# Safe Futures for Java

Adam Welc
welc@cs.purdue.edu

Suresh Jagannathan
suresh@cs.purdue.edu

Antony Hosking
hosking@cs.purdue.edu

Department of Computer Science
Purdue University
West Lafayette, IN 47907

## ABSTRACT

A future is a simple and elegant abstraction that allows concurrency to be expressed often through a relatively small rewrite of a sequential program. In the absence of side-effects, futures serve as benign annotations that mark potentially concurrent regions of code. Unfortunately, when computation relies heavily on mutation as is the case in Java, its meaning is less clear, and much of its intended simplicity lost.

This paper explores the definition and implementation of *safe* futures for Java. One can think of safe futures as truly transparent annotations on method calls, which designate opportunities for concurrency. Serial programs can be made concurrent simply by replacing standard method calls with future invocations. Most significantly, even though some parts of the program are executed concurrently and may indeed operate on shared data, the semblance of serial execution is nonetheless preserved. Thus, program reasoning is simplified since data dependencies present in a sequential program are not violated in a version augmented with safe futures.

Besides presenting a programming model and API for safe futures, we formalize the safety conditions that must be satisfied to ensure equivalence between a sequential Java program and its future-annotated counterpart. A detailed implementation study is also provided. Our implementation exploits techniques such as object versioning and task revocation to guarantee necessary safety conditions. We also present an extensive experimental evaluation of our implementation to quantify overheads and limitations. Our experiments indicate that for programs with modest mutation rates on shared data, applications can use futures to profitably exploit parallelism, without sacrificing safety.

## Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures; Control structures; Procedures, functions, and subroutines*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages; Object-oriented languages*


*OOPSLA'05,* October 16–20, 2005, San Diego, California, USA.


## General Terms

Experimentation, Languages, Measurement, Performance

## Keywords

Java, concurrency, futures, safety

## 1. INTRODUCTION

The new `java.util.concurrent` package [27] that is part of the Java 2 Platform Standard Edition 5.0 specifies an interface to define *futures*, a concurrency abstraction first proposed for MultiLisp [21]. A future defines an asynchronous thread of control. In MultiLisp, the **future** keyword annotates expressions that can be evaluated in parallel with the rest of the program. The object returned by such an annotated expression is a placeholder that ultimately holds the value yielded by the expression. When a program expression requires this value, it attempts to *claim* or *touch* the future; this operation serves to synchronize evaluation of the future with the thread performing the claim.

Futures are an elegant alternative to programming with explicit threads because they often allow concurrent programs to be created through a relatively small rewrite of its sequential counterpart. Furthermore, in the absence of side-effects, futures satisfy a simple safety property: if sequential program $P$ is annotated with futures to yield concurrent program $P_F$, then the observable behavior of $P$ is equivalent to $P_F$. Indeed, because futures were originally provided as annotations on programs, their effect was intended to be transparent, visible only in the form of improved concurrency, without altering the meaning of the original sequential program.

Of course, in the presence of mutation this pleasant property no longer holds. Indeed, by themselves, Java futures provide no special safety guarantees. Tasks spawned as a result of a future may perform updates on shared data concurrently accessed by other tasks, including the task that spawned it. While this is not a serious issue in functional or mostly-functional languages where updates to shared data occur infrequently (if at all), it is significantly more problematic in Java where computation is typically structured in terms of modification to shared objects. We believe many of the notable benefits from using futures are significantly weakened in Java as it is currently specified and implemented because of this lack of transparency with respect to access to shared data. To achieve some measure of safety, programs must be further refined to provide synchronization on potentially shared objects. Unfortunately, even adding synchronization does not guarantee that the resulting behavior is equivalent to a sequential execution since the effects of a future may be arbitrarily interleaved with the effects performed by the computation that spawned it.

Exploring and remedying this disconnect is the focus of this pa-

per. We are interested in preserving the original spirit of futures as a simple device to inject concurrency into a sequential program *without* violating any of the program's invariants. Achieving this goal, however, is non-trivial. Consider a future $f$ that executes concurrently with the task $C_f$ that evaluates $f$'s continuation. A continuation of a future is the computation that logically follows it. Safe execution of $f$ may be compromised if it observes the effects of operations performed by $C_f$; for example, if $C_f$ updates an object that is subsequently read by $f$. Similarly, safe execution of $C_f$ may be compromised if it accesses an object that is subsequently written by $f$; for example, if $C_f$ performs a read of an object that is subsequently written by $f$. Both these cases lead to different behavior than if the future and its continuation were evaluated sequentially.

To preserve these desired safety invariants, we define a semantics and implementation for *safe* Java futures. Our semantics formalizes the notion of safety by imposing constraints on the set of schedules that can be generated by a program in which concurrency is expressed exclusively through the use of futures. Our implementation automatically revokes executions that fail to observe the effects of the futures they spawned, and uses object versioning to ensure that futures do not observe the effects of their continuations.

## 1.1 Contributions

This paper presents the design, semantics, an implementation, and a performance evaluation of safe futures for Java. Our contributions are summarized as follows:

- We motivate the design of safe futures, present an API, and associated programming model. Our model allows programmers to view futures as simply benign annotations on method calls inserted where concurrency may be profitably exploited.
- We define a semantics for an object calculus similar to Classic Java [20] extended with futures. The semantics yields schedules, an interleaved trace of read and write events. We define safety conditions on schedules that capture notions of validity with respect to these operations performed by concurrently evaluating tasks. We prove a soundness result that shows every safe schedule is indeed equivalent to a serial one in which no interleavings are present.
- We present details of an implementation built on top of the Jikes RVM [9, 3] that supports object versioning and task revocation to ensure safety. In addition to providing the rationale for our implementation decisions, we describe both necessary compiler and run-time modifications critical to the implementation.
- A detailed experimental study is also given. In addition to exploring the performance impact of safe futures on parallelizable programs adapted from the Java Grande benchmark suite [41], we also provide a detailed performance study on 007, a well-known synthetic database benchmark [11] that allows us to more accurately assess implementation overheads, and performance limitations. Our experiments show that for programs with modest mutation rates on shared data, our approach can profitably exploit parallelism, without sacrificing safety.

# 2. DESIGN

Adding futures to an imperative object-oriented language like Java raises several important design issues. Our foremost design goal is to preserve the spirit of programming with futures that made it so appropriate for functional programming: the expectation that a future performs its computation *as if* it had been invoked as a synchronous function call, rather than an asynchronous thread. This is a natural evolution of functional programming languages, since functional programs typically perform infrequent mutation of shared objects in the heap, instead obtaining results of symbolic computations as constructions of heap-allocated data structures. In contrast, computations in imperative languages like Java often obtain their results via mutation of container objects (*eg*, arrays and hash tables) in which results are deposited via mutation.

Unfortunately, current designs for futures in Java [27] neglect to treat them as semantically transparent annotations as originally proposed for functional languages [15, 16]. We believe this defeats the original purpose of futures as an elegant and minimalist approach to exploiting parallelism in existing programs, since programmers are forced to reason about the side-effects of future executions to ensure correctness of programs that use them. Instead, we believe that strong notions of safety for futures is what makes them so powerful, where safety is ensured by the run-time system rather than left as a burden for the programmer.

We now proceed to discussion of an API for safe futures, their associated programming model, and their interaction with existing Java concurrency mechanisms.

## 2.1 API for safe futures

A major challenge in introducing any new language abstraction is to make it intuitive and easy to use. To ground our design, we begin with the existing Java futures API [27] that is now part of the Java 2 Platform Standard Edition 5.0 (J2SE 5.0). Snippets of this existing API appear in Figure 1, which embodies futures in the interface `Future`. The `get` operation on a `Future` simply waits if necessary for the computation it encapsulates to complete, and then retrieves its result. We omit here those operations on futures that are not germane to our remaining discussion.

We would also like to note that our notion of safe futures is independent of any particular API. A solution similar to that of Pratikakis *et al* [36] in which `get` operations are implicit and futures are effectively only annotations on method calls would be a perfectly viable alternative to using the current API from J2SE 5.0.

In J2SE 5.0 there is an implementation of the `Future` interface in the class `FutureTask`. Again, we omit details not germane to our discussion. Here, the constructor for `FutureTask` creates a future that will, upon invocation of the `run` method, execute the given `Callable` by invoking its `call` method. If the call throws an exception, it is delivered to the caller at the point where it invokes the `get` method, wrapped up in an `ExecutionException`.

Our design calls for a new implementation of `Future`, namely `SafeFuture`, which is presented in Figure 2. Our semantics for `SafeFuture` demand that the program fragments appearing in Figure 3 be semantically equivalent, regardless of the computation performed by the given `Callable<V> c`, and the code surrounding its invocation as a simple call or as a future.

To preserve the transparency of future calls, any uncaught exception thrown by the future call (*ie*, from the `call` method of the `Callable`) will be delivered to the caller at the point of invocation of the `run` method, and the effects of the code following the run method will be revoked. The effects of the future call up to the point it threw the exception will remain. These semantics preserve equivalence with the simple call.

