`javar`: A Prototype Java Restructuring Compiler*

Aart J.C. Bik, Juan E. Villacis and Dennis B. Gannon
Lindley Hall 215, Computer Science Department, Indiana University
Bloomington, Indiana 47405-4101, USA
ajcbik@extreme.indiana.edu

Abstract

This paper describes the prototype restructuring compiler `javar`, which can be used to make implicit parallelism in a Java program explicit by means of multi-threading. Although the prototype does *not* provide a complete Java front-end (unicode escapes are not supported and only limited semantic analysis has been implemented) and relies *completely* on the identification of 'implicit' parallelism by means of annotations, we hope that the research tool still provides sufficient functionality to make the parallelization of Java programs less complex and less error-prone.

1 Introduction

In this paper, we describe our prototype research tool `javar`, a simple Java restructuring compiler (written in C) that can automatically transform a Java program into a form in which implicit parallelism is made explicit by means of multi-threading [1, 6, 7, 9, 12, 14, 15, 16, 17, 19]. Although, ideally, the *detection* of implicit parallelism is also done automatically, in the prototype the programmer must identify all 'implicit' parallelism by means of annotations. The transformations that make this parallelism explicit, however, are done automatically by `javar`, which simplifies the task of the programmer substantially and makes the parallelization less error-prone.

In figure 1, we illustrate our approach to automatically exploiting implicit parallelism in Java programs. A Java program `MyClass.java` is used as input of our source to source Java restructuring compiler `javar`. Thereafter, the compiler transforms the input program into a form that uses the multi-threading mechanism of Java to make all implicit parallelism explicit. Because parallelism is expressed in Java itself, the transformed program can still be compiled into byte-code by any Java compiler (`javac` in the figure), and subsequently interpreted by any byte-code interpreter (`java` in the figure, or, alternatively, an interpreter that is embedded in a browser or appletviewer). Speedup can be obtained on any platform that supports the true parallel execution of threads.

Because filenames are essential in Java, the transformed program is stored in the file `MyClass.java` after a copy of the original program has been saved in a file `MyClass.orig`. In case changes to the original program are required, this latter file must be renamed into `MyClass.java` again before `javar` can be re-applied to the program.

2 `javar` Command

The compiler `javar` is invoked as follows:

```
javar [-cnopvw] [file ...]
```

*This project is supported by DARPA under contract ARPA F19628-94-C-0057 through a subcontract from Syracuse University.

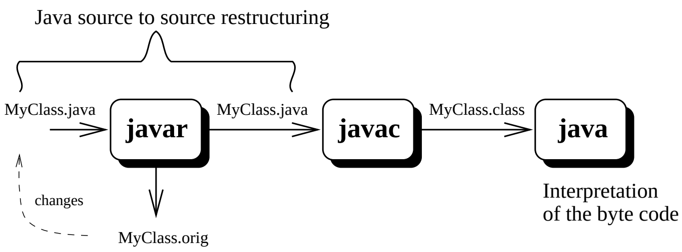

Figure 1: Restructuring, compiling, and interpreting

An arbitrary number of files may be supplied (e.g. '`javar Class1.java Class2.java`' or '`javar *.java`'), and the options may be given in any order (e.g. '`javar -op *.java`', '`javar -o -p *.java`', or '`javar -o *.java -p`').

The options have the following meaning:

- `-c` : Enforces `javar` to preserve comments in the program text.[1]
- `-n` : Enforces `javar` to print the derived meaning of names in a program.
- `-o` : Enforces the generation of output for each compilation unit.
- `-p` : Enforces the generation of parallel code.
- `-v` : Enforces `javar` to print messages about its activities.
- `-w` : Enforces the generation of an HTML presentation of all compilation units.

Without the '`-o`' option, `javar` can be used for syntax and limited semantic analysis (e.g. '`javar MyClass.java`'), and for parallel code generation without output (e.g. '`javar -p MyClass.java`'). With the '`-o`' option, `javar` renames each input file `MyClass.java` into `MyClass.orig` and generates the output for this file in `MyClass.java` again, so that `javar` can be used as a 'pretty-printer' that presents each Java program in some rigid format (e.g. '`javar -o *.java`'), or as a parallelizing compiler (e.g. '`javar -op *.java`'). Finally, the '`-w`' option can be used to construct an HTML representation of all compilation units. This HTML representation (with root file '`index.html`') is placed in the directory defined by the environment variable `JAVAR_HTML` (or the current directory by default).

