Enhancing the JR Concurrent Programming Language with New Java 5.0 Features

By

HIU NING CHAN
B.S. (University of California, Davis) 2004

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Ronald A. Olsson, Co-Chair

Professor Aaron W. Keen, Co-Chair

Professor Matthew K. Farrens

Committee in charge

2005

To my grandma,

who passed away just before I started to write this thesis,

and is forever in my heart.

Acknowledgments

First, I would like to take this unique opportunity to sincerely thank one of the co-chairs of my thesis committee, Professor Ronald Olsson, who is the best professor I have ever met. He is a responsible, observant, fair, organized, demanding, but yet friendly mentor. He is also an active supporter of conservation since he was willing to take me back up from the CS graduate admission's trash can and recycled me into a good candidate for graduate school. I appreciate his patience when reading my lengthy, boring, and problematic thesis drafts. I can not count the number of times he explained to me about the difference between "that" and "which", nor do I remember how many times he reminded me on articles. After all, I learned a lot from him not only on computer science and English writing, but also on ways of doing things. He is one of the most critical factors for me not to consider UCLA's offer of masters program. The completion of this thesis strongly proves that I had made the correct decision. I enjoyed my time working with him and I will miss him.

Second, I would like to thank another co-chair of my thesis committee Professor Aaron Keen. I have only seen him in person twice and we communicate electronically for the rest of the time, nevertheless, he gave me a lot of guidelines and technical assistance on the JR implementation ever since I worked on my undergraduate honors thesis. His ideas are inspiring, innovative, and resourceful.

Third, I would like to thank my thesis committee member Professor Matthew Farrens. Although I do not have a lot of personal interaction with him, he was prompt to be my thesis committee member after having me in two of his classes. He is a nice and fun person to work with. He gave me a lot of insightful comments on this thesis.

Another important person I must thank is William Au Yeung. This research would take me at least one year to complete without his help on porting some of the JR features and ideas on simplifying operations using generics. I wish him well in the

University of Michigan, Ann Arbor.

I am indebted to all former and current contributors of the JR Group. I treasure the time we spent in weekly group meetings where we had fruitful discussions and came up with many motivating suggestions. A special thank you is dedicated to Erik Staab who was the first to propose the original idea on porting JR to Java 5.0.

Credits also go to my friend Steven Chau. He was the person who told me that Professor Olsson was taking undergrads to work on his research projects. Without him, I would probably be remained in the CS graduate admission's trash can forever.

Nija Shi is the next to thank on my list. We do not meet each other a lot, but we always have a wonderful and interesting conversation each time we chat in the research lab. She is nice enough to let me share her lab computer Sky so that I can have a fast machine to work with. She was the one who introduced VNC to me so that I can work on my research comfortably at home. She also provided a lot of help on Latex which saves me hours on figuring out how to format my thesis in MS Word.

Next, I would like to express my gratitude to my parents for their unconditional love and caring. They gave me the freedom to choose majoring in CS when I have totally no idea on how to use a computer to check my e-mail. They tolerated my crazy schedule where day and night were totally inverted. They remained calm and supportive when I told them I fell into the CS graduate admission's trash can. I hope this thesis serves as a way for me to pay them back.

A genuine thank you to Edward Chen for being my long term loyal supporter and partner. He walked through the college journey with me and shares all the ups and downs throughout the years. My life would be totally different without him. Getting to know him is my best fortune.

Furthermore, I appreciate my best friends Betty Chen, Billy Man, Max Mai, and Lily Tse for being fantastic colleagues and companions. They provided me years of entertainment since we know each other, which keeps me hyper in this lonely little green town, Davis. They are also good listeners of my complaints and good comforters when I am down. They are always in my mind.

Lastly, I would like to say thank you to everyone else who encouraged and assisted me when applying for graduate admission and who has touched my life but I did not mention his or her name here.

Contents

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

2 Background **4**

2.1 New Java Language Features 4

2.1.1 Generics 4

2.1.2 Enhanced For Loop (For-Each Loop) 5

2.1.3 Autoboxing / Unboxing 6

2.1.4 Typesafe Enums 7

2.1.5 Varargs 7

2.1.6 Static Import 8

2.1.7 Metadata 9

2.2 New Java Implementation Feature 9

2.2.1 Remote Method Invocation (RMI) 10

2.3 Java Compiler Design 10

3 New JR Features **13**

3.1 New JR Language Features 13

3.1.1 Quantifiers 13

3.1.2 Operations 17

3.1.3 Process 18

3.1.4 Virtual Machines 20

3.2 New JR Implementation Feature 20

3.2.1 Remote Method Invocation (RMI) 20

4 Implementation **21**

4.1 Merging Scheme 21

4.2 Schedule of Tasks 22

4.3 Verification 23

4.4 Bug Fixes 24

4.4.1 Bugs in Java 5.0 24

4.4.2 Bugs in JR 1.00061 26
4.5 Known Bug 26
4.5.1 Known Bug in JR 2.00001 27

5 Performance **29**
5.1 Java Performance 29
5.1.1 Compilation Time 30
5.1.2 Execution Time 30
5.2 JR Performance 31
5.2.1 Compilation Time 32
5.2.2 Execution Time 34

6 Implementation Optimization **37**
6.1 Generated Code Optimization 37
6.2 Implementation Problems 39
6.2.1 Inni Operation Invocations 39
6.2.2 View Statement 40
6.3 Known Bugs 41
6.3.1 Array of Capabilities 41
6.3.2 Generic Capability 42
6.4 JR 2.00005 Performance 42

7 Conclusion and Future Work **46**

A Java Benchmarks Description **48**

B Java Performance Statistics **51**

C JR Benchmarks Description **54**

D Parameters Used in Running JR Benchmarks **57**

E JR 2.00001 Performance Statistics **59**

F JR 2.00005 Performance Statistics **66**

Bibliography **71**

List of Figures

2.1 Remove 4-letter words from Collection **c** using Java 1.4. Elements of **c** must be **Strings**. 4
2.2 Remove 4-letter words from Collection **c** using Java 5.0. 5
2.3 Iterating over a collection using an iterator. 5
2.4 For-each loop in Java 5.0. 5
2.5 Wrapping a primitive type using a wrapper class and then extracting the primitive type from the wrapper using Java 1.4. 6
2.6 Autoboxing and Unboxing in Java 5.0. 6
2.7 Creating an array of arguments prior to invoking a method that takes variable number of arguments using Java 1.4. 8
2.8 Declaring and invoking a method that takes variable number of arguments using Java 5.0. 8

3.1 Example of new process quantifiers. 14
3.2 Example of the new process quantifiers with a such-that expression. . 15
3.3 Example of the new inni statement quantifiers with such-that expressions. 16
3.4 Example of the new co statement quantifiers with a such-that expression. 17
3.5 Example of an operation with varargs. 18
3.6 Example of an operation with generics. 19
3.7 Example of an operation with generics. 19

4.1 Example of a capability with generics. 28

6.1 Source program illustrating the effect of generated code optimization. 38
6.2 Source program illustrating the problem on signature mismatch between an inni operation and the corresponding operation declaration. 40

List of Tables

5.1 Average elapsed exection time (in seconds) and throughput data collected from running VolcanoMark 2.5. 30
5.2 Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library. *Rmic* cache was not used during translation. 35
5.3 Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library. 35
5.4 Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library. . . . 36
5.5 Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library. 36

6.1 Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks. 44
6.2 Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks. 45

A.1 Brief description of each Java benchmark used in comparing the compilation time performance. 49
A.2 Brief description of each Java benchmark used in comparing the execution time performance. 50

B.1 Average elapsed exection time (in seconds) and throughput data collected from running VolcanoMark 2.5. 51
B.2 Average elapsed compilation time (in seconds) (on PC with 2.8G dualprocessor) on selected Java benchmarks. 52
B.3 Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected Java benchmarks. 52
B.4 Average elapsed execution time (in seconds) (on PC with 2.8G dualprocessor) on selected Java benchmarks. 53
B.5 Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected Java benchmarks. 53

C.1 Brief description of each JR benchmark that does not use the RMI library. 55
C.2 Brief description of each JR benchmark that uses the RMI library. . . 56

D.1 Command line arguments provided for each selected benchmark. . . . 58

E.1 Average elapsed compilation time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that do not use the RMI library. *Rmic* cache was not used during translation. 59
E.2 Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library. *Rmic* cache was not used during translation. 60
E.3 Average elapsed execution time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that do not use the RMI library. . 61
E.4 Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library. . . . 62
E.5 Average elapsed compilation time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that use the RMI library. 63
E.6 Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library. 64
E.7 Average elapsed execution time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that use the RMI library. 65
E.8 Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library. 65

F.1 Average elapsed compilation time (in seconds) (on PC with 2.8G dual-processor) on selected JR benchmarks. 67
F.2 Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks. 68
F.3 Average elapsed execution time (in seconds) (on PC with 2.8G dual-processor) on selected JR benchmarks. 69
F.4 Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks. 70

Chapter 1

Introduction

The JR Concurrent Programming Language [16] extends Java to provide a rich concurrency model, based on that of the SR Concurrent Programming Language [1]. JR performs synchronization using an object-oriented approach, and provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, asynchronous communication, rendezvous, and dynamic process creation. During program compilation, JR source programs are translated into standard Java programs, which are then compiled using the standard Java compiler. Both the JR language translator and the JR runtime support system are written in standard Java. The JR implementation originated from the implementation of Java 1.2 and can be executed under Java 1.4.

The current release of Java 5.0[1] [18] brings a new era to both the Java programming language and also to the Java compiler. Version 5.0 not only introduces many new language features such as a generics feature (which functions similarly to templates in C++), base libraries, improvements in performance, etc., but also comes with a compiler that is redesigned and restructured, which makes understanding, enhancing, debugging, and maintaining it much easier than its previous releases.

[1]Both version numbers “1.5.0” and “5.0” can be used to identify this release. Version “5.0” is the product version, while “1.5.0” is the developer version.

Although Java 5.0 provides backward compatibility for programs written in earlier versions of Java, a JR source code upgrade is desirable because not only will JR users be able to utilize the new language enhancements and benefit from performance gains, but JR developers will also be able to maintain the translator source more easily because of the new Java compiler design.

This thesis describes our experience with upgrading the current release of JR, JR 1.00061, to a new version, JR 2.00001, that works with Java 5.0. The performance of this new version is also discussed. The upgrade process mainly involves a source code merge between the current JR translator and the Java 5.0 compiler, with some additional support for new language features to be used within JR extensions (such as operations with variable number of arguments and generic operations). The process also includes rewriting the JR runtime support system so that it implements the new language features. We compare the performance of JR 2.00001 against the current release using compilation time and execution time of some JR benchmarks.

This thesis also describes our attempt to modify the implementation of JR 2.00001 so that new Java language features can be used in the generated code to reduce the amount of code produced for each JR program. We number this optimized version JR 2.00005. However, this optimized version is not yet fully functional. We ran the same JR benchmarks using JR 2.00005 and use the resulting data to measure possible improvements of this version over both the current JR release and JR 2.00001.