A more detailed example program appears in Figure 4. A future defined in the sample code fragment computes the sum of the elements in the array of integers `a` concurrently with a call to the static method `bar` on class `Foo`, which receives argument `a`. Note that method `bar` may access (and modify) `a` concurrently with the fu-

```
public interface Future<V> {
    V get()
      throws InterruptedException,
             ExecutionException;
}

public class FutureTask<V>
  implements Future<V>, Runnable {
    FutureTask(Callable<V> callable)
      throws NullPointerException
    { ... }
    V get()
      throws InterruptedException,
             ExecutionException
    { ... }
    void run() { ... }
}

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

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

```
public class SafeFuture<V>
  implements Future<V>, Runnable {
    SafeFuture(Callable<V> callable)
      throws NullPointerException
    { ... }
    V get()
      throws InterruptedException,
             ExecutionException
    { ... }
    void run() { ... }
}
```

**Figure 2: Safe futures API**

ture computation. Our semantics require that the observable behavior of calls to methods `serial` and `concurrent` be the same. Replacing uses of `SafeFuture` with the existing `FutureTask` from J2SE 5.0 provides no such guarantee.

## 2.2 Programming model

The programming model enabled by use of safe futures permits straightforward exploitation of latent parallelism in programs. One can think of safe futures as transparent annotations on method calls, which designate opportunities for concurrency. Serial programs can be made concurrent simply by replacing standard method calls with future invocations. This greatly eases the task of the programmer, since all reasoning about the behavior of the program can be inferred from its original serial execution. Even though some parts of the program are executed concurrently, the semblance of serial execution is preserved. Of course, the cost of using futures may outweigh exploitable parallelism, so placement of future invocations has performance implications.

Under our current programming model, safety does not extend to covering the interaction between futures and Java threads. Threads which execute concurrently with futures might observe the actions of concurrently executing futures and their continuations out-of-order. Threads could be also incorrectly used to pass partial computation results between a future and its continuation thus violating serial execution semantics. We defer consideration of these issues

```
Callable<V> c = ...;
```

```
...                    |  ...
V v = c.call();        |  Future<V> f
...                    |    = new SafeFuture<V>(c);
                       |  f.run();
                       |  ...
                       |  V v = f.get();
```

**Figure 3: Semantically equivalent code fragments**

```
public class Example
  implements Callable<Integer>
{
  int[] a = new int[]{1,2,3};