If no files are specified, `javar` reads from standard input and, if the option '`-o`' is given, writes to standard output, which enables the use of `javar` used in Unix pipes or input-output redirections [10] (e.g. `javar -o < filein > fileout`). All errors, messages, and output enforced by the '`-v`' or '`-n`' option are written to standard error. If `javar` cannot rename a file `MyClass.java` into `MyClass.orig` because this latter file already exists, `javar` assumes that the conversion already has been done. In this case, the programmer must first rename `MyClass.orig` into `MyClass.java` if the former file indeed contains the original program, or, otherwise, the programmer must first deal with the conflicting file `MyClass.orig` in some other manner.

A nonzero exit status is returned by `javar` if an error is encountered in *any* of the compilation units, and a zero exit status otherwise. Invoking `javar` as a rapid syntax and limited semantic analyzer before the Java compiler `javac` for all compilation units defined by the arguments of a UNIX shell script, for example, can be implemented as follows:

```
javar $*
if [ $? -eq 0 ]
then
  javac -O $*
fi
```

[1] In the current version, only comments that appear before class-, interface-, constructor-, and member-declarations are preserved (and these comments may float a little in the output program text).

3 Parallel Loops

As illustrated below, a parallel loop is identified using the loop annotation '/*par*/' (which is simply ignored by other Java compilers) before a for-loop:

```
/*par*/
for (int i = 500; i > 0; i -= 3)
   body(i);
```

3.1 Loop Annotations

A comma-separated list of the following constructs can be optionally used in between the '/*par' and the '*/' of a loop annotation:

- 'threads = <int>' : Defines the number of threads that must be used for parallel execution (e.g. '/*par threads = 2 */'). The default value is 4.
- one of 'block', 'cyclic', or 'guided' : Defines either block scheduling, cyclic scheduling, or guided self-scheduling as the scheduling policy for the parallel loop (see e.g. [18, 20, 21, 22]). The default policy is guided self-scheduling.
- 'multi = <int>' : Enforces the generation of multi-version code where at run-time either the serial or parallel version of the loop is executed depending on the number of iterations. The parameter defines the threshold (e.g. '/*par multi = 150 */'). The default value is 0 (i.e. no multi-version code).
- '(<name>)' : Enforces the generation of a parallel loop that catches any exception assignable to the specified exception that is thrown in one of the threads of the loop, and explicitly re-throws this exception in the thread that initiated the parallel loop.
- 'sync = <int>' : Defines the number of synchronization variables that are required for DOACROSS-like execution (e.g. '/*par sync = 2 */'). The default value is 0 (i.e. DOALL-like execution).
- one of 'sups_wait' or 'busy_wait': Defines either a suspended-waiting (the default) or a busy-waiting implementation of random synchronization in a DOACROSS-like loop.

An arbitrary stride may be used for DOALL-loops, whereas each DOACROSS-loop must have stride-1. Within the body of a DOACROSS-like parallel loop, the following two loop annotations can be used to insert random synchronization (see e.g. [13, 21, 22]) that enforces certain loop-carried data dependences:

- '/*post <int> */' : Annotation '/*post k */' inserts a **post**-statement that during each iteration i will set the ith bit of the kth synchronization variable (where $0 \leq k < n$ in case 'sync = n' has been set).
- '/*wait <int>, <expr> */' : Annotation '/*wait k, j */' inserts a **wait**-statement that will block until the jth bit of the kth synchronization variable has been set (where $0 \leq k < n$ in case 'sync = n' has been set).

For example, the annotations shown below convert the i-loop into a cyclic-scheduled DOACROSS-loop in which a busy-waiting implementation of random synchronization enforces the static flow dependences $S_1 \delta_< S_2$ and $S_3 \delta_< S_2$ with dependence distance 8 and 9, respectively:

```
       /*par sync = 2, cyclic, busy_wait */
       for (int i = 9; i < N-8; i++) {
S1:      a[i+8] /= 3.0D;
         /*post 0 */                        // post(ASYNC, i)
         /*wait 0, i-8 */                   // wait(ASYNC, i-8)
         /*wait 1, i-9 */                   // wait(CSYNC, i-9)
S2:      d[i]   += c[i-9] * a[i];
S3:      c[i]   /= 8.0D;
         /*post 1 */                        // post(CSYNC, i)
       }
```

Because random synchronization annotations are effectively handled as statements by `javar`, all labels that appear *directly before* such annotations will be associated with the corresponding statements after parallelization (rather than with the first real Java statement in the original program).

3.2 Actual Loop Parallelization

In the prototype version of `javar`, it is *the responsibility of the programmer* to identify the DOALL- and DOACROSS-like and parallel loops in a program and, for the latter kind of loops, to provide the appropriate random synchronization annotations. Each parallel loop in a Java program is simply processed by `javar` without further ascertaining whether the parallelization actually preserves the semantics of the original program.