The rest of this thesis is organized as follows. Chapter 2 gives background on the new Java features and also a high-level overview of the Java 5.0 compiler structure. Chapter 3 presents new JR features and illustrates them with some examples. Chapter 4 discusses the merging scheme, the approach to modification, bug fixes, and known bugs. Chapter 5 shows the performance results of JR 2.00001. Chapter 6 talks about our attempt at optimizing JR 2.00001. Chapter 7 concludes this thesis and presents possible future work.

This thesis assumes readers already know Java and JR. For Java language background, please refer to the Java documentation and tutorial at [11]. For further information on the JR language, please refer to the text on the JR Programming Language [16]. To distinguish plain text explanations with keywords from source code, we typeset example programs, JR distribution code fragments, and specific keywords in the **Typewriter** typeface.

Chapter 2

Background

2.1 New Java Language Features

Java 5.0 contains several new language features including generics, enhanced for loop, autoboxing / unboxing, typesafe enums, varargs, static import, and annotations[1]. This section briefly goes over each of these features and Chapters 3 and 4 talk about how JR utilizes and builds on some of these features.

2.1.1 Generics

```
static void expurgate (Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (((String) i.next()).length() == 4)
            i.remove();
}
```

Figure 2.1: Remove 4-letter words from Collection **c** using Java 1.4. Elements of **c** must be **Strings**.

Generics allows a type or method to operate on objects of various types while providing compile-time safety. It adds compile-time safety to the **Collections Frame-**

[1]We follow the Java 5.0 Documentation [11] in referring to the new Java features as singular or plural.

```
static void expurgate (Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
```

Figure 2.2: Remove 4-letter words from Collection **c** using Java 5.0.

work and eliminates the drudgery of casting [11]. Figure 2.1 (from [11]) shows an example of accessing elements in a **Collection** for Java version 1.4[2]. Compare that with the same example in Figure 2.2 (from [11]) written for Java 5.0. Generics allows users to specify the type of a collection to the compiler, so that type checking can be performed at compile time. If the collection is used consistently, the compiler can also insert the correct casts on values being taken out of the collection. Hence, users are certain that their programs will not throw any run time **ClassCastExceptions**. Programs that use generics are highly readable and robust.

2.1.2 Enhanced For Loop (For-Each Loop)

```
void cancelAll(Collection<TimerTask> c) {
    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
        i.next().cancel();
}
```

Figure 2.3: Iterating over a collection using an iterator.

```
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c)
        t.cancel();
}
```

Figure 2.4: For-each loop in Java 5.0.

[2]Most language features in Java 1.4 also exist in earlier Java versions.

Iterating over a collection involves a for loop and an iterator. Figure 2.3 (from [11]) shows a typical example of accessing elements in a collection. With the new for-each loop in Java 5.0, iterators are no longer required because those operations are already combined into a simple colon (:). Figure 2.4 (from [11]) presents the same example written using a for-each loop. Eliminating the iterator declaration, loop initialization, and loop condition reduces the chance for error. At the same time, programs are easier to read and understand. In addition to collections, this feature is also applicable to arrays so that the index variable is not needed.

2.1.3 Autoboxing / Unboxing

```
ArrayList al = new ArrayList();
al.add(new Integer(1));
int element = ((Integer)al.get(0)).intValue();
```

Figure 2.5: Wrapping a primitive type using a wrapper class and then extracting the primitive type from the wrapper using Java 1.4.

```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(1);
int element = al.get(0);
```

Figure 2.6: Autoboxing and Unboxing in Java 5.0.

In previous Java releases, a primitive type must first be "boxed" into a wrapper class before putting it into a collection. Similarly, an appropriate **typeValue** method must be used to "unbox" the object when taking an object out of a collection and extracting the value to a primitive type. Figure 2.5 shows an example of wrapping up an **int** into an **Integer** and then extracting the value from an **ArrayList**. The autoboxing and unboxing feature automates and eliminates these chores. Figure 2.6 shows the same example but uses autoboxing and unboxing: **int** is auto-wrapped into an **Integer** and invocation of **intValue** is no longer required. This feature saves programmers time and effort while at the same time reduces the opportunity for error.

2.1.4 Typesafe Enums

Before the enumerated types feature was added in Java 5.0, expressing an Enum pattern would be similar to the following (from [11]):

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL = 3;
```

This pattern is problematic. For example, it is not typesafe because it allows passing in any other **int** value where a season is required, or adding two seasons together, which is meaningless. This pattern does not use any namespace and therefore a prefix (such as **SEASON_** above) must be used to avoid name conflicts. If any new constant is added between two existing constants or the order is changed, all clients that depend on these numbers need to be recompiled. Finally, since these numbers are just plain **int**s, they do not contain any type information.

In Java 5.0, the above pattern can be written using enumerated types:

```
enum Season { WINTER, SPRING, SUMMER, FALL };
```

The enumerated types feature in the latest Java release not only solves all the problems mentioned in the previous example, but also allows adding arbitrary methods and fields to an **enum** type, implementing arbitrary interfaces and more. Enum types provide high-quality implementations of all the **Object** methods [11]. They are also **Comparable** and **Serializable**.

2.1.5 Varargs

Varargs eliminates the need for manually “boxing” up argument lists into an array when invoking methods that accept variable-length argument lists [11]. Figure 2.7

```
Object [] arguments = {
    new Integer(7),
    new Date(),
    "a disturbance in the Force"
};

String result = MessageFormat.format(
    "At {1, time} on {1, date}, there was {2} on planet " +
    "{0, number, integer}.", arguments);
```

Figure 2.7: Creating an array of arguments prior to invoking a method that takes variable number of arguments using Java 1.4.

```
public static String format(String pattern, Object ... arguments);
String result = MessageFormat.format(
    "At {1, time} on {1, date}, there was {2} on planet " +
    "{0, number, integer}.", 7, new Date(), "a disturbance in the Force");
```

Figure 2.8: Declaring and invoking a method that takes variable number of arguments using Java 5.0.

(from [11]) shows an example of creating an array of arguments and a method call that uses it. Although multiple arguments must still be passed as an array for Java 5.0, the varargs feature automates and hides the process. A method that takes multiple arguments now can declare the last parameter's type with an ellipsis (...), which indicates the last argument can be passed as an array or as a sequence of arguments. Figure 2.8 (from [11]) shows the same example coded for Java 5.0.

2.1.6 Static Import

Accessing static members using Java 1.4 or earlier requires qualifying references with the class in which they were declared. For example, it is necessary to specify **Math** in the following case (from [11]):

```
double r = Math.cos(Math.PI * theta);
```

Programmers sometimes put static members into an interface and inherit from that interface [11]. Doing so would make the class become part of the interface class's public API and hence implementation details leak into public APIs. The static import feature allows unqualified access to static members without inheriting from the type containing the static members. In order to rewrite the above example without qualifying **cos** and **PI**, we import an individual member by, for example:

```
import static java.lang.Math.PI;
```

Or, we import all members by:

```
import static java.lang.Math.*;
```

Then we can rewrite the assignment to **r** as:

```
double r = cos(PI * theta);
```

2.1.7 Metadata

Annotation mechanisms in the Java platform work like tags, which indicate information about their target fields or methods. For example, the **transient** modifier is an ad hoc annotation indicating that a field should be ignored by the serialization subsystem, and the **@deprecated** javadoc tag is an ad hoc annotation indicating that the method should no longer be used [11]. Java 5.0 has a general purpose annotation facility permitting the utilization of user defined annotation types. The facility consists of syntax for declaring annotation types, syntax for annotation declarations, APIs for reading annotations, a class file representation for annotations, and an annotation processing tool [11]. Annotations affect the way programs are treated by tools and libraries, which can indirectly impact program semantics. They can be read from source files, class files, or reflectively at run time.

2.2 New Java Implementation Feature

J2SE 5.0 improves performance in various areas [8]. Improvements to new language features include: additions to the virtual machine, enhancements to the base and integration libraries, the user interface, deployment, tools and architectures, and OS and hardware platforms optimizations [8]. Enhancements to program execution speed include: garbage collection ergonomics, the StringBuilder class, Java 2D technology improvements, and performance and memory usage improvements to image I/O [8]. One of the enhancements in integration libraries that has a direct impact on the performance of JR is the Remote Method Invocation (RMI).

2.2.1 Remote Method Invocation (RMI)

RMI in Java 5.0 supports dynamic generation of stub classes. Programmers no longer need to pre-generate stub classes for remote objects using the Java Remote Method Invocation (Java RMI) stub compiler, *rmic*. (5.0 VMs do not use pre-generated stub classes.) Pre-generation of stub classes is only required when the remote object needs to support clients running in pre-5.0 VMs. Section 3.2.1 gives a detailed discussion about the impact of the changes to RMI on JR.

2.3 Java Compiler Design

The Java compiler source code bundle is saved under the **javac** directory and contains a **Main** file for invocation purpose. The rest of the **javac** directory is subdivided into eight directories:

- The **code** directory contains files that add attributes to each subtree in the parse tree. For example, it contains **Type**, which denotes the type of each node; and **Flag**, which stores the access flags of each subtree.

- The **comp** directory contains files that make up the core of the compiler. For example, it contains **Check**, which defines utility methods to perform semantic checks; **Flow**, which performs flow analysis on the source; and **Attr**, which decorates each subtree with an environment and a symbol that stores all the necessary information.

- The **main** directory contains files that process command line arguments and start up the compiler.

- The **parser** directory contains files for the parser such as **Scanner** and **Keywords**.

- The **resource** directory contains files for output messages such as **compiler.properties**, which stores all error and warning messages.

- The **tree** directory contains files that store the parse tree and other utility classes that manipulate the tree.

- The **util** directory contains utility classes such as **List** and **Log** that are used by all other classes in the compiler.

- Finally, the **jvm** directory contains files that generate Java byte codes.

Each time the compiler is invoked, it first parses all source files and creates a parse tree for each file. Then, it traverses all the trees to record information of all class members into their corresponding environment. Next, it traverses those trees again to record local variables and resolve all function calls and variable references. After that, the compiler performs a flow analysis on those parse trees and adds type casts where necessary. Finally, it outputs Java byte code for each parse tree.

The following two design characteristics of the Java 5.0 compiler play critical roles in explaining the improvement of the JR 2.00001 performance. First, within each hierarchy of classes inside the compiler (such as the classes for parse tree nodes, variable types, and variable symbols), a tag of type **long** is assigned to each class to identify its type at run time, so that performance can be improved in run time type checks. For example, in the previous version of the Java compiler implementation, it checks the type of a parse tree using the following boolean condition:

```
tree instanceof ForStatement
```

The implementation of the 5.0 compiler checks the type of a parse tree using the following boolean condition:

```
tree.tag == Tree.FOR
```

where **tag** and **FOR** are of type **long**. Second, class members are declared public to eliminate the use of accessor methods, which improves efficiency. Section 5.2.1 further discusses the impact of both.

Chapter 3

New JR Features

3.1 New JR Language Features

Basing the new JR on Java 5.0 leads to two kinds of changes in the JR language. First, all the new Java features become new JR features. Second, a few of the JR language extensions incorporate the new Java features. Those JR language extensions mainly appear in quantifiers and operations. In addition, we also changed the syntax of **process** and changed the definition of the JR vm type so that it is now an **Object** in the new release.

3.1.1 Quantifiers

Quantifiers in **process** definitions, **inni** statements, and **co** statements originally borrowed and extended their syntax from Java's **for** statement. A quantifier has the syntax:

```
Quantifier :
    (ForInit ; [ Expression ] ; ForUpdate [ ; [ Expression ] ] )
```

The fourth, optional boolean expression serves as a such-that expression; an operation is performed for a particular quantifier value only if the expression evaluates to true. Examples of a quantifier with a such-that expression are given later in this section.

With the new for-each loop in Java 5.0, JR quantifiers now have the new syntax:

```
Quantifier :
    ( ForInit ; [ Expression ] ; ForUpdate [ ; [ Expression ] ] )
    | ( Type Ident : Collection [ ; [ Expression ] ] )
```

The new syntax inherits the semantics from Java's for-each loop, which means the identifier is assigned a value equal to each element in the collection one at a time. Specifically, in a quantified **process**, each quantifier obtains a unique element from the collection. Figure 3.1 presents an example of the new form of **process** quantifier and the expected output.