  public Integer call() {
    int sum = 0;
    for (int v : a) sum += v;
    return sum;
  }
  int serial() {
    Integer sum = call();
    Foo.bar(a);
    return sum;
  }
  int concurrent() {
    Future<Integer> f
      = new SafeFuture<Integer>(this);
    f.run();
    Foo.bar(a);
    return f.get();
  }
  public static void main (String[] args) {
    int serial = new Example().serial();
    int concurrent = new Example().concurrent();
    assert serial == concurrent;
  }
}
```

**Figure 4: Using safe futures (with automatic boxing/unboxing of `int`/`Integer` supported by J2SE 5.0)**

to future work.

# 3. SEMANTICS

To examine notions of safety with respect to interleavings of actions that operate within a future and its continuation, we define a semantics for a call-by-value object calculus similar to Classic Java [20] extended with threads, and a `future` construct. The semantics yields a *schedule* – a sequence of read and write operations performed during the execution of a program. A schedule is *serial* when all the operations of a program are executed within a single (main) thread. A schedule is *concurrent* if fragments of a program are executed concurrently by separate threads; in this case, the actions of these threads may be interleaved with one another. We impose safety conditions on concurrent schedules to verify that operation interleavings do not violate safety invariants. Informally, a concurrent schedule is safe if it is equivalent, in terms of its actions on shared data, to some serial schedule.

SYNTAX:

$$
\begin{array}{rcl}
P & ::= & P \mid P \mid \mathtt{t}[\mathtt{e}]_{\mathtt{l}} \\
L & ::= & \mathtt{class\ C}\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{m}(\overline{\mathtt{x}})\{\mathtt{e}\} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{l} \mid \mathtt{this} \mid \mathtt{e.f} \mid \mathtt{e.f} := \mathtt{e} \\
 & \mid & \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{e;e} \mid \mathtt{new\ C}() \\
 & \mid & \mathtt{future}\ (\mathtt{e}) \mid \mathtt{get}\ (\mathtt{e})
\end{array}
$$

PROGRAM STATES:

$$
\begin{array}{rcll}
\mathtt{t} & \in & \textit{Tid} & \\
P & \in & \textit{Process} & \\
\mathtt{x} & \in & \textit{Var} & \\
\mathtt{l} & \in & \textit{Loc} & \\
\mathtt{v} & \in & \textit{Val} & = \ \mathtt{null} \mid \mathtt{C}(\overline{\mathtt{l}}) \mid \mathtt{l} \\
\Gamma & \in & \textit{Store} & = \ \textit{Loc} \rightarrow \textit{Val} \\
\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}} & \in & \textit{Ops} & = \ (\{\mathbf{rd}, \mathbf{wr}\} \times \textit{Tid} \times \textit{Loc}) \\
S = \overline{\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}}} & \in & \textit{Schedule} & = \ \mathcal{P}(\textit{Ops}) \\
\Lambda & \in & \textit{State} & = \ \textit{Process} \times \textit{Store} \times \textit{Schedule}
\end{array}
$$

EVALUATION CONTEXTS:

$$
\begin{array}{rcl}
E & ::= & \bullet \mid \\
 & & E[\bullet].\mathtt{f} := \mathtt{e} \mid \mathtt{l.f} := E[\bullet] \mid \\
 & & E[\bullet].\mathtt{m}(\overline{\mathtt{e}}) \mid \mathtt{l.m}(\overline{\mathtt{l}}\ E[\bullet]\ \overline{\mathtt{e}}) \mid E[\bullet]\ ;\mathtt{e} \mid \\
 & & \mathtt{get}\ (E[\bullet]) \\
 & & \\
E^{\mathtt{t}}_{P}[\mathtt{e}]_{\mathtt{l}} & ::= & P \mid \mathtt{t}[E[\mathtt{e}]]_{\mathtt{l}}
\end{array}
$$

$$\frac{E[\mathtt{e}], \Gamma, S \rightarrow E[\mathtt{e}'], \Gamma', S'}{E^{\mathtt{t}}_{P}[\mathtt{e}]_{\mathtt{l}}, \Gamma, S \Longrightarrow E^{\mathtt{t}}_{P}[\mathtt{e}']_{\mathtt{l}}, \Gamma', S'}$$

SEQUENTIAL EVALUATION RULES:

$$\frac{}{E[\mathtt{l;e}], \Gamma, S \rightarrow E[\mathtt{e}], \Gamma, S}$$

$$\frac{\mathtt{class\ C}\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \in L \qquad \mathtt{m}(\overline{\mathtt{x}})\{\mathtt{e}\} \in \overline{\mathtt{M}}}{E[\mathtt{l.m}(\overline{\mathtt{l}})], \Gamma, S \rightarrow E[[\mathtt{l}/\mathtt{this}, \overline{\mathtt{l}}/\overline{\mathtt{x}}]\mathtt{e}], \Gamma, S}$$

$$\frac{\mathtt{class\ C}\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \in L \qquad \Gamma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}'}) \qquad S' = S.\mathbf{rd}_{\mathtt{t}}\ \mathtt{l}'_i}{E[\mathtt{l.f}_i], \Gamma, S \rightarrow E[\mathtt{l}'_i], \Gamma, S'}$$

$$\frac{\Gamma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}''}) \quad \Gamma(\mathtt{l}') = \mathtt{v} \qquad \Gamma' = \Gamma[l''_i \mapsto \mathtt{v}] \qquad S' = S.\mathbf{rd}_{\mathtt{t}}\ \mathtt{l}'.\mathbf{wr}_{\mathtt{t}}\ \mathtt{l}''_i}{E[\mathtt{l.f}_i := \mathtt{l}'], \Gamma, S \rightarrow E[\mathtt{l}'], \Gamma', S'}$$

$$\frac{\mathtt{l}', \overline{\mathtt{l}}\ \textit{fresh} \qquad \Gamma' = \Gamma[l' \mapsto \mathtt{C}(\overline{\mathtt{l}}), \overline{\mathtt{l}} \mapsto \mathtt{null}] \qquad S' = S.\mathbf{wr}_{\mathtt{t}}\ \mathtt{l}_1.\ \ldots \mathbf{wr}_{\mathtt{t}}\ \mathtt{l}_n.\mathbf{wr}_{\mathtt{t}}\ \mathtt{l}' \qquad l_1, \ldots, \mathtt{l}_n \in \overline{\mathtt{l}}}{E[\mathtt{new\ C}()], \Gamma, S \rightarrow E[\mathtt{l}'], \Gamma', S'}$$

EVALUATION RULES FOR FUTURES:

$$\frac{\mathtt{t}', \mathtt{t}''\ \textit{fresh} \quad t \leq t' \leq t'' \quad \mathtt{l}'\ \textit{fresh}}{E^{\mathtt{t}}_{P}[\mathtt{future}\ (\mathtt{e})]_{\mathtt{l}}, \Gamma, S \Longrightarrow P \mid \mathtt{t}'[\mathtt{e}]_{\mathtt{l}'} \mid \mathtt{t}''[E[\mathtt{l}']]_{\mathtt{l}}, \Gamma, S}$$

$$\frac{P = P' \mid \mathtt{t}'[\mathtt{l}'']_{\mathtt{l}'}}{E^{\mathtt{t}}_{P}[\mathtt{get}\ (\mathtt{l}')]_{\mathtt{l}}, \Gamma, S \Longrightarrow E^{\mathtt{t}}_{P}[\mathtt{l}'']_{\mathtt{l}}, \Gamma, S}$$

**Figure 5: Syntax and semantics of of safe futures.**

The syntax and semantics of the calculus are given in Figure 5. A program defines a collection of class definitions, and a collection of threads. Classes are all uniquely named, and define a collection of instance fields and instance methods that operate over these fields. Every method consists of an expression whose value is returned as the result of a call to that method. An expression is either a variable, a location that references an object, the pseudo-variable `this`, a field reference, an assignment, a method invocation, a sequencing operation, an object creation operation, a `future` creation, or a `get` expression that claims a future.

Every class has a unique (nullary) constructor to initialize object fields. The application of a constructor returns a reference to an object instantiated from the class definition. A value is either `null`, an object instantiated from a class containing locations for the fields declared by the class, or a location that serves as a placeholder to hold the result of evaluating a future. A thread is uniquely labeled with a thread identifier, and a placeholder location.

We take metavariables $L$ to range over class declarations, `C` to range over class names, `M` to range over methods, `m` to range over method names, `f` and `x` to range over fields and parameters, respectively, `l` to range over locations, and `v` to range over object values. We also use $P$ for process terms, and `e` for expressions. We use over-bar to represent a finite ordered sequence, for instance, $\overline{f}$ represents $f_1\ f_2 \ldots f_n$. The term $\overline{\alpha}\alpha$ denotes the extension of the sequence $\overline{\alpha}$ with a single element $\alpha$, and $\overline{\alpha}\overline{\alpha'}$ denotes sequence concatenation, $S.\mathrm{OP}_{\mathtt{t}}$ denotes the extension of schedule $S$ with operation $\mathrm{OP}_{\mathtt{t}}$.

Program evaluation and schedule construction are specified by a global reduction relation, $P, \Gamma, S \Longrightarrow P', \Gamma', S'$, that maps a program state to a new program state. A program state consists of a collection of evaluating threads ($P$), a global store ($\Gamma$) to map locations to values (specifically, either `null` or an object), and schedules ($S$) to define a global interleaved sequence of actions performed by threads. Local reductions within a thread are specified by an auxiliary relation, $\mathtt{e}, \Gamma, S \rightarrow \mathtt{e}', \Gamma', S'$ that evaluates expression `e` within some thread to a new expression $\mathtt{e}'$; in doing so, a new store, and

schedule may result. The only actions that are recorded by a schedule are those that read and write locations. The interpretation of schedules with respect to safety is the topic of the next section.

We use evaluation contexts to specify order of evaluation within a thread, and to prevent premature evaluation of the expression encapsulated within a `future` annotation. We define a process context $E_P^{\mathtt{t}}[\mathtt{e}]_{\mathtt{l}}$ to denote an expression `e` available for execution by thread $\mathtt{t} \in P$ in a program state; the label `l` denotes a placeholder location that holds the result of `e`'s evaluation.

The sequential evaluation rules are standard: holes in evaluation contexts can be replaced by the value of the expression substituted for the hole, sequence operations evaluate left-to-right, method invocation evaluates the method body by substituting occurrences of the pseudo-variable `this` with the location holding the reference to the receiver object, in addition to substituting occurrences of formals with the locations yielded by evaluation of the actuals, assuming variables are suitably $\alpha$-renamed. Read and write operations augment the schedule in the obvious way. A `new` expression extends the schedule with writes to all instance fields (with null values).

An expression of the form `future` (`e`) causes `e`'s evaluation to take place in a new thread $\mathtt{t}'$. A fresh location $\mathtt{l}'$ is created as a placeholder to hold the result of evaluating this future. Thus, $\mathtt{t}'[\mathtt{e}]_{\mathtt{l}'}$ denotes a thread with identifier $\mathtt{t}'$ that evaluates expression `e` and stores the result of this evaluation into $\mathtt{l}'$.

In addition to the thread responsible for computing the value of the future, a new thread $\mathtt{t}''$ is created to evaluate the future's continuation. As a result, the parent thread is no longer relevant. This specification simplifies the safety conditions discussed below. The thread identifiers associated with threads created by a future expression are related under a total ordering ($\leq$). Informally, this ordering captures the logical (sequential) order in which actions performed by the threads must be evaluated. Thus, if $\mathtt{t}' \leq \mathtt{t}''$, then either $\mathtt{t}' = \mathtt{t}''$, or all actions performed by $\mathtt{t}'$ must logically take place before $\mathtt{t}''$. In particular, effects induced by actions performed by $\mathtt{t}''$ must not be visible to operations in $\mathtt{t}'$.

Synchronization takes place through the `get` expression. In the rule for `get`, the location label $\mathtt{l}'$ represents a placeholder or synchronization point that holds the value of a task spawned by a future. The rule is satisfied precisely when the associated future (say, `future` (`e`)) has completed. When this occurs, the process state will contain a thread with shape $\mathtt{t}[\mathtt{l}'']_{\mathtt{l}'}$ where $\mathtt{l}''$ is the location yielded by evaluation of `e`.

## 3.1 Safety

A schedule defines a sequence of possibly interleaved operations among threads. The correctness of a schedule, therefore, must impose safety constraints on read and write operations. These constraints guarantee that the injection of futures into an otherwise sequential program does not alter the meaning of the program. Thus, these constraints must ensure that interleavings are benign with respect to read and write operations. The semantics does not permit reordering of operations within a thread.

There are two conditions (roughly equivalent to the well-known Bernstein conditions [5]) that must hold on schedules to guarantee this property: (1) an access to a location *l* (either a read or a write) performed by a future should not witness a write to *l* performed earlier by its continuation, and (2) a write operation to some location *l* performed by a future should be visible to the first access (either a read or a write) made to *l* by its continuation. In other words, no write to a location *l* by a future's continuation can occur before any operations on *l* by the future, and all writes to a location *l* by the future must occur before any operation to *l* by the continuation. Note that these conditions do not prohibit interleaved operations by a future and its continuation to distinct locations.

We summarize these constraints in terms of two safety rules, *csafe* and *fsafe*, resp. The former captures the notion of when an operation performed by a continuation is safe with respect to the actions performed by the future within a schedule, and the latter captures the notion of when an operation performed by a future is safe with respect to its continuation within a schedule.