The source to source restructuring compiler `javar` uses the method described in [2] to make loop parallelism explicit by means of multi-threading. Given a parallel loop, the following steps are taken: (i) the annotation is examined, (ii) the loop is analyzed to determine whether the method of [2] is applicable and, (iii) parallel code is generated.

3.2.1 Examination of Parallel Loop Annotation

A compile-time error occurs, if no or an invalid value is supplied for any of the constructs '`threads`', '`sync`', or '`multi`', in case a value is supplied if none is required, in case an unknown keyword appears in the loop annotation, or if an annotation '`/*par*/`' does not appear directly before a for-loop.

For example, `javar` generates the following error for the fragment shown below:

```
 ...
 /*par sync=10, cpu=5 */                // <- line 14
 for (int i = 0; i < N; i++)
    a[i] = 10.0;
 ...

> javar -op Loop.java
javar :: [line 14, Loop.java] cannot process cpu in loop annotation
javar :: no output for Loop.java
```

Specifications of parallel loops *within* another parallel loop are ignored (also in the serial branch of multi-version code). Likewise, `javar` ignores random synchronization annotations that appear *outside* parallel loops or that operate on undeclared synchronization variables.

3.2.2 Analysis of Loop

If the '`-p`' flag is given, the following tests are performed for each parallel loop to determine whether the method of [2] is applicable:

(1) The loop header must have the form '`for (int i = init; e1; e2)`', where
`e1` is any of `i<high`, or `i<=high`, `high>i`, or `high>=i` and `e2` is `i++` or `i+=stride`, or
`e1` is either `i>low`, `i>=low`, `low<i`, or `low<=i` and `e2` is `i--` or `i-=stride`.
Loop index `i` may not occur in `init`, `low`, or `high`.

Failure to adhere to this form results in a compile-time error. Below we give some examples of loops that must be rewritten before `javar` can be used for parallelization:

Original Loop	Rewritten Loop	Comments

`int i;`
`for (i = 0; i < 100; i++)`
`body(i);`
	`for (int i = 0; i < 100; i++)`
	`body(i);`
		use a locally declared loop index

`for (int k = 500; k >= 3-2*k; k--)` `body(k);`	`for (int k = 500; k >= 1; k--)` `body(k);`	use simple comparisons

In order to allow `javar` to re-use the loop bound expressions in several integer arithmetic operations (for example, to implement loop normalization), the following constraints are also imposed on the loop:

(2) The initialization expression `init` and the bound expression `low` or `high` must be *loop-invariant integer* expressions. If used, `stride` must be a positive integer literal.

As illustrated below, loops that do not adhere to these constraints must be rewritten first:

Original Loop	Rewritten Loop	Comments

`for (int i = low++; i <= up; i++)`
`  body(i);`
	`int lw = low++;`
	`for (int i = lw; i <= up; i++)`
	`  body(i);`
		avoid side effects in
		initialization expression

`for (int d = 0.0; d < 11.5; d += 1.0)` `  body(d);`	`for (int d = 0; d < 12; d++)` `  body(d);`	use integer bounds and stride

`int s = 3;`
`for (int i = 100; i > 0; i -= s)`
`  body(i);`
	`int s = 3;`
	`for (int i = 100; i > 0; i -= 3)`
	`  body(i);`
		use positive integer
		literal as stride

In this prototype implementation, `javar` conservatively assumes that field accesses, array element accesses, and method invocations are not loop invariant. Moreover, `javar` assumes that a *non-local* variable may be changed by *any* method invocation in the loop-body. For example, below we present a fragment in which `javar` cannot detect the trivial loop invariance of the upper bound `N * N`:

```
public class Init {
  int[] a;
  int   N;
  public void init() {
    int[] b;
    /*par block */
    for (int k = 0;  k < N * N; k++)
      b[k] = Math.abs( a[k] );                  // <- line 9
  }
}

> javar -op Init.java
javar :: [line 9, Init.java] parallelization prohibited by potential change of N in Math.abs()
javar :: no output for Init.java
```

This problem is easily resolved, however, by storing the loop invariant upper bound in a local variable '`int b = N * N;`' and replacing the comparison with '`k < b`', because `javar` sees that the local variable cannot be changed during execution of the loop. Moreover, because `javar` may fail to determine the type of an initialization or bound expression, in some cases a warning informs the programmer that the parallelization is only correct under the assumption that loop bounds are integers.