```
import edu.ucdavis.jr.JR;

public class ProcessQuant1 {
    public static int [] x = new int [5];

    static {
        for (int i = 0; i < x.length; i++)  x[i] = i;
    }

    static process p ((int i : x)) {
        System.out.println("i is: " + i);
    }

    public static void main(String [] args) {}
}

/*
Expected Output (its order is non-deterministic):
i is: 0
i is: 1
i is: 2
i is: 3
i is: 4
*/
```

Figure 3.1: Example of new process quantifiers.

The new quantifier syntax preserves the optional such-that expression. Figure 3.2 gives an example of using a **process** quantifier with a such-that expression. Notice from the output that the process with quantifier equal to 2 is missing because the such-that expression fails for that case. Similar to the original form, this new **process** quantifier syntax requires the first expression to specify a new variable.

```
import edu.ucdavis.jr.JR;

public class ProcessQuant2 {
    public static int [] x = new int [5];

    static {
        for (int i = 0; i < x.length; i++)
            x[i] = i;
    }

    static process p ((int i : x; i != 2)) {
        System.out.println("i is: " + i);
    }

    public static void main(String [] args) {}
}

/*
Expected Output (its order is non-deterministic):
i is: 0
i is: 1
i is: 3
i is: 4
*/
```

Figure 3.2: Example of the new process quantifiers with a such-that expression.

The semantics of a for-each quantifier in **inni** statements and **co** statements is similar. Figure 3.3 gives a sample program that consists of two **inni** statements, where one has a **st** expression and the other specifies the same condition inside the quantifier. The two statements are equivalent and yield the same output. Figure 3.4 shows a quantified **co** statement and the correct output. Note that the program does not execute the **send** invocation when the such-that expression evaluates to false, and also skips the post processing code.

```
import edu.ucdavis.jr.JR;

public class InniQuant {
    public static int [] x = new int [5];

    static {
        for (int i = 0; i < x.length; i++) x[i] = 10 - i;
    }

    public static op void test();

    public static void main(String [] args) {
        // k is only used to drive this loop, unused in the body
        for (int k : x) {
            send test();
            send test();
        }
        for (int k : x) {
            inni ((int j : x)) void test() st j == k {
                System.out.println("received j: " + j);
            }
        }
        System.out.println("-----------------------------");
        for (int k : x) {
            inni ((int j : x; j == k)) void test() {
                System.out.println("received j: " + j);
            }
        }
    }
}

/*
Expected Output:
received i: 10
received i: 9
received i: 8
received i: 7
received i: 6
-----------------------------
received i: 10
received i: 9
received i: 8
received i: 7
received i: 6
*/
```

Figure 3.3: Example of the new inni statement quantifiers with such-that expressions.

```
import edu.ucdavis.jr.JR;

public class CoQuant {
    public static int [] x = new int [5];

    static {
        for (int i = 0; i < x.length; i++) x[i] = i;
    }

    public static op void test() {
        System.out.println("test invoked");
    }

    public static void main(String [] args) {
        co ((int i : x; i != 2)) send test() {
            System.out.println("test: " + i);
        }
    }
}

/*
Expected Output (its order is non-deterministic):
test invoked
test invoked
test invoked
test invoked
test: 0
test: 1
test: 3
test: 4
*/
```

Figure 3.4: Example of the new co statement quantifiers with a such-that expression.

3.1.2 Operations

Varargs

One of the two new features added to operations is the varargs feature. As introduced in Section 2.1.5, methods in Java 5.0 that take in a variable number of arguments can declare the last parameter with an ellipsis. The same principle works for operations in the new version of JR. Operations with a variable number of arguments can declare the last parameter as varargs and accept actuals passed in as an array or as a sequence of arguments. The program in Figure 3.5 includes an operation with varargs.

```
import edu.ucdavis.jr.JR;

public class VarargsOp {
    public static op void test(int... x) {
        for (int i = 0; i < x.length; i++)
            System.out.print(x[i] + " ");
        System.out.println("");
    }

    public static void main(String [] args) {
        test(0);
        test(1, 2, 3);
        int [] a = {1, 2, 3, 4};
        test(a);
    }
}

/*
Expected Output:
0
1 2 3
1 2 3 4
*/
```

Figure 3.5: Example of an operation with varargs.

Generic Operations

Another new feature is generic operations. Generic operations in JR and generic methods in Java work in a similar way. Figures 3.6 and 3.7 show a JR program with a generic operation **get()**. Class **MyShape** contains a single member variable **id** of type **E**, where **E** is a parameterized type provided to the class **MyShape**. Operation **get()** will return **id** of whatever type passed into the class.

3.1.3 Process

The syntax of a JR process is now:

```
process Identifier [ ( Quantifiers ) ] Block
```

For example, an unquantified process may look like:

```
process p { ... }
```

```
import java.util.ArrayList;
import java.util.Iterator;

public class GenericsOp {
    public static void main(String [] args) {
        ArrayList<MyShape> shapes = new ArrayList<MyShape>();

        shapes.add(new MyShape<String>("square"));
        shapes.add(new MyShape<String>("triangle"));
        shapes.add(new MyShape<String>("rectangle"));

        for (Iterator<MyShape> it = shapes.iterator(); it.hasNext(); ) {
            System.out.println("MyShape " + it.next().get());
        }
    }
}

/* Expected Output:
MyShape square
MyShape triangle
MyShape rectangle
*/
```

Figure 3.6: Example of an operation with generics.

```
public class MyShape <E> {
    protected E id;

    public MyShape (E id) {
        this.id = id;
    }

    public op E get() {
        return id;
    }
}
```

Figure 3.7: Example of an operation with generics.

However, JR 1.00061 allows declarations with a pair of empty parentheses after the JR process's name, such as:

```
process p () { ... }
```

We decided to keep the syntax simple and disallow the above declaration, so that a JR process without quantifiers cannot contain a pair of empty parentheses.

3.1.4 Virtual Machines

In the new JR, a JR virtual machine behaves similarly to a subtype of **Object** so that programmers can declare **vm** as a parameterized type of an **Object** collection. However, a JR virtual machine is not exactly a subtype of **Object** because we do not provide any methods that are available to all the **Objects** (such as **equals()** and **getClass()**).

3.2 New JR Implementation Feature

3.2.1 Remote Method Invocation (RMI)

In JR version 1.00061 and earlier, before a JR program is ready for execution, users need to invoke the JR translator to translate their JR program into a Java program, then use a standard Java compiler to compile the generated program, and then generate stub classes using the *rmic*. (JR provides tools to automate this process, allowing users to compile and execute JR programs using a single command.) A user can speed up the *rmic* translation process by enabling the JR *rmic* cache feature, where stub classes are copied from a user-specified *rmic* cache folder instead of generated by the RMI compiler. Since Java 5.0 performs RMI compilation at run time, JR 2.00001 users can safely skip the RMI compilation step and execute their program after compiling the generated Java program. As mentioned in Section 2.2.1, pre-generation of stub classes is only required when the remote object needs to support clients running in pre-5.0 VMs. Notice that pre-generating stub classes does not help performance when all clients are running in 5.0 VMs, which is what the JR implementation assumes.

Chapter 4

Implementation

This chapter discusses the merging scheme we chose and issues regarding upgrading the JR implementation at the source level.

4.1 Merging Scheme

Language implementations with Java support either follow a static scheme or a dynamic scheme. A static scheme refers to a system that consists of a runtime system and a compiler, where the compiler's source originates from a publicly distributed Java compiler and is modified for language extensions. A dynamic scheme refers to a system that combines with Java using a linker, and a stub generator to generate class files that are compatible with normal Java compilers.

JCilk [10] is a Java-based multithreaded programming language that extends the semantics of Java by introducing "Cilk-like" [6] [13] linguistic constructs for parallel control. JCilk follows a static scheme where the runtime system is written in Java and the compiler is implemented using the Polyglot compiler toolkit [3] and the Gnu Compiler for Java (GCJ) [19]. The system is easily portable because JCilk only adds three new keywords to Java and so the changes to the GCJ compiler are minimal. During compilation, a JCilk program is translated into an intermediate GoJava [10] program, which in turn invokes the JCilk runtime system. The modified GCJ compiler then compiles the generated GoJava program and outputs Java bytecode.

Jiazzi [12] is a system that adds support for large-scale component programming in Java and employs a dynamic scheme. Jiazzi combines with Java using a linker, which manipulates components, and a stub generator, which allows Jiazzi to be used with normal Java source compilers. Jiazzi does not extend the Java language. Instead, Java code for a component is written using the normal Java language and a separate linking language is used to manipulate components.

If we use a static merging scheme, JR becomes immutable, which means any major change in the Java language syntax implies a major code change in both the JR translator and the runtime system. However, since the JR language extends Java and the current distribution uses a static scheme, it would be easier to upgrade if we continue to use the same scheme. Hence, we follow the static scheme and merge by porting JR extensions into the Java compiler's source, so that the merged version will still be compatible with future Java releases if changes in the language are minor.

4.2 Schedule of Tasks

The entire upgrade process spanned approximately eight months. The features were ported in the following order:

1. **call** statements serviced by general functions
2. **op** method
3. **call** statements serviced by **op** method
4. **return** statements inside **op** method
5. **send** statements with exception handling, serviced by **op** method

6. **JR.exit(int)**
7. **op** declaration, invoked by **call** or **send**
8. local **op** declaration, invoked by **call** or **send**
9. **process**
10. semaphore declarations (member, local), **P**, **V**
11. **receive**
12. **reply**, **forward**, **return**
13. **capabilities**, **null**, **noop**
14. quiescence
15. new instance **op** declaration, new instance semaphores
16. **inni** statement
17. **co** statement
18. virtual machines, remote objects

In parallel to porting JR features to Java 5.0, the runtime system was also modified so that it implements the new language features.

4.3 Verification

The JR distribution includes a verification suite (vsuite) for testing the correctness of the translator and the runtime system. A "codeextract" that contains all the example programs from the JR textbook [16] also serves as a test suite to verify the correctness of JR. Each test case consists of the source, a test script, and a file with the correct program output if the program does not have any syntax errors, or the correct error messages if it does. The JR distribution contains a *jrv* tool that launches each test case in the test suite according to the commands in the script file, compares the output against the expected output, and reports the result.

Each time a feature was successfully ported, a small test suite was written specifically for that feature to make sure everything worked and nothing was broken. The collection made up by these small test suites is now included in the regular JR test suite (vsuite/adeveloper) and future JR developers can use it as a preliminary test for their implementation. In addition, test scripts in the vsuite and in the codeextract were changed accordingly so that the expected error messages match those of the new release.

4.4 Bug Fixes

We found several bugs in the Java 5.0 distribution and also in the JR 1.00061 distribution. They have been fixed in JR 2.00001. Below is a discussion of each bug:

4.4.1 Bugs in Java 5.0

Anonymous Class Constructor

If a class is declared with no constructor, the Java compiler would automatically insert a default constructor that takes no argument for that class. If a class is anonymous and has no constructor, the Java compiler still outputs a constructor with no name. We submitted a bug report to the Sun Developer Network [18], but this bug still exists in the latest Java version (1.5.0_05).

Object And Boolean Constant Comparison

When comparing an **Object** with a boolean constant, if the boolean constant is on the left of the equality operator (==, !=), and the **Object** is on the right, *javac* fails to recognize this semantic error. We also submitted a bug report to the Sun Developer Network [18], but this bug still exists in the latest Java version (1.5.0_05).

For-each Loop

Javac erases all the parameterized type information during compilation and converts them into **Objects** after type checking. This becomes a problem when we have parameterized type specific operations. The new for-each loop is one of the affected operations. When we ask *javac* to output the source files instead of the class files after compilation, the generated code introduces a type error at re-compilation. For example, consider the following code fragment:

```
ArrayList<Integer> myArray = new ArrayList<Integer>();
for (Integer elem : myArray)
    System.out.println(elem.toString());
```

The generated code becomes:

```
ArrayList myArray = new ArrayList();
for (Integer elem : myArray)
    System.out.println(elem.toString());
```

which introduces a type error at the for-each loop between **elem** and **myArray** when we compile this code, because **elem** is an **Integer** but elements of **myArray** have type **Object**. JR 2.00001 inherits this problem from Java 5.0. We fixed this problem by changing the type of the loop variable to **Object** and casting all occurrences of the loop variable to its original type inside the body of the for-each loop. After we applied the fix, the generated code of our example code fragment becomes:

```
ArrayList myArray = new ArrayList();
for (Object elem : myArray)
    System.out.println(((Integer)elem).toString());
```

4.4.2 Bugs in JR 1.00061

Capability Comparisons

When a capability is compared with another capability or **noop** using an equality operator (==, !=), for example:

```
myCap1 == myCap2
myCap3 == noop
```

the generated Java code becomes:

```
myCap1.getOp() == myCap2.getOp()
myCap3.isNoop()
```

which causes **NullPointerException** if the dereferencing capability is null.

Member Capability Initialization

When initializing a member capability with a **noop**, the JR translator outputs **noop** instead of **Cap_AToB.noop**, where **A** stands for the destination capability's return type and **B** stands for the destination capability's argument types.

Also when initializing a member capability with a member operation, a cast to wrap the operation into a capability before the assignment is missing in the generated code.

Remote Object Accessing An Instance Capability

A remote object can only access static members of a class. If a remote object attempts to access an instance capability (non-array type) of a class, the error will not be caught until the generated Java program is being compiled.

4.5 Known Bug

This section gives a brief discussion and the possible fix for a known bug in JR 2.00001.

4.5.1 Known Bug in JR 2.00001

Generic Capability

A capability with generic return and argument types in a JR source program is translated into a capability that returns an **Object** and takes in **Object** for all arguments of parameterized types. In the current implementation, we do not organize different kinds of capabilities into a hierarchy. Although type **A** and type **B** are both subtypes of **Object**, a capability of **A(B)** is not a subtype of a capability of **Object(Object)**. Therefore, all capabilities that use this relationship fail in our current implementation.

Figure 4.1 is an example program that illustrates a problem caused by a generic capability. In the generated code, the type of **mycap** is translated to a capability that takes a single argument of type **Cap_ObjectToObject** and returns **Cap_ObjectToObject**. In the **main** function, **gcap** is an instance of type **GenericCap<String>**. Its member capability **mycap** is supposed to take a single argument of type **Cap_StringToString** and returns **Cap_StringToString**. However, **Cap_StringToString** is not a subtype of **Cap_ObjectToObject**. Therefore, the Java compiler complains that the invocation of **gcap.mycap** is passing an argument of a wrong type and also complains that the assignment of the return value of **gcap.mycap** to **test** is incompatible.

We are still looking into ways to solve this problem.

```
import edu.ucdavis.jr.JR;

public class GenericCap<E> {

    public cap cap E(E) (cap E(E)) mycap = new op cap E(E) (cap E(E));

    process p {
        while (true) {
            inni cap E(E) mycap (cap E(E) x) {
                x = noop;
                reply x;
            }
        }
    }