$$\frac{\mathbf{wr}_{\mathtt{t}'}\,\mathtt{l}, \mathbf{rd}_{\mathtt{t}'}\,\mathtt{l} \notin S', \quad \mathtt{t}' \leq \mathtt{t}}{csafe(S.\mathbf{wr}_{\mathtt{t}}\,\mathtt{l}.S')} \qquad \frac{\mathbf{wr}_{\mathtt{t}'}\,\mathtt{l}, \mathbf{rd}_{\mathtt{t}'}\,\mathtt{l} \notin S, \quad \mathtt{t} \leq \mathtt{t}'}{fsafe(S.\mathbf{wr}_{\mathtt{t}}\,\mathtt{l}.S')}$$

DEFINITION 1. *(Schedule Safety)*
*A schedule S is safe if csafe(S) and fsafe(S) hold.*

To validate the safety of an interleaved schedule, we must ensure that its observable behavior is equivalent to the behavior of a corresponding program in which futures have no computational effect. In such a program, evaluation of the future's continuation is delayed until the future itself is fully evaluated. This trivially enforces sequential order between all operations executed by the future and all operations executed by the continuation and thus automatically yields a serial schedule.

We first introduce the notion of a schedule *permutation* that allows us to define an equivalence relation on schedules:

DEFINITION 2. *(Permute)*
*Schedule S is a* permutation *of schedule S′ (written $S \leftrightarrow S'$), if len(S) = len(S′) and for every $\mathrm{OP}_{\mathtt{t}}^{\mathtt{l}_i} \in S$, there exists a unique $\mathrm{OP}_{\mathtt{t}}^{\mathtt{l}_j} \in S'$.*

A *serial* schedule is a schedule in which no interleaving among operations of different threads occurs:

DEFINITION 3. *(Serial Schedule)*
*Schedule $S = \mathrm{OP}_{\mathtt{t}_1}^{\mathtt{l}_1} \ldots . \mathrm{OP}_{\mathtt{t}_n}^{\mathtt{l}_n}$ is serial if for all $\mathrm{OP}_{\mathtt{t}_j}^{\mathtt{l}_j}$ there does not exist $\mathrm{OP}_{\mathtt{t}_k}^{\mathtt{l}_j}$, $\mathtt{k} > \mathtt{j}$ such that $\mathtt{t}_k < \mathtt{t}_j$.*

We wish to show that any safe schedule can be permuted to a serial one since a serial schedule reflects an execution in which operations executed by a future are not interleaved with operations performed by its continuation. Effectively, a serial schedule reflects an execution in which a spawned future runs to completion before any operations in its continuation are allowed to execute; in other words, a serial schedule corresponds to a program execution in which futures have no computational effect.

We first appeal to a lemma that allows us to permute adjacent operations belonging to different threads in a safe schedule:

LEMMA 1. *(Permutation)*
*Let schedule $S = \mathrm{OP}_{\mathtt{t}_1}^{\mathtt{l}_1}.\mathrm{OP}_{\mathtt{t}_2}^{\mathtt{l}_2}$ be safe. Then if S is safe, there exists a serial schedule S′ such that $S \leftrightarrow S'$.*

**Proof.** If $\mathtt{t}_1 \leq \mathtt{t}_2$, then the schedule is trivially serial. If $\mathtt{t}_1 > \mathtt{t}_2$, and because *S* is safe, it must be the case that either (a) $\mathtt{l}_1 \neq \mathtt{l}_2$, or (b) $\mathtt{l}_1 = \mathtt{l}_2 = \mathtt{l}$, and $\mathrm{OP}_{\mathtt{t}_2}^{\mathtt{l}} = \mathbf{rd}_{\mathtt{t}_2}\,\mathtt{l}$. In both cases, we can choose $S' = \mathrm{OP}_{\mathtt{t}_2}^{\mathtt{l}_2}.\mathrm{OP}_{\mathtt{t}_1}^{\mathtt{l}_1}$.

Our soundness result generalizes this lemma over schedules of arbitrary length:

THEOREM 1. *(Soundness)*
*If schedule S is safe, then there exists a serial schedule S′ such that $S \leftrightarrow S'$.*

**Proof.** The proof is by induction on schedule length. Lemma 1 satisfies the base case. Suppose $S = S_1.\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}}$ where $len(S_1) > 2$. By the induction hypothesis, there exists a serial schedule $S'_1$ such that $S'_1 \leftrightarrow S_1$. Suppose $S'_1 = \mathrm{OP}^{\mathtt{l}_1}_{\mathtt{t}_1}.\cdots.\mathrm{OP}^{\mathtt{l}_k}_{\mathtt{t}_k}$. First, we need to show that $S'' = S'_1.\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}}$ is safe. Suppose otherwise. Then, it must be the case that either (a) there exists some $\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}'} \in S'_1$ such that $\mathtt{t} < \mathtt{t}'$, and $\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}} = \mathbf{wr}_{\mathtt{t}}\,\mathtt{l}$, or (b) there exists a $\mathbf{wr}_{\mathtt{t}'}\,\mathtt{l} \in S'_1$ such that $t' > t$. If either of these conditions hold, however, $S$ would not be safe. Thus, by Lemma 1, we can permute $\mathrm{OP}^{\mathtt{l}_k}_{\mathtt{t}_k}$ with $\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}}$ to yield a new safe schedule $S_p = S'''.\mathrm{OP}^{\mathtt{l}_j}_{\mathtt{t}_j}.\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}}.\mathrm{OP}^{\mathtt{l}_k}_{\mathtt{t}_k}$. We can apply Lemma 1 again to $\mathrm{OP}^{\mathtt{l}_j}_{\mathtt{t}_j}.\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}}$, and so on, repeatedly shifting $\mathrm{OP}^{\mathtt{l}}_{\mathtt{t}}$ until a serial schedule is constructed.

# 4. IMPLEMENTATION

In the previous sections, we discussed our principal design goals in bringing futures into Java. The overarching intention of our design is to ensure that the observable behavior of a future-annotated program is not dependent on whether futures are evaluated synchronously (serially) or asynchronously (concurrently). The task of maintaining this *logical* serial order of operations in the presence of potentially concurrent updates to shared state is non-trivial. Our solution is to divide the entire program execution into a sequence of *execution contexts*. An execution context is a run-time structure that encapsulates a fragment of computation that is fully evaluated within a single thread. We define a total order (called a *context order*) over execution contexts that represents a logical serial execution order, and use this order to identify data dependencies between operations from different execution contexts. If futures are evaluated asynchronously, multiple execution contexts may update shared state concurrently, which in turn may violate the logical serial order of operations. An implementation must prevent this from happening by tracking all shared data operations and revoking execution contexts that violate this order. To help reduce the number of violations, execution contexts always operate on local versions (*ie*, copies) of shared objects which guarantee that out-of-order updates do not violate serial dependencies. The versioning mechanism is also used to enable revocations of execution contexts violating the logical serial execution order of operations. The remainder of this section provides further details.

Our prototype implementation is based on IBM's Jikes Research Virtual Machine (RVM) [3]. The Jikes RVM is a state-of-the-art Java virtual machine with performance comparable to many production virtual machines. It is itself written almost entirely in Java and is self-hosted (*ie*, it does not require another virtual machine to run). Java bytecodes in the Jikes RVM are compiled directly to machine code. The Jikes RVM's public distribution includes both a "baseline" and an optimizing compiler. The "baseline" compiler performs a straightforward expansion of each individual bytecode into a corresponding sequence of assembly instructions. The optimizing compiler generates high quality code due in part to sophisticated optimizations implemented at various levels of intermediate representation, and because it uses adaptive compilation techniques [4] to target code selectively for optimization. Our implementation targets the Intel x86 architecture.

## 4.1 Execution contexts

An execution context encapsulates a fragment of computation that is executed within a single *thread*. A thread defines a separate locus of control. It makes no guarantees, however, on the relationship between the operations it executes and those of other threads executing concurrently. In contrast, execution contexts impose strong safety constraints on the operations they execute. Specifically, concurrent evaluation of multiple execution contexts by different threads cannot lead to observable behavior different from serial evaluation of those contexts by a single thread. We refer to an execution context that represents a completed task (either a future or its continuation) as being *inactive*; contexts that are still under evaluation are considered *active*.

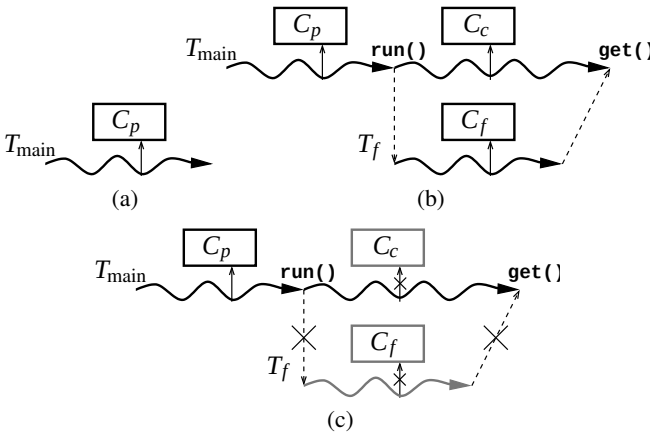



**Figure 6: Execution context creation**

The execution of a program begins within a *primordial context* evaluated within the main thread of computation. Consider what happens when a future is scheduled for evaluation – *ie*, its `run` method is executed. Logically, the code fragment encapsulated within a future executes *before* the code fragment following the call to the `run` method up to the point where the future is claimed by the `get` (the future's continuation). In order to preserve logical execution order, we create two new contexts: one associated with a thread used to evaluate the future – a *future context* evaluated within a freshly created thread; and one associated with the thread used to execute the future's continuation – a *continuation context* evaluated using the *same thread* as the primordial context. At this point we establish an execution order over these two execution contexts that reflects a serial order of execution in which the effects of the primordial context are visible to the future context whose effects are in turn visible to the continuation context. This ordering between future and continuation contexts is applied to all subsequently evaluated futures. The continuation of a future created within a (non-primordial) continuation context $C'$ executes within the same thread that executes $C'$, while the future itself executes within a freshly created thread. Once evaluation of all futures has successfully completed and they have been claimed (their respective `get` methods invoked), the continuation context also completes and the execution returns to the original state – *ie*, only the primordial execution context is active and all operations are performed within the main thread of computation.

As an example, consider execution of the program shown in Figure 4. A primordial context is created by the run-time system for execution of the `main` method. Invoking `concurrent` on the `Example` instance from this primordial context proceeds by creating two additional contexts at the point that `run` is invoked on the future `f`: one to execute the future and one to execute the continuation in which the call to `bar` occurs. At the point that `get` is invoked on the future, both of these contexts are discarded, and execution resumes in the primordial context. To reduce the overhead of creating both contexts and the fresh threads to evaluate future contexts, our implementation caches and re-uses both threads and execution contexts.

The scenario described above is illustrated in Figure 6 (wavy lines represent threads and boxes represent execution contexts). Initially, a primordial context ($C_p$) is created and bound to $T_{\text{main}}$, the thread evaluating the main method (Figure 6(a)). When a future is scheduled for execution (*ie*, its `run` method is invoked), two more contexts are created (Figure 6(b)): context $C_f$ to evaluate the future ($C_f$ is bound to $T_f$, a new thread used to execute the code encapsulated within the future), and context $C_c$ to evaluate the continuation of the future ($C_c$ is bound to the same thread as the primordial context $C_p$, in this case $T_{\text{main}}$). The execution of the program proceeds concurrently until the `get` method is invoked (the result computed by the future is then claimed) and then goes back to executing entirely within $T_{\text{main}}$, the main thread of computation. Note that at this point both contexts $C_f$ and $C_c$ as well as thread $T_f$ are discarded (Figure 6(c)) and can be cached for later re-use.

## 4.2 Preserving serial semantics

When two or more execution contexts execute concurrently, their operations may be arbitrarily interleaved and thus the semblance of serial execution may be violated. Consider two execution contexts: one representing a future ($C_f$) and one representing a continuation of this future ($C_c$). Under a (logical) serial order of execution, $C_f$ precedes $C_c$. If $C_f$ and $C_c$ execute concurrently, this order may be violated in one of two ways:

- $C_c$ *does not* observe the effect of an operation performed by $C_f$ (*eg*, a read in $C_c$ does not see modification of shared data by $C_f$), even though it *would have* observed this effect if $C_f$ and $C_c$ were executed serially. We call this a *forward dependency violation*.[1]

- $C_f$ *does* observe the effect of an operation performed by $C_c$ that could never occur if $C_f$ and $C_c$ were executed serially because $C_f$ would execute *fully* before $C_c$. We call this a *backward dependency violation*.

