems, 2006. URL http://research.sun.com/projects/plrg/.
[3] M. Arnold, S. Fink, D. Grove, M. Hind, P. Sweeney, Adaptive optimization in the Jalapeño JVM, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2000.
[4] S.M. Blackburn, A.L. Hosking, Barriers: Friend or foe?, in: International Symposium on Memory Management, 2004, pp. 143–151.
[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, in: ACM Conference on Principles of Programming Languages, 1995, pp. 207–216.
[6] D. Callahan, 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, 1990, pp. 95–113.
[7] R. Campbell, B. Randell, Error recovery in asynchronous systems, IEEE Trans. Softw. Eng. 12 (8) (1986) 811–826.

[8] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the chapel language, International Journal of High Performance Computing Applications 21 (3) (2007) 291–312.
[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: An object-oriented approach to non-uniform cluster computing, in: ACM Conference on Object Oriented Programming Systems Languages and Applications, 2005, pp. 519–538.
[10] F. Cristian, Exception handling and software fault tolerance, IEEE Trans. Comput. 31 (6) (1982) 531–540.
[11] J. Danaher, The JCilk-1 runtime system, Master's Thesis, MIT Department of Electrical Engineering and Computer Science, Jun. 2005.
[12] T. Ellis, I. Phillips, T. Lahey, Fortran 90 Programming, 1st ed., Addison Wesley, 1994.
[13] M. Frigo, C. Leiserson, K. Randall, The implementation of the Cilk-5 multithreaded language, in: ACM Conference on Programming Language Design and Implementation, PLDI, 1998, pp. 212–223.
[14] G. Hill, ACM Alg. 395: Student's T-distribution, Commun. ACM 13 (10) (1970) 617–619.
[15] J. Gosling, B. Joy, G. Steel, G. Bracha, The Java Language Specification Second Edition, 2nd ed., Addison Wesley, 2000.
[16] A. Grimshaw, Easy-to-use object-oriented parallel processing with mentat, Computer 26 (5) (1993) 39–51.
[17] R. Halstead, Multilisp: A language for concurrent symbolic computation, ACM Trans. Program. Lang. Syst. 7 (4) (1985) 501–538.
[18] Intermetrics (Ed.) Information technology – Programming languages – Ada, ISO/IEC 8652:1995(E), 1995.
[19] V. Issarny, An exception handling model for parallel programming and its verification, in: Conference on Software for Citical Systems, 1991, pp. 92–100.
[20] V. Issarny, An exception handling mechanism for parallel object-oriented programming: Towards reusable, robust distributed software, J. Object-Oriented Program. 6 (6) (1993) 29–39.
[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, K. Yelick, Parallel programming in Split-C, in: ACM/IEEE Conference on Supercomputing, 1993, pp. 262–273.
[25] D. Lea, A Java fork/join framework, in: ACM Java Grande, 2000, pp. 36–43.
[26] I. Lee, The JCilk multithreaded language, Master's Thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Aug. 2005.
[27] B. Liskov, A. Snyder, Exception handling in CLU, IEEE Trans. Softw. Eng. SE-5 (6) (1979) 546–558.
[28] D.B. Lomet, Process structuring, synchronization, and recovery using atomic actions, in: ACM Conference on Language Design for Reliable Software, 1977, pp. 128–137.
[29] B. Meyer, Eiffel: The Language, 2nd ed., Prentice Hall, 1992.
[30] E. Mohr, D.A. Kranz, J.R.H. Halstead, Lazy task creation: A technique for increasing the granularity of parallel programs, IEEE Trans. Parallel Distrib. Syst. 2 (3) (1991) 264–280.
[31] OpenMP specifications. http://www.openmp.org/specs.
[32] B. Randell, System structure for software fault tolerance, in: International Conference on Reliable Software, 1975, pp. 437–449.
[33] A. Romanovsky, J. Xu, B. Randell, Exception handling and resolution in distributed object-oriented systems, in: International Conference on Distributed Computing Systems, ICDCS '96, IEEE Computer Society, Washington, DC, USA, 1996, p. 545.
[34] B. Stroustrup, The C++ Programming Language, 2nd ed., Addison Wesley, 1991.
[35] S. Tazuneki, T. Yoshida, Concurrent exception handling in a distributed object-oriented computing environment, in: International Conference on Parallel and Distributed Systems: Workshops, IEEE Computer Society, Washington, DC, USA, 2000, p. 75.
[36] R. van Nieuwpoort, J. Maassen, T. Kielmann, H. Bal, Satin: Simple and efficient Java-based grid programming, Scalable Computing: Practice and Experience 6 (3) (2005) 19–32.
[37] A. Welc, S. Jagannathan, A. Hosking, Safe futures for Java, in: ACM Conference on Object Oriented Programming Systems Languages and Applications, 2005, pp. 439–453.
[38] J. Xu, A. Romanovsky, B. Randell, Coordinated exception handling in distributed object systems: From model to system implementation, in: International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 1998, p. 12.
[39] J. Xu, A. Romanovsky, B. Randell, Concurrent exception handling and resolution in distributed object systems, IEEE Trans. Parallel Distrib. Syst. 11 (10) (2000) 1019–1032.
[40] L. Zhang, C. Krintz, P. Nagpurkar, Language and virtual machine support for efficient fine-grained futures in Java, in: International Conference on Parallel Architectures and Compilation Techniques, PACT, 2007.
[41] L. Zhang, C. Krintz, P. Nagpurkar, Supporting exception handling for futures in Java, in: International Conference on Principles and Practice on Programming in Java, PPPJ, 2007.
[42] L. Zhang, C. Krintz, S. Soman, Efficient support of fine-grained futures in Java, in: International Conference on Parallel and Distributed Computing Systems, PDCS, 2006.