    public static void main (String [] args) {
        GenericCap<String> gcap = new GenericCap<String>();
        cap String(String) test = null;
        test = gcap.mycap(test);
    }
}
```

Figure 4.1: Example of a capability with generics.

Chapter 5

Performance

This chapter presents our performance evaluation of JR 2.00001. We performed all tests on two PCs (1.4 gigahertz uniprocessor with 512 megabytes RAM and 2.8 gigahertz dual-processor with 1 gigabytes RAM) running Linux. Each benchmark was run five times to obtain the average. We measured performance using the Linux *time* command, when the systems were lightly loaded. The results use the elapsed time, which is close to the system CPU time. Although we calculate the average elapsed time based on data from five trial runs, we ran all benchmarks many times to observe that the results of all trials are comparable with each other even though the system load varies. Appendices B and E contain the detailed statistics including variances. The variance relative to each average is insignificant (with one exception, see Section 5.1.2).

5.1 Java Performance

Appendix B includes the results of running certain Java programs on both Java 1.4 and 5.0 so that we can better analyze the performance of JR. Appendix A gives a brief description of each benchmark. We picked large programs for compilation tests and non user-input-dependent programs for execution tests.

5.1.1 Compilation Time

Sun focuses Java's performance on execution time; therefore, they did not release any official performance data on compilation. The average elapsed compilation times for large Java programs we collected show that Java 5.0 takes 30-40% longer than Java 1.4 to compile each program on both platforms. Although Section 2.3 mentions the new design characteristics in the 5.0 compiler that can improve the compile time performance, the improvements are less than the other overheads. Tables B.2 and B.3 show the average elapsed compilation time for large Java programs.

5.1.2 Execution Time

2.8G dual-processor		
	Java 1.4	**Java 5.0**
Average Elapsed Time	15.78	15.81
Average Throughput	25379.40	25339.60
1.4G uniprocessor		
	Java 1.4	**Java 5.0**
Average Elapsed Time	42.72	40.77
Average Throughput	9363.40	9813.00

Table 5.1: Average elapsed exection time (in seconds) and throughput data collected from running VolcanoMark 2.5.

As a way to compare our performance results against those released by Sun, we ran one of the benchmarks[1] they used in the J2SE 5.0 Performance White Paper [8] on our machines. VolcanoMark 2.5 [20] involves over 200 CPU bound processes competing with each other for the CPU. The variance values for the average throughput[2] of VolcanoMark 2.5 are reasonable[3]; in fact, each throughput value is within 8% of

[1] The paper uses two benchmarks: VolcanoMark 2.5 and SPECjbb2000. Only the first one is available for free.

[2] We report the throughput calculated by VolcanoMark 2.5.

[3] A variance can be larger than the mean, as the data in Table B.1 illustrate.

the mean. Although Sun reports that Java 5.0 gives a 20% higher throughput than Java 1.4 when the benchmark was tested on a system with four 2.4 gigahertz AMD Opteron CPU's and 8 gigabytes RAM running Solaris 10, Java 5.0 gives a lower throughput when tested on our 2.8G dual-processor machine running Linux and gives a 4% increase on our 1.4 G uniprocessor machine. Table 5.1 shows our result running VolcanoMark 2.5. We also ran some Java programs that exhibit polynomial time complexity to collect the average elapsed execution time. Programs that do not use **Vector** execute slightly faster using Java 1.4 on both machines, while programs that use **Vector** execute faster using Java 5.0 on both machines. Tables B.4 and B.5 show the average elapsed execution time of those Java programs. These results indicate the performance of Java 5.0 compared with Java 1.4 is both program and platform dependent.

From these numbers we also observe that the dual-processor machine performs almost two times as fast as the uniprocessor machine.

5.2 JR Performance

We evaluated the performance of JR 2.00001 by comparing the average compilation time collected to those collected from JR 1.00061.

The code generated by JR 1.00061 and the code generated by JR 2.00001 is very similar. So a comparison of the average execution times is merely a comparison between the Java 1.4 virtual machine and the Java 5.0 virtual machine. However, we have provided the execution times of our benchmarks to give the reader an idea of the runtime performance difference between using JR 1.00061 with the Java 1.4 VM and using JR 2.00001 with the Java 5.0 VM. Notice that although we could compile and run the generated code of both JR 1.00061 and JR 2.00001 on the same version of Java VM, we chose to run our tests on a Java 1.4 VM for the code generated by

JR 1.00061 and on a Java 5.0 VM for the code generated by JR 2.00001 to simulate the environments that most JR users would use.

The benchmarks used were taken from the JR test suite (vsuite) that are specially designed for timing and virtual machine creation, as well as application programs we and others wrote that represent long and complex computations. Appendix C gives a brief description of each JR benchmark. Appendix D shows the arguments used for each selected benchmark during execution. Appendix E presents the average elapsed compilation time[4] and execution time of five trial runs, as well as the corresponding variance.

It is easy to notice through a quick glance at the data that JR performs twice as fast on the dual-processor machine than on the uniprocessor machine. This agrees with the trend we saw from running Java benchmarks. Since the results on the two platforms show the same general trend, we show only the data collected from the uniprocessor machine in the tables in this chapter.

5.2.1 Compilation Time

The average elapsed compilation times for each JR version on programs that do not involve the RMI library show that JR 2.00001 is faster than JR 1.00061 for normal program compilation. The positive speedup is obvious from the data collected on the uniprocessor. On the other hand, the speedup seems trivial on the dual-processor machine for small programs, but it is significant in more complex programs such as simulationAllPPC and simulationAllPPC2. This is because of the clock speed difference between the two machines. Table 5.2 presents the average elapsed compilation time for each JR version on the uniprocessor of programs that do not involve the RMI library.

[4]Compilation time refers to the time for translating a JR program to a Java program, compiling the generated Java code, and, in JR 1.00061, performing the RMI compilation.

For programs that use the RMI library, JR 1.00061 with *rmic* cache disabled[5] requires the longest compilation time, while JR 1.00061 with *rmic* cache enabled performs slightly faster than the new JR on short programs but performs slower on programs with long source codes. Table 5.3 shows the average elapsed compilation time on the uniprocessor of programs that use the RMI library.

We know from Section 5.1 that Java 5.0 generally performs slower than Java 1.4 when compiling Java programs; the speedup for JR programs that do not use the RMI library mainly comes from the difference in the implementation of the extended Java. The JR 1.00061 translator uses a lot of runtime type checks and non-final accessor methods, while the new JR translator follows the design of the new Java compiler and uses **long** for type tags and replaces accessor methods with public members, as mentioned in Section 2.3, to improve efficiency. On the other hand, since Java 5.0 performs RMI compilation at run time, JR 2.00001 requires much less time than JR 1.00061 without *rmic* cache enabled to compile short programs that use the RMI. JR 1.00061 with *rmic* cache enabled takes slightly less time than the new JR to complete the compilation process because the *rmic* cache not only stores all the stub classes, but also some of the **op** classes[6]; therefore, time is saved on regeneration of files. However, when a program is long, the cost of performing regular translation becomes higher using JR 1.00061, as we have seen this behavior from the results of running benchmarks with no RMI. Therefore, benchmarks such as the dining philosopher visualization, the readers writers simulation, and the distributed file system require much longer time to compile under JR 1.00061 even when *rmic* cache is used.

[5] JR provides the *rmic* cache feature to cache some of the generated source files in order to speed up the compilation time.

[6] The *rmic* cache also stores **op** classes such as **InOp_typeTotype_impl.java**, **ProcOp_typeTotype_impl.java**, etc.

5.2.2 Execution Time

After we ran benchmarks that do not use the RMI library, we found that the program generated by JR 1.00061 and the program generated by JR 2.00001 from the same JR program take almost the same amount of time to execute if the JR program is short-running. The generated programs of simulationAllPPC and simulationAllPPC2, each containing a total of 4000 sends and receives, take much less time to run on the Java 5.0 VM than on the Java 1.4 VM. This speedup is due to the performance difference between the Java 1.4 VM and the Java 5.0 VM. It is easier to notice the difference when the startup overhead is relatively small compared to the overall execution time. Table 5.4 presents the average elapsed execution time on benchmarks that do not use the RMI library.

For benchmarks that use the RMI library, the generated programs of JR 2.00001 take a little longer to execute on the Java 5.0 VM than the generated programs of JR 1.00061 executed on a Java 1.4 VM. The new Java 5.0 VM feature that generates stub classes dynamically is the main cause for the overhead. Benchmark vm/many creates a total of 20 JR virtual machines. The difference in execution time for that case on the dual-processor machine is almost unnoticeable, while the difference on the uniprocessor machine is minor. Table 5.5 gives the average elapsed execution time on benchmarks that use the RMI library.

	Compilation Time	
Benchmarks	**JR 1.00061**	**JR 2.00001**
timings/asynch	7.10	4.55
timings/ircall	7.12	4.62
timings/irnew	7.19	4.69
timings/locall	7.05	4.56
timings/loop	7.00	4.50
timings/msgcsw	7.11	4.85
timings/pcreate	7.07	4.93
timings/rend	7.13	4.87
timings/semP	7.03	4.88
timings/semV	7.02	4.81
timings/semcsw	7.11	4.97
timings/sems	7.06	4.83
simulationAllPPC (w/ 1000 voters)	9.87	5.81
simulationAllPPC2 (w/ 1000 voters)	9.91	6.27

Table 5.2: Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library. *Rmic* cache was not used during translation.

	Compilation Time		
Benchmarks	**JR 1.00061**	**JR 1.00061 (w/ *rmic* cache)**	**JR 2.00001**
vm/basic	10.05	4.95	5.59
vm/basicchain	11.40	5.48	6.12
vm/callme	10.22	5.08	5.84
vm/many	6.99	3.99	4.93
vm/vmonvm	6.73	3.79	5.07
misc/dp/dpVis	45.79	20.32	14.25
codeextract2.0/dfs/all	24.16	11.17	8.77
RW visualization	44.86	26.15	16.59

Table 5.3: Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library.

	Execution Time	
Benchmarks	**JR 1.00061 on 1.4 VM**	**JR 2.00001 on 5.0 VM**
timings/asynch	5.52	5.57
timings/ircall	3.61	3.70
timings/irnew	5.35	5.49
timings/locall	3.97	3.67
timings/loop	3.00	3.07
timings/msgcsw	4.58	5.00
timings/pcreate	4.64	5.06
timings/rend	6.19	6.62
timings/semP	5.31	4.88
timings/semV	3.26	3.45
timings/semcsw	4.51	4.88
timings/sems	5.40	5.78
simulationAllPPC (w/ 1000 voters)	49.21	42.29
simulationAllPPC2 (w/ 1000 voters)	143.04	127.59

Table 5.4: Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library.

	Execution Time	
Benchmarks	**JR 1.00061 on 1.4 VM**	**JR 2.00001 on 5.0 VM**
vm/basic	4.65	4.97
vm/basicchain	5.90	6.28
vm/callme	9.02	9.77
vm/many	22.68	26.74
vm/vmonvm	5.00	6.20

Table 5.5: Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library.

Chapter 6

Implementation Optimization

Although the performance results in Chapter 5 show that JR 2.00001 has some improvements in compilation time compared with the current release, we would like to simplify the code generated during JR program translation hoping that we can further shorten the compilation time. Minimizing the number of files and simplifying each file generated for each JR program can shorten the total compilation time because time is not only saved in outputting source files, but also in compiling all the files.

Currently our optimized version of JR 2.00001, JR 2.00005, is still preliminary. It has bugs that we have not fixed and fixing them might reveal additional problems.

6.1 Generated Code Optimization

Both the current JR release and JR 2.00001 create a number of files for operations and capabilities. Specifically, JR generates a set of interfaces and implementations for each operation and capability with different formal parameter types and return type. For example, an operation that takes in a **char** and returns an **int** creates a set of classes with suffixes charToint and intTovoid, where inside each of these classes are functions that are specially written for argument type **char** and return type **int** or argument type **int** and return type **void**.

With generics and varargs in Java 5.0, we define in the JR runtime system parameterized operations and capabilities with suffix **Object** or **void**. The generated Java code in turn contains instances of those parameterized operations and capabilities, where each **op** or **cap** return type becomes the parameterized type. Functions inside each of these parameterized operation classes declare the last parameter as **Object ...**, so that they can take any argument types. Hence, the number of files generated by each JR program has decreased but the same functionality has been achieved.

Figure 6.1 is a simple example that illustrates the effect of this optimization. For

```
import edu.ucdavis.jr.JR;

public class SimpleOp {

    public static op int myOp1(char);

    public static op int myOp2(char, double);

    public static op int myOp3(double, char);

    public static void main(String [] args) {}

}
```

Figure 6.1: Source program illustrating the effect of generated code optimization.

this program, JR 2.00001 generates during translation the following files:

```
Cap_charToint.java                      ProcOp_charToint.java
Cap_charXdoubleToint.java               ProcOp_charToint_impl.java
Cap_doubleXcharToint.java               ProcOp_charXdoubleToint.java
Cap_intTovoid.java                      ProcOp_charXdoubleToint_impl.java
Cap_voidTovoid.java                     ProcOp_doubleXcharToint.java
InOp_charToint.java                     ProcOp_doubleXcharToint_impl.java
InOp_charToint_impl.java                ProcOp_intTovoid.java
InOp_charXdoubleToint.java              ProcOp_intTovoid_impl.java
InOp_charXdoubleToint_impl.java         ProcOp_voidTovoid.java
InOp_doubleXcharToint.java              ProcOp_voidTovoid_impl.java
InOp_doubleXcharToint_impl.java         Recv_char.java
InOp_intTovoid.java                     Recv_charToint.java
InOp_intTovoid_impl.java                Recv_charXdouble.java
InOp_voidTovoid.java                    Recv_charXdoubleToint.java
InOp_voidTovoid_impl.java               Recv_double.java
JRSimpleOp.java                         Recv_doubleXchar.java
JRjavadotlangdotObject.java             Recv_doubleXcharToint.java
```

```
Op_charToint.java                  Recv_int.java
Op_charXdoubleToint.java           Recv_intTovoid.java
Op_doubleXcharToint.java           Recv_void.java
Op_intTovoid.java                  Recv_voidTovoid.java
Op_voidTovoid.java                 SimpleOp.java
```

For the same program, JR 2.00005 generates:

```
JRSimpleOp.java
JRjavadotlangdotObject.java
SimpleOp.java
```

This example demonstrates that optimizing the implementation using generics and varargs decreases the number of files generated for each JR program dramatically. We will analyze the performance of this optimization in Section 6.4.

6.2 Implementation Problems

6.2.1 Inni Operation Invocations

In order to avoid confusion when having a single argument that is made up of a single array and a group of arguments made up of several variables, we wrap all the user-supplied actual parameters into an **Object** array. In addition to argument wrapping, we also put a cast in front of each actual argument so that the compiler knows exactly to which overloaded operation or method the invocation is referring. This mechanism leads to a problem for invocations serviced by an **inni** operation. The signature of an **inni** operation can be different than its corresponding operation declaration as long as the **inni** operation parameters are convertible to those of the operation declaration. While the casting of individual actual parameters follows the signature of the operation declaration, the casting of **inni** operation parameters in the generated code follows the signature of the **inni** operation. Hence, a runtime error occurs inside the **inni** statement when the two signatures do not match. Figure 6.2 gives an example that illustrates this.