An example of schedules demonstrating both forward and backward dependency violations between $C_f$ and $C_c$, along with code snippets representing execution contexts appear in Figure 7. In Figure 7(a) the continuation represented by context $C_c$ should see the result of the write to `o.foo` performed by the future represented by context $C_f$. In Figure 7(b) the future represented by context $C_f$ should not see the result of the write to `o.bar` performed by the continuation represented by context $C_c$. Note that the notion of a dependency violation capture the same properties as the schedule safety rules from Section 3.1 (forward dependency violations are captured by the *csafe* rule and backward dependency violations are captured by the *fsafe* rule).

We prevent forward dependency violations by tracking all data accesses performed within execution contexts. We use per-context read and write *bit-maps* to record accesses to shared state. Each item of shared state (*ie*, object, array, or static variable) hashes to a bit in each map. How we use these bit-maps to detect violations is described below.

In the case of a forward dependency violation, the execution context responsible for the violation is revoked automatically (without programmer intervention) and restarted. Backward data dependency violations are prevented by *versioning* items of shared state. We preserve a copy-on-write invariant to ensure that each execution context updates its own private versions of shared items, preventing

[1] *Forward* in the sense that an operation from the "logical future" causes the violation.

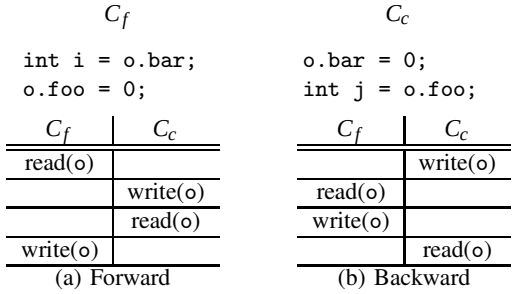



**Figure 7: Dependency violations**

it from seeing updates performed by execution contexts in its logical future. A detailed description of these mechanisms is presented below.

With the copy-on-write invariant we must make sure that subsequent reads occur to the copy and not to the original. To achieve this, we simply scan the copying context's thread stack whenever an item of shared state is copied, patching any references to the original version to refer instead to the copy – this is called *forwarding*. Whenever a reference is loaded from the heap we ensure that it is forwarded to the appropriate version. Thus, a context can never *use* a reference to the wrong version.

## 4.3 Tracking data accesses

We track data accesses using compiler-inserted read and write *barriers*: code snippets responsible for maintaining the meta-data required to detect dependency violations, and inserted by the compiler at the point where shared data access operations occur. The meta-data consists of the two bit-maps associated with each execution context: one for reads (*ie*, the *read-map*) and one for writes (*ie*, the *write-map*). Whenever a read operation is performed on an item of shared state (*ie*, object, array, or static variable), its hashed bit is set in the read-map. Hashes for objects and arrays are their natural hash value, while for a static variable it is its address (these addresses are constants established when the class containing the static variable is loaded). The same read-map is used for all shared items. Write-maps are similarly maintained within write barriers, though write barriers must always ensure that a new version is created on first write by a context to a given item.

Since reads significantly outnumber writes in most Java programs, reducing the number of read barriers is critical to achieving reasonable performance. Our implementation therefore trades off accuracy for efficiency in detecting dependency violations. Instead of placing barriers on all read accesses to shared items (*eg*, reading an integer field from an object), we assume that once a reference is read from the heap, the context reading it will eventually read from the object targeted by that reference. Thus, the read barrier is placed only on loads of *references* from the heap (*eg*, `getfield` or `arrayload` bytecodes in which the type of the field or element is a reference). In other words, we "pre-read" all objects to which a context holds references (when a context is initialized this means we must apply the pre-read barrier to all references in the current activation record). The pre-read optimization is applied only for objects and arrays to eliminate read barriers on them. All other accesses, including reads from static variables, and all writes to shared items incur the appropriate barrier.

Note that bit-maps are maintained only if there is more than one active context present in the system (*ie*, there is potential for concurrency and thus logical serial order violations). That is, barriers are responsible only for fetching the most recent version of an item

if only the primordial context is active. Bit-map maintenance is in reality optimized even further: the first of a series of execution contexts does not need to record its reads because versioning ensures they cannot be compromised by any concurrent writes. Thus, it does not need to maintain a read-map.

## 4.4 Execution context revocation

Actions performed by a context may violate the semblance of sequential execution because of a forward dependency violation. We now describe how such violations are handled. First, note that we defer detection of forward violations until the point where a context has finished executing (*ie*, when a future context completes its computation, or a continuation context invokes `get` on the associated future[2]). At that time, the context must wait for completion of contexts that logically precede it, before validating itself against their actions. This means forming the intersection of the current context's read-map with the write-maps of the preceding contexts. If there is overlap then the current context is revoked and transparently re-executed. Because a context operates over its own local versions of shared data, its updates become visible only when it successfully passes this forward violation check.

The implementation of revocation and re-execution for future contexts is simple. Because a future context is evaluated within a separate thread, we can simply terminate the thread, and re-start the future execution. Revoking and re-executing continuation contexts is more challenging. Our solution adapts the Jikes RVM exception handling sub-system to return control to the beginning of the continuation, using bytecode rewriting to save program state (values of local variables and the state of the Java operand stack) at the start of the continuation. Each method that invokes a future, at which point a continuation context begins, is wrapped with an exception scope that catches an internal `revoke` exception, restores the local state and transfers control to the beginning of the continuation context. The `revoke` exception is thrown internally by the runtime system, but the code to catch it (implementing re-execution) is injected into the bytecode. We also modify the compiler and runtime system to suppress generation (and invocation) of "default" exception handlers during the rollback operation. The "default" handlers include both `finally` blocks, and `catch` blocks for exceptions of type `Throwable`, of which all exceptions (including the `revoke` exception) are instances. Running these intervening handlers would violate the transparency requirement that a revoked execution context produces no side-effects.

Our current implementation does not preserve context state other than that of the method containing the continuation context, so futures that are invoked, but not claimed by the end of the method are implicitly claimed before the invoking method can return (*ie*, we wait for all spawned futures to complete their execution), even though the matching `get` operation is still to be invoked.

We illustrate how our system handles forward dependency violations using the code fragment and sample schedule from Figure 7(a). We assume that execution context $C_f$ (representing a future) is executed by (and thus bound to) thread $T_f$ and execution context $C_c$ (representing a continuation) to thread $T_c$. The entire scenario is illustrated in Figure 8, where wavy lines represent threads $T_f$ and $T_c$, and a circle represents object `o` (it is marked gray when updated). The execution contexts are linked together in order to allow execution contexts from the logical future to access the maps of contexts from the logical past. There is a read-map and write- map associated with each context (each map has three slots and we assume that object `o` hashes to the second slot).

[2]As described previously, futures and continuations are created "in pairs".

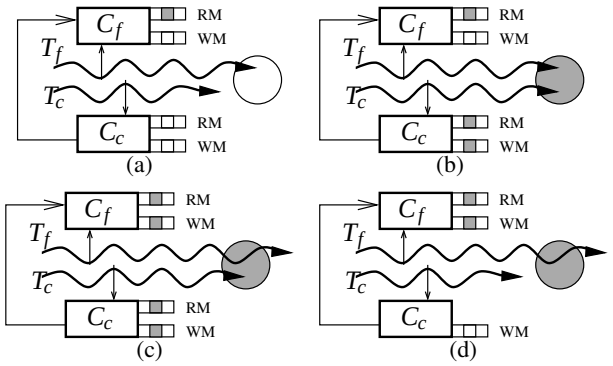



**Figure 8: Handling of a forward dependency violation.**

Execution starts with the future reading a field of object `o` (Figure 8(a)) and setting the appropriate slot in the read-map of its execution context $C_f$. The continuation then both reads and updates the same object, setting the read-map and write-map associated with its execution context $C_c$ (Figure 8(b)). Subsequently, the future writes to the field of object `o` and sets the write-map associated with its execution context $C_f$ (Figure 8(c)). At this point execution of the future is complete (no dependency violations that could cause its revocation are possible since there were no other concurrent contexts executing in its logical past). However, before the continuation can complete, it must check for forward dependency violations. This check fails since $C_c$'s read-map and $C_f$'s write-map overlap. The continuation is revoked and re-executed (Figure 8(d)). Note that after revocation, there are no more active execution contexts in its logical past. As a consequence, re-execution is guaranteed to succeed so maintaining its read-map is unnecessary.

## 4.5 Shared state versioning

We use versioning of shared state to avoid backward data dependency violations and prevent updates of shared data from being made prematurely visible to other threads in case of revocations. We only version objects when more than one execution context is concurrently active, since it is only then that concurrent shared state access may occur. Whenever an active execution context attempts to write to an object, array, or static variable, we create a private version of that item on which to perform the write. We handle object and array updates identically and use a similar procedure to handle updates to static variables. The code implementing the versioning procedure resides in the read and write barriers described in Section 4.3.

### *4.5.1 Object and array versioning*

Because objects and arrays are treated identically, we refer only to objects when describing the versioning procedure. We extend the header of every object with a forwarding pointer. At object allocation time, this forwarding pointer is initialized to `null`. As the program executes, subsequent versions are appended to a circular list rooted at the forwarding pointer of the original object (*ie*, the original object is the head and tail of this version list). Each version is tagged with the unique identifier of the context that created it. This enables each context to locate its version in the list. The versions are sorted under logical context order.

We now describe the implementation of read and write operations on objects as performed by an active execution context in the presence of versioning. If only one execution context is active (no concurrency), a read or write operation retrieves and accesses the

most recent version of the object (or the original object in case no versions of this object have been created). Otherwise, a more complicated access procedure is required.

#### 4.5.1.1 Reads.

A reference to a shared object $o$ referenced by a context $C$ must point to $o$'s most recent version with respect to $C$; in particular, $C$ must not access any version of $o$ that has been created by another context which occurs in $C$'s logical future. To implement this invariant, we traverse the versions list (in context order) and either load the reference for the version tagged by $C$, or the version corresponding to $C$'s most recent predecessor context. If a context $C'$ that occurs in $C$'s logical past has written to $o$, and $C$ reads the version corresponding to this write, a forward dependency violation exists and will be captured using the map tagging mechanism described earlier. Indeed, the only instance when a read by $C$ to a version written by $C'$ would be safe is precisely when, at the point the read occurs, $C'$ is already inactive.

The implementation of read operations is additionally complicated by the fact that read barriers are only executed at reference loads. Thus, in order for loads of primitive values to proceed correctly, we maintain an invariant that no existing reference on the thread stack belonging to context $C$ can point to a version created by any other context executing in $C$'s "logical" future. This invariant is relatively easy to maintain since the run-time system monitors all reference loads within read barriers. However, we must take special care to make sure that if a context creates a version, all references on the thread stack are updated correctly to point to this version (in other words, all reads performed by $C$ must observe $C$'s writes). We implement the invariant using a thread stack inspection mechanism described below.

#### 4.5.1.2 Writes.

When multiple execution contexts are active, all of them operate over their own local versions of shared data. In order to reduce the number of copies created, our implementation employs a copy-on-write strategy – a new version is created only when an execution context $C$ updates an object *for the first time*; we guarantee that all subsequent accesses by $C$ will refer to that version.

All object update operations (including writes to primitive fields) are mediated by write barriers. When $C$ performs an initial write to an object $o$, a local version of $o$ does not yet exist. It is therefore created and inserted at the appropriate position in the versions list rooted at $o$ to reflect context order. At this point, other references to the same object may exist on $C$'s thread stack. For example, $C$ might have previously read $o$, but not yet written to it. All such references must then be forwarded to point to the freshly created version of $o$ in order to avoid accessing stale versions. All versions of $o$ created by contexts in $C$'s logical past are considered stale with respect to $C$ if $C$ creates a new version of $o$.

Reference forwarding requires thread stack inspection as described below. Note that once the new version is created and all the references on the stack are forwarded, *all* the references on the stack throughout the *entire* execution of this context will always point to the right version (because subsequent reference loads are forwarded to the appropriate version). As a result, we avoid having to locate the correct version on the versions list when executing writes so long as a private copy exists. We only have to traverse the versions list upon version creation (when the object is first written). New versions are inserted at the appropriate place in the version list to maintain it in order.

All contexts maintain a list of their versions and use this list to purge revoked versions upon context revocation. Our prototype implementation does not currently purge stale versions from an object's versions list. Instead, we defer such cleanup to the garbage collector.

#### 4.5.1.3 Thread stack inspection.

We use a modified version of the thread stack inspection mechanism used by the garbage collector to support both pre-reading and forwarding of references on the stack. However, the essence of the mechanism remains the same. One of the major differences between the original stack inspection mechanism used during garbage collection and our modified version lies in the choice of the client using this mechanism. Garbage collection assumes that the stacks of inactive threads are being inspected. As a result, the entire execution state (including registers) of the inspected thread is available for inspection. In our system, it is the active thread that inspects its own state. We artificially create a "snapshot" of the current thread's execution state, execute the stack inspection routine, and restore the execution state to the point before the inspection routine was invoked. This snapshot procedure is implemented in assembler. The stack inspection routine either tags a read-map for every reference encountered (when pre-reading the stack) or forwards all references encountered to point to the correct version (when forwarding references during copy-on-write).

### 4.5.2 Versioning of static variables

In Jikes RVM static variables are stored in a global array called the JTOC. Static variables are versioned similarly to objects. A copy-on-write strategy is used, with a versions list holding per-context versions of static variables. Because we must version static variables of both primitive and reference type, we introduce the notion of a *version container*: a small object that *boxes* a value of the static variable into an object that can be put on the versions list.

Upon initial write to a static variable by the current context, a version container for the corresponding variable is created. The type of the slot in the JTOC representing this variable is then modified to indicate that its value has been copied aside to the list of version containers. For subsequent writes, the version created by the current context must be retrieved and the value it contains updated. When reading the value of a static variable, the current execution context attempts to locate the appropriate version container on the versions list similarly to retrieving an object version (it is either the container created by the current context or the one directly preceding its logical position in the list).

Indirections in the JTOC are lazily collapsed after all futures have been successfully evaluated and the program reverts to executing within a single (primordial) context. From this point on, only the most recent value of each static variable can ever be used.

## 5. EXPERIMENTS

Our experiments with safe futures for Java explore their performance on both standard benchmarks and a synthetic benchmark. In both cases, we use futures in a straightforward rewrite of initially sequential benchmark programs. The standard benchmarks are drawn from the Java Grande benchmark suite. We choose a subset of naturally parallelizable benchmarks, namely `series`, `sparse`, `crypt` and `mc`.

The synthetic benchmark is intended to expose the performance of our implementation across a range of benchmark parameters, such as read/write ratio and degree of shared access. The synthetic benchmark is based on the OO7 object database benchmark suite [11], modified to use futures in a parallel traversal of the OO7 design database.

For all benchmarks, we also run their original sequential version

on the *unmodified* Jikes RVM, and use this as a baseline to which we normalize for comparison with their future-enabled parallel versions.

## 5.1 Experimental platform

Our implementation uses version 2.3.4+CVS (with 2005/06/23 13:35:22 UTC timestamp) of Jikes RVM for both the configuration that is futures-enabled and the baseline configuration against which we compare. Jikes RVM is configured with the defaults for the Intel x86 platform, using the adaptive compiler framework.

We run each benchmark in its own invocation of the Jikes RVM, repeating the benchmark six times in each invocation, and discarding the results of the first iteration, in which the benchmark classes are loaded and compiled, to elide the overheads of compilation. We report mean execution times, with 90% confidence intervals, to illustrate the degree of variation.

Our hardware platform is an 700MHz Intel Pentium III symmetric multi-processor (SMP), with four CPUs, and 2GB of RAM running Linux kernel version 2.4.20-20.9 (RedHat 9.0). Our parallel executions run four futures simultaneously on the SMP, though we note that such runs create multiple sets of four futures for each iteration of the benchmark, so a series of futures are created in each run. This permits utilization of our context-caching mechanisms.

## 5.2 Benchmarks

As mentioned earlier, we draw upon benchmarks from the Java Grande suite, as well as the OO7 synthetic design database benchmark. The former are representative of ideal candidate applications for parallelization using futures. The latter is less amenable to parallelization due to the density of the benchmark data structures and degree of sharing among them. Nevertheless, OO7 represents a benchmark in which meaningful parameters can be varied easily to demonstrate their impact on the performance of our futures implementation.

### 5.2.1 Java Grande

Each of the selected Java Grande benchmarks was chosen for being straightforwardly parallelizable. They perform substantial computations over elements stored in Java arrays or in Java vectors, where access to the data structures is encoded into loops over the respective elements. We parallelized these benchmarks by substituting futures for subsets of the loop iterations similarly to the way these benchmarks have been parallelized for distributed execution via a Java message-passing interface (MPJ) [41]. For the benchmarks that use the arrays, this rewriting also includes partitioning the arrays into subarrays in order to capture locality (such transformations were also used with MPJ), and because the conflict detection mechanisms described earlier function at per-array granularities, rather than for fragments of arrays. The `series` benchmark performs Fourier coefficients computation, `sparse` multiplies an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure and `crypt` performs IDEA (International Data Encryption Algorithm) encryption and decryption and `mc` benchmark is an implementation of a Monte Carlo Simulation.

We believe benchmarks like these are prime candidates for parallelization using futures. Note, however, that even though they could be rewritten to use futures with only small changes to their source code, and straightforward partitioning of their data, rewriting the OO7 benchmark was even simpler – no data partitioning was required – and involved modifying only the top-level control loop (detailed below).

### 5.2.2 The OO7 benchmark

| Component | Number |
|---|---|
| Modules | $M+1$, for $M$ futures |
| Assembly levels | 7 |
| Subassemblies per complex assembly | 3 |
| Composite parts per assembly | 3 |
| Composite parts per module | 5000 |
| Atomic parts per composite part | 20 |
| Connections per atomic part | 3 |
| Document size (bytes) | 2000 |
| Manual size (bytes) | 100000 |

**Table 1: Component organization of the OO7 benchmark**

The OO7 benchmark suite [11] provides a great deal of flexibility for benchmark parameters (*eg*, database structure, fractions of reads/writes to shared/private data). The multi-user OO7 benchmark [10] allows control over the amount of contention for access to shared data. By varying these parameters we are able to characterize the performance of safe futures over a mixed range of workloads.

The OO7 benchmarks operate on a synthetic design database, consisting of a set of *composite parts*. Each composite part comprises a graph of *atomic parts*, and a `document` object containing a small amount of text. Each atomic part has a set of attributes (*ie*, fields), and is connected via a bi-directional association to several other atomic parts. The connections are implemented by interposing a separate connection object between each pair of connected atomic parts. Composite parts are arranged in an *assembly* hierarchy; each assembly is either made up of composite parts (a *base* assembly) or other assemblies (a *complex* assembly). Each assembly hierarchy is called a *module*, and has an associated *manual* object consisting of a large amount of text. Our results are all obtained with an OO7 database configured as in Table 1.

Our implementation of OO7 conforms to the specfication of the standard OO7 database. Our traversals are a modified version of the multi-user OO7 traversals. A traversal chooses a single path through the assembly hierarchy and at the composite part level randomly chooses a fixed number of composite parts to visit (the number of composite parts to be visited during a single traversal is a configurable parameter). When the traversal reaches the composite part, it has two choices:

1. Do a *read-only* depth-first traversal of the atomic part subgraph associated with that composite part; or
2. Do a *read-write* depth-first traversal of the associated atomic part subgraph, swapping the *x* and *y* coordinates of each atomic part as it is visited.

Each traversal can be done beginning with either a *private* module or a *shared* module. The parameter's of the workload control the mix of these four basic operations: read/write and private/shared. To foster some degree of interesting interleaving and contention in the case of concurrent execution, our traversals also take a parameter that allows extra overhead to be added to read operations to increase the time spent performing traversals.

The top-level execution of our sequential OO7 benchmark operates as shown in Figure 9(a). It performs $I$ benchmark iterations, each benchmark iteration comprises $M$ sets of traversals in which the private module ranges from module 1 to module $M$, module $M+1$ is used as the shared module, and the parameter $p$ controls the mix of operations performed by the traversals.

The top-level execution of our futures-enabled OO7 benchmark operates as shown in Figure 9(b). It performs $I$ benchmark itera-