```
import edu.ucdavis.jr.JR;

public class InniSig {
     public static op int a(String, int, int);
     public static void main(String [] args) {
         send a("a", 3, 4);
         inni int a(Object i, int j, float f) {
             System.out.println("inni a: " + i + " " + j + " " + f);
             return 1;
         }
     }
}
```

Figure 6.2: Source program illustrating the problem on signature mismatch between an inni operation and the corresponding operation declaration.

Parameters in the invocation of operation **a** in Figure 6.2 are internally wrapped into an **Object** array having elements of type **String, Integer, Integer**. When the invocation is serviced by the **inni** statement, **inni** treats the parameters as an **Object** array having elements of type **Object, Integer, Float**. A runtime error occurs for the last parameter when we cast an **Integer** to **Float**.

Although the problem can be fixed by making the **inni** operation's signature the same as the corresponding operation declaration's signature exactly, doing so loses the flexibility of having a one to many relationship between each inni operation and all the overloaded operation declarations.

6.2.2 View Statement

In a **view** statement, each **as** arm declares a unique list of parameters to be matched with the **Invocation** object. For each matching request, JR 2.00001 performs a runtime type check on the **Invocation** arguments and compares those with the types of the parameters in each **as** arm. With our optimization, all arguments are wrapped into subtypes of **Object**, which disables the runtime type check from locating the right **as** arm to execute. A fix to this problem is to add a signature **String** that represents the actual argument types to all JR runtime classes so that whenever we need to do a runtime type check, we can just perform a **String** comparison. The tradeoff for this solution is that each time an instance of a JR runtime class is created, it needs to pass in an extra **String** in the parameter list (and **String** comparison is slow).

6.3 Known Bugs

6.3.1 Array of Capabilities

Java 5.0 does not allow the delcaration of a parameterized type array because it leads to type unsafe operations [5]. Hence, each time a user declares an array of capabilities such as:

```
cap int(int) [] myCap = new cap int(int) [2];
```

JR translates the above code into:

```
Cap_Object<Integer> [] myCap = new Cap_Object [2];
```

Although it is not type safe, it keeps the generated code simple. This does not affect the static type checking of JR programs because all unsafe operations are caught by the JR translator. The Java compiler generates an unchecked warning for this unsafe operation, but it does not affect program compilation and execution. In fact, since this unsafe operation is internal to JR, we can suppress this warning to hide the details from JR users. There are two possible ways to hide the unchecked warnings. First, we can modify the scripts of all the JR commands so that a flag is added to the command for compiling the generated code to suppress all the unchecked warnings. For example, the command to compile the generated code using *javac* becomes:

```
javac -Xlint:-unchecked
```

Suppressing all the unchecked warnings does not eliminate all the warnings generated for unsafe operations made by a JR user because the JR translator is able to generate those warnings during program translation. Second, we can generate annotations for all the methods that contain unsafe operations. The annotation for suppressing unchecked warnings is the following:

```
@SuppressWarnings("unchecked")
```

However, this annotation feature is not yet implemented in the latest Java 5.0 release.

6.3.2 Generic Capability

This bug is inherited from the bug in JR 2.00001 mentioned in Section 4.5.1. Although we hide all the parameterized type arguments with an **Object** array, the return type of a capability still causes problems. A capability with return type **Object<A>** is not a subtype of **Object<Object>**. Therefore, problems in JR 2.00001 still exist in the optimized version.

6.4 JR 2.00005 Performance

Although the implementation of JR 2.00005 is not yet complete, we are able to run all benchmarks except the readers/writers simulation, which gives us some insight into the overall performance. The statistics show that JR 2.00005 compiles faster than JR 2.00001 for all our benchmarks. On large programs such as the Dining Philosophers Visualization and the Distributed File System, JR 2.00005 compiles 37-38% faster than JR 2.00001. On programs that require the generation of many different operation classes such as op_intXintToint and op_charXintXbooleanTodouble, JR 2.00005 compiles 52% faster than JR 2.00001. Elimination of files for capabilities and operations being generated at translation time is the main reason for the speedup. Table 6.1 below summarizes the average elapsed compilation time using JR 2.00001 and JR 2.00005. Appendix F contains the detailed statistics on compilation time and execution time performance of JR 2.00005.

On the other hand, the performance gain in compilation time does not bring a large performance loss in execution time. JR 2.00005 takes about the same time as JR 2.00001 to run programs with RMI and without RMI. The worst case among our benchmarks, timings/semP, shows a 13% increase in execution time. Table 6.2 shows the average elapsed execution time. The extra time in execution involves wrapping up primitive types to Objects in order to use the varargs feature and the generics feature, and unwrapping them to extract their actual values.

Benchmarks	**JR 2.00001**	**JR 2.00005**	**2.00005/ 2.00001**
timings/asynch	4.55	4.00	0.88
timings/ircall	4.62	3.99	0.86
timings/irnew	4.69	4.03	0.86
timings/locall	4.56	3.94	0.86
timings/loop	4.50	3.89	0.86
timings/msgcsw	4.85	4.02	0.83
timings/pcreate	4.93	3.97	0.81
timings/rend	4.87	4.05	0.83
timings/semP	4.88	3.90	0.80
timings/semV	4.81	3.94	0.82
timings/semcsw	4.97	4.03	0.81
timings/sems	4.83	3.92	0.81
simulationAllPPC (w/ 1000 voters)	5.81	3.89	0.67
simulationAllPPC2 (w/ 1000 voters)	6.27	3.95	0.63
misc/dp/dpVis	14.25	8.84	0.62
codeextract2.0/dfs/all	8.77	5.58	0.64
vm/basic	5.59	3.97	0.71
vm/basicchain	6.12	4.10	0.67
vm/callme	5.84	4.11	0.70
vm/many	4.93	3.93	0.80
vm/vmonvm	5.07	3.85	0.76
op_intXintToint	7.74	3.74	0.48
op_charXintXbooleanTodouble	7.77	3.72	0.48
		Average Ratio	0.75

Table 6.1: Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks.

Benchmarks	JR 2.00001	JR 2.00005	2.00005/ 2.00001
timings/asynch	5.57	5.75	1.03
timings/ircall	3.70	3.78	1.02
timings/irnew	5.49	3.86	0.70
timings/locall	3.67	3.74	1.02
timings/loop	3.07	3.17	1.03
timings/msgcsw	5.00	4.75	0.95
timings/pcreate	5.06	4.82	0.95
timings/rend	6.62	6.39	0.97
timings/semP	4.88	5.54	1.14
timings/semV	3.45	3.30	0.96
timings/semcsw	4.88	4.73	0.97
timings/sems	5.78	5.52	0.96
simulationAllPPC (w/ 1000 voters)	42.29	43.11	1.02
simulationAllPPC2 (w/ 1000 voters)	127.59	133.78	1.05
vm/basic	4.97	4.78	0.96
vm/basicchain	6.28	6.12	0.97
vm/callme	9.77	9.10	0.93
vm/many	26.74	24.09	0.90
vm/vmonvm	6.20	5.21	0.84
op_intXintToint	18.79	18.91	1.00
op_charXintXbooleanTodouble	18.77	18.65	0.99
		Average Ratio	0.97

Table 6.2: Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks.

Chapter 7

Conclusion and Future Work

This thesis discussed our work on porting the implementation of the JR translator and the runtime system from Java 1.4 to 5.0. We created new JR features from new Java features, introduced modifications to the current JR language, and fixed bugs in the old JR implementation. We followed a static merging scheme and verified our implementation using a series of test suites.

We compared the performance of JR 1.00061 and JR 2.00001 by running benchmarks to measure compilation time and execution time. The results show that JR 2.00001 performs better when compiling programs that do not use the RMI library. It takes a little longer than JR 1.00061 with *rmic* cache enabled to compile small programs that require RMI compilation, but the performance gain in the translator covers up the penalty for re-outputting when compiling large programs with RMI. The execution time of JR 2.00001 on programs that do not use the RMI library is slightly longer than that of the current release on small programs, but it is faster for larger programs that use **Object** collections. Although dynamic generation of stub classes hurts the performance of JR 2.00001, the difference from that of the current release is minor.

We also tried, with JR version 2.00005, to reduce the amount of code generated for each JR program by using generics and varargs. Although this version is not fully functional, it demonstrates a promising improvement in compilation time when tested with our benchmarks and it only introduces a small increase in execution time.

Our future work beyond this thesis includes completing the implementation of JR 2.00005 and fixing known bugs in JR 2.00001 and JR 2.00005. Further research on subtyping is needed to solve the bug in generic capabilities. Moreover, keeping JR up to date with Java is an on-going task.

Appendix A

Java Benchmarks Description

Benchmarks	Approx. # of lines	Description	Language features used / characteristics
Einterpreter	3083	An interpreter of a small imperative language	Inheritance and polymorphism
sequence	829	Manipulates lists, matrices, and pairs	Inheritance and polymorphism
BLOAT [2]	72377	Java bytecode optimizer	Large scale application source
MonteCarlo [9]	3073	A financial simulation	Inheritance, file I/O, Vector
RayTracer [9]	1224	3D ray tracer	Vector and array
Euler [9]	1170	Solves the time-dependent Euler equations	Vector and array
Gcold [4]	400	A benchmark to stress old-generation collections	Binary tree
Interesting [7]	819	A small set of simple benchmarks	Loops, array, field, int arithmetic, method call, exception, thread, I/O, String
Linpack [14]	1049	Measures the floating point performance of computers	Applet, awt
Proguard [17]	59278	Java class file shrinker, optimizer and obfuscator	Large scale application source

Table A.1: Brief description of each Java benchmark used in comparing the compilation time performance.

Benchmarks	**Description**	**Language features used**
mmseq	Multiplies two 900 x 900 matrices	Array
mergeSort [15]	Performs merge sort on 50 sets of integers with 900.000 elements each	Inheritance and polymorphism
LUFact [9]	Linpack benchmark with N = 2,000.	Array and double arithmetic operations
FFT [9]	Performs a one-dimensional forward transform of 2,097,152 complex numbers	Array and math library functions
SOR [9]	Performs 100 iterations of successive over-relaxation on a 2,000 x 2,000 grid	Array and loop
Search [9]	Solves a game of connect-4 on a 6 x 7 board using an alpha-beta pruning technique, evaluates a total of 34,517,760 positions	Loop, if-else
Euler [9]	Solves time-dependent Euler equations. A 96 x 384 mesh is employed	Vector and Array
MolDyn [9]	Models 2048 particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions	Loop, if-else, array, math library functions
MonteCarlo [9]	A financial simulation	Inheritance, file I/O, Vector
RayTracer [9]	3D ray tracer	Vector and arrays

Table A.2: Brief description of each Java benchmark used in comparing the execution time performance.

Appendix B

Java Performance Statistics

2.8G dual-processor				
	Java 1.4		**Java 5.0**	
	Average	**Variance**	**Average**	**Variance**
Elapsed Time (in seconds)	15.78	0.380815	15.81	0.401262
Throughput	25379.40	905574.3	25339.60	949083.3
1.4G uniprocessor				
	Java 1.4		**Java 5.0**	
	Average	**Variance**	**Average**	**Variance**
Elapsed Time (in seconds)	42.72	0.029308	40.77	0.431239
Throughput	9363.40	1416.8	9813.00	24174

Table B.1: Average elapsed exection time (in seconds) and throughput data collected from running VolcanoMark 2.5.

	Compilation Time in seconds			
Benchmarks	**Java 1.4**	**Variance**	**Java 5.0**	**Variance**
Einterpreter	1.03	0.00005	1.41	0.00063
sequence	0.67	0.00002	0.91	0.00003
BLOAT	17.60	0.00702	23.03	0.00438
MonteCarlo	0.92	0.00068	1.22	0
RayTracer	0.79	0	1.03	0.00003
Euler	0.85	0	1.16	0
Gcold	0.73	0	0.92	0.00003
Interesting	0.81	0.00083	1.10	0.00002
Linpack	0.88	0.00098	1.17	0.00072
Proguard	6.44	0.00087	8.44	0.00153

Table B.2: Average elapsed compilation time (in seconds) (on PC with 2.8G dual-processor) on selected Java benchmarks.

	Compilation Time in seconds			
Benchmarks	**Java 1.4**	**Variance**	**Java 5.0**	**Variance**
Einterpreter	2.04	0.00175	2.82	0.00128
sequence	1.31	0.00072	1.72	0.00170
BLOAT	36.92	0.00803	48.69	0.05705
MonteCarlo	1.79	0.00150	2.48	0.00072
RayTracer	1.51	0.00065	1.94	0.00037
Euler	1.72	0.00043	2.26	0.00253
Gcold	1.37	0.00115	1.78	0.00063
Interesting	1.61	0.00287	2.15	0.00112
Linpack	1.71	0.00082	2.30	0.00052
Proguard	12.72	0.00135	17.26	0.05528

Table B.3: Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected Java benchmarks.

	Execution Time in seconds			
Benchmarks	**Java 1.4**	**Variance**	**Java 5.0**	**Variance**
mmseq	20.92	0.15747	20.97	0.38943
mergeSort	16.96	0.02037	17.79	0.00243
LUFact	25.35	0.01790	26.20	0.02493
FFT	9.52	0.00122	10.04	0.01082
SOR	8.28	0.00083	8.13	0.00068
Search	29.42	0.01628	28.70	0.06853
Euler	15.73	0.00393	13.88	0.00212
MolDyn	4.34	0.00972	4.39	0.00722
MonteCarlo	11.39	0.00122	10.52	0.00093
RayTracer	9.74	0.00080	6.58	0.00212

Table B.4: Average elapsed execution time (in seconds) (on PC with 2.8G dual-processor) on selected Java benchmarks.

	Execution Time in seconds			
Benchmarks	**Java 1.4**	**Variance**	**Java 5.0**	**Variance**
mmseq	75.51	0.06948	76.27	0.17993
mergeSort	33.15	0.17068	34.39	0.04243
LUFact	44.73	0.00295	46.36	0.00863
FFT	17.31	0.01672	17.20	0.00118
SOR	17.56	0.00077	17.65	0.00057
Search	52.77	0.04417	56.92	0.18595
Euler	27.98	0.00207	25.51	0.00510
MolDyn	8.82	0.01093	9.01	0.00588
MonteCarlo	20.76	0.05332	18.83	0.00187
RayTracer	21.38	0.00275	14.14	0.00187

Table B.5: Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected Java benchmarks.

Appendix C

JR Benchmarks Description

Benchmarks	Approx. # of lines	Description	Language features used
timings/asynch	60	Performs asynchronous send/receive	call, operation, inni, send, receive
timings/ircall	68	Interclass call, no new process	call, operation, remote
timings/irnew	75	Interclass call with new process creation	call, operation, remote
timings/locall	59	Local call	call
timings/loop	57	Tests the overhead of an empty loop	call
timings/msgcsw	81	Message passing requiring context switch	send, receive, P, V, sem
timings/pcreate	64	Process create	P, V, sem, send
timings/rend	68	Rendezvous	send, call, inni, operation
timings/semP	60	Performs semaphore P operation	P, V, sem
timings/semV	57	Performs semaphore V operation	P, V, sem
timings/semcsw	79	Semaphore requiring context switch	send, P, V, sem
timings/sems	57	Performs semaphore operation P and V as a pair	P, V, sem
simulationAllPPC	41	An election simulation	process, inni, send, receive, operation
simulationAllPPC2	45	An election simulation using an array of operations	process, inni, send, receive, array of operation
op_intXintToint	17	Performs 2000 sends and receives on an operation that takes two int and returns int	send, receive
op_charXintXbooleanTodouble	17	Performs 2000 sends and receives on an operation that takes a char, an int, and a boolean and returns int	send, receive

Table C.1: Brief description of each JR benchmark that does not use the RMI library.

Benchmarks	Approx. # of lines	Description	Language features used
vm/basic	38	Creates a remote object on a JR virtual machine	remote object, vm, quiescence operation
vm/basicchain	53	Creates a remote object on a JR virtual machine then that object creates another remote object on another JR virtual machine	remote object, vm, operation
vm/callme	94	Invocation of remote object functions	remote object, vm, operation
vm/many	70	Creates 20 virtual machines, maximum of 3 at a time	remote object, vm array
vm/vmonvm	45	Creates a JR virtual machine on the same physical machine as another JR virtual machine	remote object, vm
misc/dp/dpVis	2178	Visualization of the dining philosophers problem	process, send, receive, sem, P, V, operations, capability, vm, remote object
codeextract2.0/dfs/all	644	Distributed file system	vm, remote object, inni, send, receive, reply, forward, capability
RW visualization	2678	Visualization of the readers writers problem	vm, remote object, inni, send, receive, reply, forward, capability

Table C.2: Brief description of each JR benchmark that uses the RMI library.