```
for (i = 1; i <= I; i++)
  for (m = 1; m <= M; m++)
    traversals(m, p);
```

(a) Sequential OO7 benchmark

```
for (i = 1; i <= I; i++) {
  for (m = 1; m <= M; m++)
    f[m] = future(traversals(m, p));
  for (m = 1; m <= M; m++)
    f[m].get();
}
```

(b) Parallel OO7 benchmark

**Figure 9: Top-level loop of the OO7 benchmark**

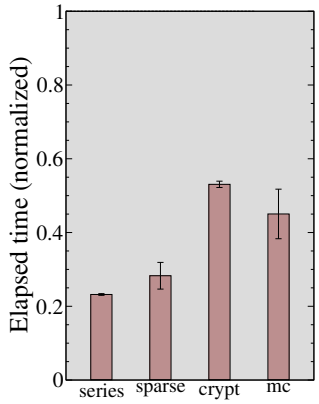



**Figure 10: Java Grande: Elapsed time (normalized)**

tions, each benchmark iteration comprises $M$ futures, each of which performs a set of traversals operating on a distinct private module $m$, module $M+1$ is used as the shared module, and the parameter $p$ controls the mix of operations performed by the traversals.

We seed the traversals with the same random seed in both the sequential and futures-enabled executions of the benchmark, such that both versions perform identical workloads.

## 5.3 Results

We present results for the Java Grande benchmarks first, to indicate the behavior of futures under ideal circumstances. OO7 is more demanding, but also more tuneable, revealing the underlying performance characteristics of our implementation.

### *5.3.1 Java Grande results*

Figure 10 reports the elapsed time for execution of the future-enabled versions of the Java Grande benchmarks, normalized against the average elapsed time for execution of the unmodified sequential benchmarks running on the unmodified Jikes RVM. Times are arithmetic means of the 5 hot runs of each benchmark, with 90% confidence intervals revealing minimal variability. Recall that we parallelize the benchmarks using four futures. Thus, observe that speedups range from perfect (or even slightly super-linear – 4× for `series`), to a little less than 2× speedup for `crypt`. We believe that the reason for the super-linear speedup for `series` is due to improved locality as a result of the array partitioning.

### *5.3.2 OO7 results*

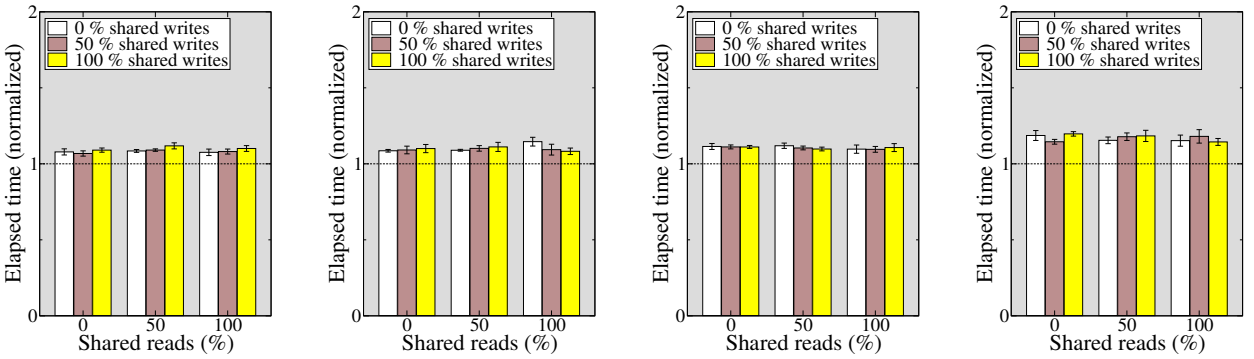
We report results for two basic versions of OO7, one for a database containing only 2 ($M = 1$) modules and one for a database comprising 5 ($M = 4$) modules. Again, we compare the future-enabled parallel versions against the sequential version of the benchmark. We vary the ratio of writes to reads performed within each set of traversals as 4%, 8%, 16% and 32% writes (96%, 92%, 84%, and 68% reads, respectively), in an attempt to model workloads with mutation rates ranging from low to moderate. We also vary the ratio of shared/private accesses for each mix of reads/writes as 0%, 50% and 100%. Thus, for 4% writes, 50% shared, a set of 100 traversals will on average perform 2 read-write traversals to shared data, 2 read-write traversals to private data, 48 read-only traversals on shared data, and 48 read-only traversals on private data.

With just 2 ($M = 1$) modules, both the original and future-enabled versions are inherently sequential, since the degree of future-enabled parallelism is equal to $M$ for a database containing $M+1$ modules. Moreover, because only one future is ever active revocation cannot occur. Thus, the comparison for $M = 1$ yields a measure of the fundamental overheads in our system for creating and claiming futures (and indirectly the effectiveness of our context-caching mechanisms), for the read and write barriers used to track accesses, and for versioning. The elapsed time results, normalized against the sequential version running on the unmodified Jikes RVM, are presented in Figure 11. These reveal a per-future performance hit of 8-12% for 4% writes. As write ratios increase, we see overheads of 15-20% for 32% the write ratio. Figure 12 graphs the number of versions created per benchmark iteration, showing that the number of versions created increases with sharing and the write ratio.

Of course, for more futures, this performance hit may come to dominate. Some of the overhead results from the lack of efficient support in Jikes RVM for caching of thread state (*eg*, stacks) from one thread activation to another. Thus, spawning a future is relatively expensive. Our context caching mechanisms ameliorate some of the overhead, but there is much more that could be done along the lines of Mul-T [30, 33]. Still, our overheads are low enough to justify the use of safe futures for a range of applications, as the Java Grande results already illustrate.

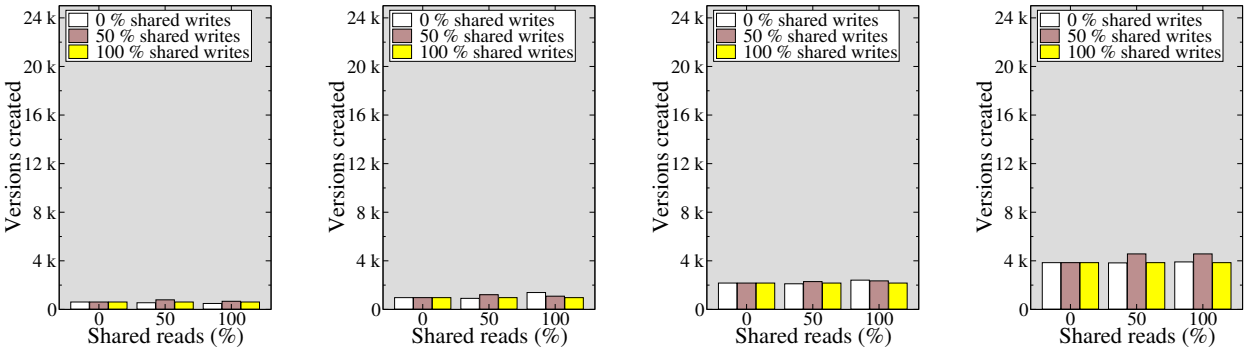
Adding concurrency yields opportunity for parallelism, as illustrated in the results for OO7 using four futures, shown in Figure 13. With four futures executing concurrently there is the possibility of revocation, which we graph in Figure 14. Without sharing there are no revocations. Thus for the unshared executions we see uniform gains of 52-56% across the range of write ratios, as expected. The performance gains vary depending on the configuration; even at 32% write ratio with 100% sharing we still observe a performance benefit of about 25% (Figure 13(d)). In all configurations, the revocations seem to impact performance significantly, since their rise is correlated with increased sharing, as well as write ratio (see Figure 14). The increase in versions created (Figure 15) also affects execution times – as write ratios increase, elapsed times in Figure 13 also increase slightly even for configurations where no revocations are observed.

The cost of creating versions constitutes part of the "base" overhead common across all configurations, though clearly non-existent in the sequential version of the benchmark. Another large base overhead results from executing large numbers of read barriers. We observe on average 63 million read barriers (30 million for objects, 18 million for arrays and 15 million for static variables) per benchmark iteration (these numbers remain much the same across all configurations). This indicates that our initial decision to minimize the number of barriers by inserting them only at reference loads was prescient. We also observe a large number of write barriers – 16 million on average per benchmark iteration (6.5 million for objects, 9.5 million for arrays, and a negligible number for static




(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

**Figure 11: OO7 with 1 future: Average elapsed time per iteration (normalized)**




(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

**Figure 12: OO7 with 1 future: Versions created per iteration**

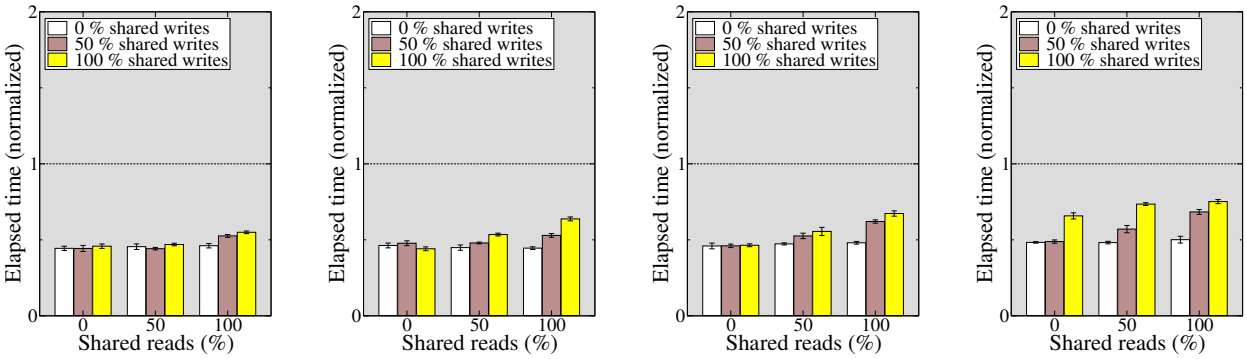
variables). The number of write barriers for objects increases as the number of writes to shared objects grows across different configurations. We are particularly penalized by the number of static variable accesses for this implementation of OO7, which uses them to capture the traversal parameters. In general, static analyses such as escape analysis also have much grist for the mill here, in optimizing away unnecessary barriers [12, 6, 7]. We note, however, that for OO7 such analyses are unlikely to have much impact, because all futures operate over a single recursively-defined data structure. Nonetheless, even without the benefit of advanced compiler optimizations, the performance of our implementation using just run-time optimizations is encouraging.

## 6. RELATED WORK

Concurrent programming using threads that operate over shared state can be difficult and error-prone. Besides the algorithmic issues involved in structuring programs to exploit multiple threads of control, access to shared state must be protected to ensure safety. Typically, such protection is expressed using synchronization mechanisms such as locks or abstractions like monitors, built on locks, that allow reasoning about control within the protected region in terms of serial execution of the threads that access it. Unfortunately, this programming model is highly invasive: since there is often little coupling between threads and the data they access, safety demands often dictate that *any* object be protected which potentially might be shared. Notwithstanding the negative impact on performance, restructuring programs to guarantee safety can lead to increased program complexity. On the other hand, underspecifying synchronization can lead to critical safety violations such as race conditions. In some cases, compiler optimizations can elide unwanted synchronization [1, 2, 7, 42] to alleviate performance issues. When synchronization is underspecified, tools can be employed to detect potential data races that may result. These tools can be expressed using type systems [17, 8], such as ownership types [14] to verify the absence of data races and deadlock. Other tools such as Eraser [39] employ dynamic techniques to check for races in programs [34, 32, 43]. There have also been attempts to leverage static analyses to reduce overheads and increase precision of purely dynamic implementations [13, 44].

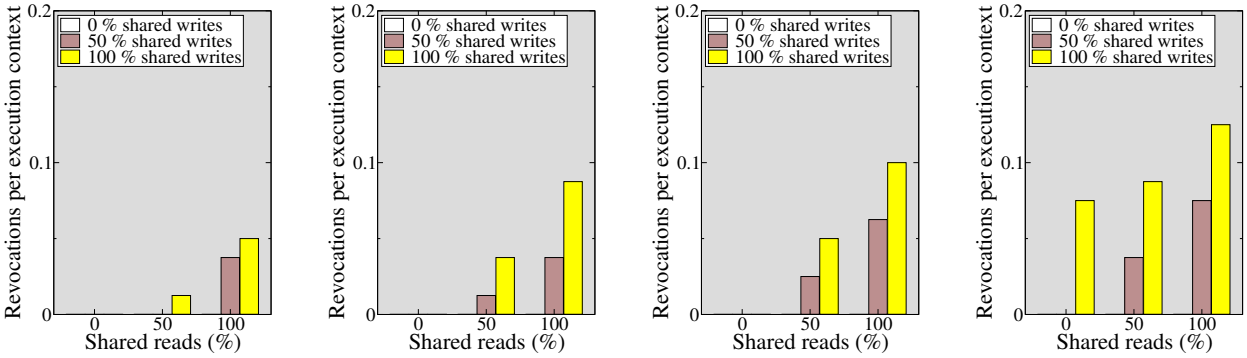
Our main contribution in this paper is a semantics and implementation of *safe* futures [21], a simple easily-understood abstraction for injecting concurrency into sequential programs. Unsafe futures when incorporated into programs that make heavy use of imperative features, can exhibit behavior inconsistent with their original intent. To ensure that annotating a program with futures does not violate expected data dependencies and thus unexpectedly change program behavior, we define a compiler and run-time infrastructure that employs object versioning and task revocation techniques to identify safety violations and remedy program executions when such violations are detected.

In this vein, our motivation is similar to recent proposals that have argued in favor of higher-level abstractions that enforce desirable properties on concurrent programs such as atomicity [19, 18, 22] or transaction-based isolation [23, 40, 45] without requiring the low-level (and thus potentially error-prone) operational reasoning demanded by locks. Other approaches include lock-free data structures [37, 26] and transactional memory [25]. These efforts share goals similar to ours insofar as they attempt to provide alternatives to lock-based abstractions for concurrent programming that preserve desirable safety properties, although the techniques



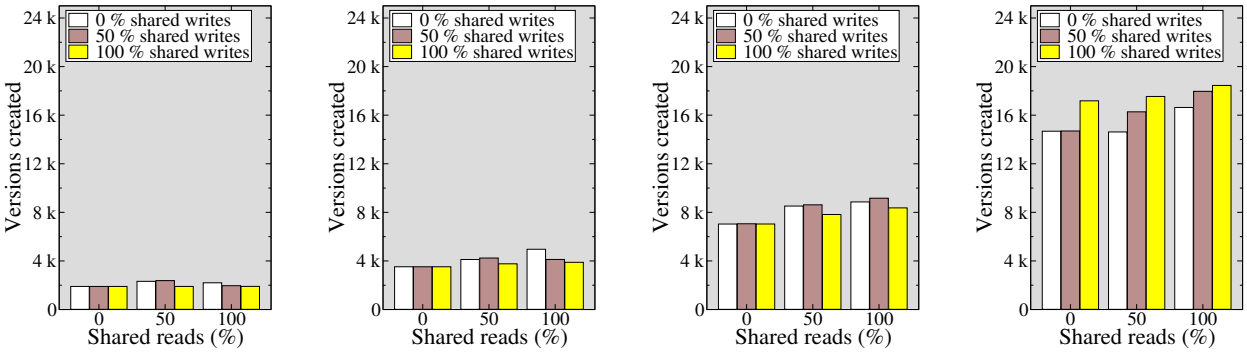

(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

**Figure 13: OO7 with 4 futures: Average elapsed time per iteration (normalized)**




(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

**Figure 14: OO7 with 4 futures: revocations per iteration**




(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

**Figure 15: OO7 with 4 futures: Versions created per iteration**

employed are different in substantial and obvious ways.

The semantics of futures [15, 16] and their implementation [30, 33] have been well-studied in the context of functional languages like Scheme [29]. Abstractions similar to futures (*eg*, *promises* [31]) have also been proposed for statically-typed imperative languages, but we are unaware of any previous effort that ensures their injection into a sequential stateful program is transparent with respect to preservation of data dependencies that exist in the original.

More recently, Pratikakis *et al* [36] present a static analysis to allow Java programs to use futures without requiring wholesale changes to the program to satisfy type restrictions. Their analysis tracks how a future flows through a program, and injects coercions that perform a claim operation on the future at points where the value yielded by the future, rather than the future itself, is required. The analysis uses qualifier inference to track how futures are used. Our goals are similar in spirit to this work in that both attempt to treat futures as a transparent concurrency mechanism. However, these two efforts are unrelated in their focus. Unlike our design and implementation, Pratikakis *et al* make no guarantees that the evaluation of a future does not introduce behavior inconsistent with the sequential program from which it was derived. By the same token, our notion of transparent execution does not extend automatically to injecting claims where necessary: a program that fails to claim a future when required may still lead to transparency violations of the kind described by Pratikakis *et al* in our implementation. We believe both forms of transparency are important. Although we expect that futures are primarily useful for spawning concurrent tasks that exhibit relatively little to modest sharing, it is nonetheless critical that safety violations be detected when they do occur.

The ParaTran project [28] was an attempt to parallelizing sequential Lisp programs in presence of side-effects. Like our implementation of safe futures, ParaTran used optimistic concurrency control techniques to monitor data accesses performed by concurrently executing tasks and to revoke fragments of computation after detecting violations of the (logical) serial execution order. To the best of our knowledge however, ParaTran has never been implemented to run on a real system and the available simulation results do not include the potential costs of such an implementation. One of the major goals of our work was a thorough performance evaluation of an implementation based on a realistic language execution environment.

Another approach to parallelizing sequential programs in the presence of side-effects has been explored in the context of the Jade parallel programming language [38]. A Jade programmer is responsible for delimiting code fragments (tasks) that could be executed concurrently and explicitly specifying invariants describing how different tasks access shared data. The run-time system is then responsible for exploiting available concurrency and verifying data access invariants in order to preserve the semantics of the serial program. Violations of data access invariants result in run-time errors.

# 7. CONCLUSIONS

In this paper, we have presented a detailed study of *safe* futures for Java. Futures provide a simple API for concurrent programming that allows a concurrent program to be constructed often through only a small rewrite of a sequential one. Unfortunately, futures as currently specified in Java are not treated as a semantically transparent annotation, thus significantly weakening their utility. Programmers who use futures must reason about subtle interaction among future-encapsulated computations, in much the same way they must reason about the interaction of threads in a typical multi-threaded Java program. Safe futures obviate the need for such reasoning by guaranteeing their injection into a sequential program does not violated any existing data dependencies. Furthermore, the cost of providing this added level of safety is not prohibitive. The evaluation of our prototype implementation indicates that futures can be used to exploit concurrency even for applications with modest mutation rates on shared data.

Our implementation leverages two general techniques that may have wider applicability: object versioning and task revocation. For example, object versioning can be used to implement a general checkpointing facility since each version is tagged with a context, a general dynamic region of code. Task revocation is the basis for implementing optimistic concurrency-based transaction implementations [22, 23, 45] for Java, and the validation mechanism used to determine if safety violations between a future and its continuation have occurred can easily be generalized to capture violations of properties such as atomicity or isolation for multithreaded applications that use language-level transactions or that build upon transactional memory [24].

## Acknowledgments

This work is supported by the National Science Foundation under grants Nos. CCR-9711673, STI-0334141, IIS-9988637, and CCR-0085792, by the Defense Advanced Research Program Agency, and by gifts from Microsoft, IBM, Sun Microsystems, and NEC. We also thank the reviewers for their detailed comments and suggestions.

# 8. REFERENCES

[1] ALDRICH, J., CHAMBERS, C., SIRER, E. G., AND EGGERS, S. J. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the International Static Analysis Symposium* (Venice, Italy, Sept.), A. Cortesi and G. Filé, Eds. vol. 1694 of *Lecture Notes in Computer Science*. Springer, 1999, pp. 19–38.

[2] ALDRICH, J., SIRER, E. G., CHAMBERS, C., AND EGGERS, S. J. Comprehensive synchronization elimination for Java. *Science of Computer Programming 47*, 2-3 (2003), 91–120.

[3] ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M., SHEPHERD, J. C., AND SMITH, S. Implementing Jalapeño in Java. In OOPSLA'99 [35], pp. 314–324.

[4] ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Minneapolis, Minnesota, Oct.). *ACM SIGPLAN Notices 35*, 10 (Oct. 2000), pp. 47–65.

[5] BERNSTEIN, A. Program analysis for parallel processing. *IEEE Transactions on Computers 15*, 5 (Oct. 1966), 757–762.

[6] BLANCHET, B. Escape analysis for object-oriented languages: Application to java. In OOPSLA'99 [35], pp. 20–34.

[7] BOGDA, J., AND HÖLZLE, U. Removing unnecessary synchronization in Java. In OOPSLA'99 [35], pp. 35–46.

[8] BOYAPATI, C., LEE, R., AND RINARD, M. C. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Seattle, Washington, Nov.). *ACM SIGPLAN Notices 37*, 11 (Nov. 2002), pp. 211–230.

[9] BURKE, M. G., CHOI, J.-D., FINK, S. J., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Java Grande Conference* (San Francisco, California, June). 1999, pp. 129–141.

[10] CAREY, M. J., DEWITT, D. J., KANT, C., AND NAUGHTON, J. F. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, Oct.). *ACM SIGPLAN Notices 29*, 10 (Oct. 1994), pp. 414–426.

[11] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The OO7 benchmark. In *Proceedings of the ACM International Conference on Management of Data* (Washington, DC, May). *ACM SIGMOD Record 22*, 2 (June 1993), pp. 12–21.

[12] CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. Escape analysis for Java. In OOPSLA'99 [35], pp. 1–19.

[13] CHOI, J.-D., LEE, K., LOGINOV, A., O'CALLAHAN, R., SARKAR, V., AND SRIDHARAN, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Berlin, Germany, June). *ACM SIGPLAN Notices 37*, 5 (May 2002), pp. 258–269.

[14] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, Oct.). *ACM SIGPLAN Notices 33*, 10 (Oct. 1998), pp. 48–64.

[15] FLANAGAN, C., AND FELLEISEN, M. The semantics of future and its use in program optimizations. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (San Francisco, California, Jan.). 1995, pp. 209–220.

[16] FLANAGAN, C., AND FELLEISEN, M. The semantics of future and an application. *J. Funct. Program. 9*, 1 (2005), 1–31.

[17] FLANAGAN, C., AND FREUND, S. N. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Vancouver, Canada, June). *ACM SIGPLAN Notices 35*, 6 (June 2000), pp. 219–232.

[18] FLANAGAN, C., AND FREUND, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages*

(Venice, Italy, Jan.). 2004, pp. 256–267.

[19] FLANAGAN, C., AND QADEER, S. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation* (New Orleans, Louisiana, Jan.). 2003, pp. 1–12.

[20] FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1998, pp. 171–183.

[21] HALSTEAD, JR., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 501–538.

[22] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Anaheim, California, Nov.). *ACM SIGPLAN Notices 38*, 11 (Nov. 2003), pp. 388–402.

[23] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* (Boston, Massachusetts, July). 2003, pp. 92–101.

[24] HERLIHY, M. P., AND MOSS, J. E. B. Lock-free garbage collection on multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 3*, 3 (May 1992), 304–311.

[25] HOSKING, A. L., AND MOSS, J. E. B. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Washington, DC, Sept.). *ACM SIGPLAN Notices 28*, 10 (Oct. 1993), pp. 288–303.

[26] JENSEN, E. H., HAGENSEN, G. W., AND BROUGHTON, J. M. A new approach to exclusive data access in shared memory multiprocessors. Tech. rep., Lawrence Livermore National Laboratories, 1987.

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

[28] KATZ, M. Paratran: A transparent, transaction based runtime mechanism for parallel execution of scheme. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1989.

[29] KELSEY, R., CLINGER, W. D., AND REES, J. Revised$^5$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998), 26–76.

[30] KRANZ, D., HALSTEAD, JR., R. H., AND MOHR, E. Mul-T: A high-performance parallel Lisp. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Portland, Oregon, June). *ACM SIGPLAN Notices 24*, 7 (July 1989), pp. 81–90.

[31] LISKOV, B., AND SHRIRA, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June). *ACM SIGPLAN Notices 23*, 7 (July 1988), pp. 260–267.

[32] MELLOR-CRUMMEY, J. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, Nov.). 1991, pp. 24–33.

[33] MOHR, E., KRANZ, D. A., AND HALSTEAD, JR., R. H. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (Nice, France, June). 1990, pp. 185–197.

[34] O'CALLAHAN, R., AND CHOI, J.-D. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, June). 2003, pp. 167–178.

[35] *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Nov.). *ACM SIGPLAN Notices 34*, 10 (Oct. 1999).

[36] PRATIKAKIS, P., SPACCO, J., AND HICKS, M. W. Transparent proxies for java futures. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, Oct.). *ACM SIGPLAN Notices 39*, 10 (Oct. 2004), pp. 206–223.

[37] RAJWAR, R., AND GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, Oct.). *ACM SIGPLAN Notices 37*, 10 (Oct. 2002), pp. 5–17.

[38] RINARD, M. C., SCALES, D. J., AND LAM, M. S. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer 26*, 6 (1993), 28–38.

[39] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (Nov. 1997), 391–411.

[40] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Canada, Aug.). 1995, pp. 204–213.

[41] SMITH, L. A., BULL, J. M., AND OBDRZÁLEK, J. A parallel Java Grande benchmark suite. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Denver, Colorado, Nov.). 2001, p. 8.

[42] UNGUREANU, C., AND JAGANNATHAN, S. Concurrency analysis for Java. In *Proceedings of the International Static Analysis Symposium* (Santa Barbara, California, Jun./Jul.), J. Palsberg, Ed. vol. 1824 of *Lecture Notes in Computer Science*. 2000, pp. 413–432.

[43] VON PRAUN, C., AND GROSS, T. R. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Tampa, Florida, Oct.). *ACM SIGPLAN Notices 36*, 11 (Nov. 2001), pp. 70–82.

[44] VON PRAUN, C., AND GROSS, T. R. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (San Diego, California, June). 2003, pp. 115–128.

[45] WELC, A., JAGANNATHAN, S., AND HOSKING, A. L. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming* (Oslo, Norway, June), M. Odersky, Ed. vol. 3086 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004, pp. 519–542.