Appendix D

Parameters Used in Running JR Benchmarks

Benchmarks	Command Line Arguments
timings/asynch	10 10 1
timings/ircall	10 10 1
timings/irnew	10 10 1
timings/locall	10 10 1
timings/loop	10 10 1
timings/msgcsw	10 10 1
timings/pcreate	10 10 1
timings/rend	10 10 1
timings/semP	10 10 1
timings/semV	10 10 1
timings/semcsw	10 10 1
timings/sems	10 10 1
simulationAllPPC (w/ 1000 voters)	n/a
simulationAllPPC2 (w/ 1000 voters)	n/a
vm/basic	n/a
vm/basicchain	n/a
vm/callme	n/a
vm/many	20 3
vm/vmonvm	n/a
op_intXintToint	n/a
op_charXintXbooleanTodouble	n/a

Table D.1: Command line arguments provided for each selected benchmark.

Appendix E

JR 2.00001 Performance Statistics

	JR 1.00061		JR 2.00001	
Benchmarks	**Compilation Time (s)**	**Variance**	**Compilation Time (s)**	**Variance**
timings/asynch	2.66	0.00073	2.42	0.00008
timings/ircall	2.72	0.00108	2.48	0.00008
timings/irnew	2.77	0.00608	2.48	0.00003
timings/locall	2.71	0.00572	2.48	0.00005
timings/loop	2.68	0.00302	2.40	0.00063
timings/msgcsw	2.73	0.00518	2.48	0.00002
timings/pcreate	2.73	0.00128	2.42	0
timings/rend	2.77	0.00313	2.48	0
timings/semP	2.69	0.00653	2.41	0.00047
timings/semV	2.70	0.00733	2.41	0.00043
timings/semcsw	2.76	0.00265	2.48	0.00003
timings/sems	2.69	0.00127	2.42	0.00005
simulationAllPPC (w/ 1000 voters)	4.33	0.00063	2.71	0.00083
simulationAllPPC2 (w/ 1000 voters)	4.35	0.00128	2.71	0.00063

Table E.1: Average elapsed compilation time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that do not use the RMI library. *Rmic* cache was not used during translation.

	JR 1.00061		JR 2.00001	
Benchmarks	**Compilation Time (s)**	**Variance**	**Compilation Time (s)**	**Variance**
timings/asynch	7.10	0.00477	4.55	0.00077
timings/ircall	7.12	0.00133	4.62	0.00132
timings/irnew	7.19	0.00038	4.69	0.03588
timings/locall	7.05	0.00057	4.56	0.00007
timings/loop	7.00	0.00062	4.50	0.00067
timings/msgcsw	7.11	0.00048	4.85	0.00808
timings/pcreate	7.07	0.00017	4.93	0.01177
timings/rend	7.13	0.00032	4.87	0.00273
timings/semP	7.03	0.00123	4.88	0.03158
timings/semV	7.02	0.00093	4.81	0.00368
timings/semcsw	7.11	0.00078	4.97	0.01592
timings/sems	7.06	0.00117	4.83	0.00605
simulationAllPPC (w/ 1000 voters)	9.87	0.00113	5.81	0.02048
simulationAllPPC2 (w/ 1000 voters)	9.91	0.00262	6.27	0.81008

Table E.2: Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library. *Rmic* cache was not used during translation.

	JR 1.00061 on 1.4 VM		JR 2.00001 on 5.0 VM	
Benchmarks	**Execution Time (s)**	**Variance**	**Execution Time (s)**	**Variance**
timings/asynch	2.77	0.00125	2.76	0.00043
timings/ircall	2.06	0.00003	2.09	0.00002
timings/irnew	2.76	0.00080	2.74	0.00003
timings/locall	2.05	0	2.07	0.00058
timings/loop	1.81	0	1.84	0.00003
timings/msgcsw	2.65	0.00360	2.60	0.00147
timings/pcreate	2.45	0.00108	2.45	0.00003
timings/rend	3.17	0.00092	3.15	0.00087
timings/semP	2.71	0.00003	2.74	0.00002
timings/semV	1.87	0	1.90	0.00003
timings/semcsw	2.65	0.00360	2.65	0.00135
timings/sems	2.71	0.00003	2.74	0.00027
simulationAllPPC (w/ 1000 voters)	21.66	0.07182	18.33	0.00273
simulationAllPPC2 (w/ 1000 voters)	51.90	0.00893	48.61	0.34363

Table E.3: Average elapsed execution time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that do not use the RMI library.

	JR 1.00061 on 1.4 VM		JR 2.00001 on 5.0 VM	
Benchmarks	**Execution Time (s)**	**Variance**	**Execution Time (s)**	**Variance**
timings/asynch	5.52	0.05223	5.57	0.00503
timings/ircall	3.61	0.00063	3.70	0.00103
timings/irnew	5.35	0.00188	5.49	0.00082
timings/locall	3.97	0.14403	3.67	0.00097
timings/loop	3.00	0.00032	3.07	0.00007
timings/msgcsw	4.58	0.00055	5.00	0.00603
timings/pcreate	4.64	0.00157	5.06	0.02057
timings/rend	6.19	0.00248	6.62	0.00803
timings/semP	5.31	0.01232	4.88	0.03158
timings/semV	3.26	0.01903	3.45	0.02557
timings/semcsw	4.51	0.03618	4.88	0.01933
timings/sems	5.40	0.00057	5.78	0.00933
simulationAllPPC (w/ 1000 voters)	49.21	0.08953	42.29	0.58957
simulationAllPPC2 (w/ 1000 voters)	143.04	0.16077	127.59	0.08372

Table E.4: Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that do not use the RMI library.

	JR 1.00061		JR 1.00061 (w/ *rmic* cache)		JR 2.00001	
Benchmarks	**Compilation Time (s)**	**Variance**	**Compilation Time (s)**	**Variance**	**Compilation Time (s)**	**Variance**
vm/basic	4.26	0	2.66	0.00008	2.73	0.00003
vm/basicchain	4.99	0.00072	2.96	0.00002	2.92	0.00003
vm/callme	4.38	0	2.71	0.00003	2.81	0.00088
vm/many	2.63	0.00252	2.19	0.00008	2.36	0.00008
vm/vmonvm	2.64	0	2.06	0.00005	2.30	0.00002
misc/dp/dpVis	24.66	0.04635	12.56	0.02012	7.30	0.01267
codeextract2.0/dfs/all	14.08	0.00098	6.23	0.00057	4.49	0.01207
RW visualization	28.21	0.01548	16.71	0.00852	8.61	0.00212

Table E.5: Average elapsed compilation time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that use the RMI library.

	JR 1.00061		JR 1.00061 (w/ *rmic* cache)		JR 2.00001	
Benchmarks	**Compilation Time (s)**	**Variance**	**Compilation Time (s)**	**Variance**	**Compilation Time (s)**	**Variance**
vm/basic	10.05	0.00067	4.95	0.00012	5.59	0.02272
vm/basicchain	11.40	0.00158	5.48	0.00082	6.12	0.07450
vm/callme	10.22	0.00058	5.08	0.00148	5.84	0.07523
vm/many	6.99	0.00213	3.99	0.00108	4.93	0.01932
vm/vmonvm	6.73	0.00897	3.79	0.00058	5.07	0.12498
misc/dp/dpVis	45.79	5.56622	20.32	0.00393	14.25	0.03653
codeextract2.0/dfs/all	24.16	0.06438	11.17	0.00283	8.77	0.00103
RW visualization	44.86	0.01537	26.15	0.02153	16.59	0.00870

Table E.6: Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library.

	JR 1.00061 on 1.4 VM		JR 2.00001 on 5.0 VM	
Benchmarks	**Execution Time (s)**	**Variance**	**Execution Time (s)**	**Variance**
vm/basic	2.62	0.00087	2.70	0.00095
vm/basicchain	3.29	0.00215	3.34	0.00127
vm/callme	4.80	0.00308	4.93	0.00072
vm/many	11.90	0.00963	12.00	0.00012
vm/vmonvm	2.84	0.00003	2.88	0.00057

Table E.7: Average elapsed execution time (in seconds) (on PC with 2.8G dual-processor) on selected benchmarks that use the RMI library.

	JR 1.00061 on 1.4 VM		JR 2.00001 on 5.0 VM	
Benchmarks	**Execution Time (s)**	**Variance**	**Execution Time (s)**	**Variance**
vm/basic	4.65	0.04167	4.97	0.15388
vm/basicchain	5.90	0.00482	6.28	0.01605
vm/callme	9.02	0.02065	9.77	0.44165
vm/many	22.68	0.04993	26.74	0.59390
vm/vmonvm	5.00	0.03978	6.20	0.39583

Table E.8: Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected benchmarks that use the RMI library.

Appendix F

JR 2.00005 Performance Statistics

	JR 2.00001		JR 2.00005	
Benchmarks	**Average**	**Variance**	**Average**	**Variance**
timings/asynch	2.42	0.00008	2.11	0.00005
timings/ircall	2.48	0.00008	2.09	0.00103
timings/irnew	2.48	0.00003	2.11	0.00002
timings/locall	2.48	0.00005	2.04	0.00093
timings/loop	2.40	0.00063	2.01	0.00057
timings/msgcsw	2.48	0.00002	2.11	0.00007
timings/pcreate	2.42	0	2.07	0.00057
timings/rend	2.48	0	2.11	0.00003
timings/semP	2.41	0.00047	2.03	0.00152
timings/semV	2.41	0.00043	2.04	0.00072
timings/semcsw	2.48	0.00003	2.11	0.00003
timings/sems	2.42	0.00005	2.03	0.00135
simulationAllPPC (w/ 1000 voters)	2.71	0.00083	2.01	0.00068
simulationAllPPC2 (w/ 1000 voters)	2.71	0.00063	2.06	0.00018
misc/dp/dpVis	7.30	0.01267	4.31	0.00087
codeextract2.0/dfs/all	4.49	0.01207	2.80	0.00050
vm/basic	2.73	0.00003	2.11	0
vm/basicchain	2.92	0.00003	2.13	0.00063
vm/callme	2.81	0.00088	2.12	0.00003
vm/many	2.36	0.00008	2.05	0.00002
vm/vmonvm	2.30	0.00002	1.99	0.00003
op_intXintToint	2.53	0.00003	1.99	0.00012
op_charXintXbooleanTodouble	2.54	0.00007	1.99	0.00002

Table F.1: Average elapsed compilation time (in seconds) (on PC with 2.8G dual-processor) on selected JR benchmarks.

	JR 2.00001		JR 2.00005	
Benchmarks	**Average**	**Variance**	**Average**	**Variance**
timings/asynch	4.55	0.00077	4.00	0.00477
timings/ircall	4.62	0.00132	3.99	0.00075
timings/irnew	4.69	0.03588	4.03	0.00172
timings/locall	4.56	0.00007	3.94	0.00270
timings/loop	4.50	0.00067	3.89	0.00183
timings/msgcsw	4.85	0.00808	4.02	0.00103
timings/pcreate	4.93	0.01177	3.97	0.00165
timings/rend	4.87	0.00273	4.05	0.00750
timings/semP	4.88	0.03158	3.90	0.00183
timings/semV	4.81	0.00368	3.94	0.00148
timings/semcsw	4.97	0.01592	4.03	0.00243
timings/sems	4.83	0.00605	3.92	0.00102
simulationAllPPC (w/ 1000 voters)	5.81	0.02048	3.89	0.00403
simulationAllPPC2 (w/ 1000 voters)	6.27	0.81008	3.95	0.00132
misc/dp/dpVis	14.25	0.03653	8.84	0.00152
codeextract2.0/dfs/all	8.77	0.00103	5.58	0.00113
vm/basic	5.59	0.02272	3.97	0.00095
vm/basicchain	6.12	0.07450	4.10	0.00180
vm/callme	5.84	0.07523	4.11	0.00470
vm/many	4.93	0.01932	3.93	0.00117
vm/vmonvm	5.07	0.12498	3.85	0.00050
op_intXintToint	7.74	0.00360	3.74	0.00360
op_charXintXbooleanTodouble	7.77	0.00080	3.72	0.00073

Table F.2: Average elapsed compilation time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks.

	JR 2.00001		JR 2.00005	
Benchmarks	**Average**	**Variance**	**Average**	**Variance**
timings/asynch	2.76	0.00043	2.80	0.00087
timings/ircall	2.09	0.00002	2.11	0.00003
timings/irnew	2.74	0.00003	2.13	0.00108
timings/locall	2.07	0.00058	2.09	0.00080
timings/loop	1.84	0.00003	1.86	0.00003
timings/msgcsw	2.60	0.00147	2.69	0.00440
timings/pcreate	2.45	0.00003	2.48	0.00063
timings/rend	3.15	0.00087	3.17	0.00072
timings/semP	2.74	0.00002	2.76	0.00002
timings/semV	1.90	0.00003	1.92	0.00003
timings/semcsw	2.65	0.00135	2.68	0.00297
timings/sems	2.74	0.00027	2.74	0.00117
simulationAllPPC (w/ 1000 voters)	18.33	0.00273	18.32	0.00657
simulationAllPPC2 (w/ 1000 voters)	48.61	0.34363	51.74	0.25317
vm/basic	2.70	0.00095	2.63	0.00057
vm/basicchain	3.34	0.00127	3.24	0.00003
vm/callme	4.93	0.00072	4.49	0.00080
vm/many	12.00	0.00012	11.42	0.01927
vm/vmonvm	2.88	0.00057	2.82	0.00003
op_intXintToint	7.56	0	7.63	0.00252
op_charXintXbooleanTodouble	7.60	0.00648	7.58	0.00573

Table F.3: Average elapsed execution time (in seconds) (on PC with 2.8G dual-processor) on selected JR benchmarks.

	JR 2.00001		JR 2.00005	
Benchmarks	**Average**	**Variance**	**Average**	**Variance**
timings/asynch	5.57	0.00503	5.75	0.07875
timings/ircall	3.70	0.00103	3.78	0.00155
timings/irnew	5.49	0.00082	3.86	0.00037
timings/locall	3.67	0.00097	3.74	0.00047
timings/loop	3.07	0.00007	3.17	0.00070
timings/msgcsw	5.00	0.00603	4.75	0.00527
timings/pcreate	5.06	0.02057	4.82	0.00082
timings/rend	6.62	0.00803	6.39	0.00297
timings/semP	4.88	0.03158	5.54	0.00338
timings/semV	3.45	0.02557	3.30	0.00038
timings/semcsw	4.88	0.01933	4.73	0.00288
timings/sems	5.78	0.00933	5.52	0.00143
simulationAllPPC (w/ 1000 voters)	42.29	0.58957	43.11	0.03517
simulationAllPPC2 (w/ 1000 voters)	127.59	0.08372	133.78	5.54758
vm/basic	4.97	0.15388	4.78	0.00010
vm/basicchain	6.28	0.01605	6.12	0.00775
vm/callme	9.77	0.44165	9.10	0.01237
vm/many	26.74	0.59390	24.09	0.00873
vm/vmonvm	6.20	0.39583	5.21	0.00075
op_intXintToint	18.79	0.00692	18.91	0.00230
op_charXintXbooleanTodouble	18.77	0.00967	18.65	0.04112

Table F.4: Average elapsed execution time (in seconds) (on PC with 1.4G uniprocessor) on selected JR benchmarks.

Bibliography

[1] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language : Concurrency in Practice*. Benjamin/Cummings Pub. Co., 1993. http://www.cs.arizona.edu/sr/.

[2] *BLOAT, The Bytecode-Level Optimizer and Analysis Tool*. http://www.cs.purdue.edu/s3/projects/bloat/.

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

[4] *GCOld: a benchmark to stress old-generation collection*. http://www.experimentalstuff.com/Technologies/GCold/index.html.

[5] *Generics Tutorial*, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

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

[7] *Bill and Paul's Excellent UCSD Benchmarks for Java*. http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.

[8] *J2SE 5.0 Performance White Paper*, 2005. http://java.sun.com/performance/reference/whitepapers/5.0_performance.html.

[9] *The Java Grande Forum Sequential Benchmarks*. http://www.epcc.ed.ac.uk/javagrande/sequential.html.

[10] *JCilk — A Java-Based Multithreaded Programming Language*. http://publications.csail.mit.edu/abstracts/abstracts05/jsd_angelee_cel/jsd_angelee_cel.html.

[11] *JDK 5.0 Documentation*, 2004. http://java.sun.com/j2se/1.5.0/docs/index.html.

[12] *Jiazzi*. http://www.cs.utah.edu/plt/jiazzi.

[13] M. Frigo C.E. Leiserson and K.H.Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223. Montreal, Quebec, Canada, June 1998.

[14] *Linpack Benchmark – Java Version*. http://www.netlib.org/benchmark/linpackjava/.

[15] *Merge Sort in Java*. http://www.cs.uiowa.edu/~sriram/21/fall05/ExamplePrograms/ReaderFiles/Chap06/mergeSort/mergeSort.java.

[16] Ronald A. Olsson and Aaron W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer International series in engineering and computer science; SECS 774. Boston : Kluwer Academic, 2004. http://www.cs.ucdavis.edu/~olsson/research/jr/.

[17] *Proguard*. http://proguard.sourceforge.net/.

[18] *Sun Developer Network*, 1994-2005. http://java.sun.com/.

[19] *The GNU compiler for the Java programming language*, Oct 2004. http://gcc.gnu.org/java.

[20] *Volcano: The Volcano Report And Benchmark Tests*. http://www.volano.com/benchmarks.html.