The method of [2] also imposes the following constraint on parallel loops:

(3) The values of all local variables that are declared *outside* the loop, but referred to *within* the loop-body must remain unaltered during execution of the loop (thereby *not* counting any changes to elements or fields of array or object references).

Just as an academic example, `javar` can still apply loop parallelization to the following `i`-loop because the value of `s`, of `a` itself, and of `o` itself remain unaltered during execution of the loop (although the loop-carried output dependence on `o.field` makes it is unlikely that parallelization of this loop is truly desired):

```
void method(double s) {
  SomeObject o = new SomeObject();
  double[][] a = new double[100][100];
  /*par*/
  for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
      a[i][j] = s * o.field++;
}
```

Parallelization would be disabled if a statement like '`s++`', '`a = ...`' or '`o = ...`' would occur in the loop-body (note, however, that in such cases there is even less likely any parallelism in the loop). This constraint is imposed because the parallelization performed by `javar` in essence consists of letting a number of threads execute a newly generated class or instance method for different subsets of iterations.

All dynamically allocated data, class and instance variables, the loop index and all variables declared *inside* the loop-body remain visible in this new method. The value of local data declared outside the loop but referred to within the loop-body, however, must be explicitly passed to this new method. Constraints (3) avoid the requirement to propagate changes of such local variables back to the main thread.

Finally, to avoid problems in transferring control back to the initial thread, `javar` imposes the following constraint on parallel loops (typically only violated in serial loops anyway):

(4) The loop-body does not contain a `break`-, `continue`-, or `return`-statement.

Loop parallelization can be applied, however, to loops in which exceptions may be thrown (either implicitly or explicitly), although the programmer must list all exceptions that have to be explicitly transferred back to the thread that initiated the parallel loop.

In the following fragment, for instance, the programmer has specified that in the parallel loop, exception `RuntimeException` must be explicitly transferred back to the initial thread:

```
static void method(int len, int[] a, int[] b) {
  try {
    /*par (RuntimeException) */
    for (int i = 0; i < len; i++)
      a[i] = 1 / b[i];
    }
    catch(RuntimeException e) { System.out.println("a has not been initialized"); }
}
```

If an exception that is not assignable [9] to any of the exceptions listed in the loop annotation may be thrown during parallel execution of a loop, then (i) in case of a *checked* exception, the transformed program is *not* valid Java because `javar` did not add a `throws` declaration to the method used to execute different iterations of the loop in parallel, or (ii) in case of an *unchecked* exception, one of the threads may invoke the method `uncaughtException()` at run-time. Likewise, if a *checked* exception that is not explicitly handled in the context of the parallel loop is listed, the transformed program is also not valid Java. For all exceptions that are explicitly transferred back to the initial thread, the programmer must be aware that because the order in which iterations of a parallel loop are executed is completely unpredictable, *no* assumptions about which iterations have or have not been executed successfully can be made in any subsequent explicit exception handling.

3.2.3 Parallel Code Generation

Eventually, if the flags '`-op`' are given and all parallel loops satisfy the constraints given in the previous section, `javar` generates a Java program in which parallelism is made explicit using multi-threading. This new program requires the package '`parlooppack`' (supplied in the `EXAMPLES` directory). Make sure that this package, containing some auxiliary classes that support the parallel execution of loops, is accessible during compilation and execution of the new parallel program (see [2] for a detailed presentation of this package).

4 Parallel Multi-way Recursive Methods

The prototype `javar` also provides some elementary support to exploit parallelism in so-called parallel multi-way recursive methods. We refer to a *non*-private, *un*-synchronized, and *final* method (i.e. overriding or hiding is not possible) of the following form, where the different recursive method invocations in between executing `pre_code` and `post_code` can be done in parallel as a **parallel multi-way recursive method**:

```
class MyClass {
  ...
  [qualifiers] type myMethod(type1 f1, ..., typek fk) {
     if (cond)
       ... alt_code ...
     else {
       ... pre_code ...
       /*par_inv*/
       {
         r1 = target1.myMethod(a11, ..., a1k)
         ...
         rn = targetn.myMethod(an1, ..., ank)
        }
       ... post_code ...
     }
  }
}
```

Each `targeti` either denotes (implicitly) '`MyClass`' if `myMethod()` is a class method, or otherwise an arbitrary variable of type `MyClass`, including (an implicit) `this`. If `myMethod()` is a `void`-method, there are no assignments to the different `ri`. As illustrated above, the sequence of self-invocations that may be done in parallel is identified using the method annotation '`/*par_inv*/`' before a block that contains these self-invocations.

4.1 Method Annotations

A comma-separated list of the following constructs can be optionally used in between the '`/*par_inv`' and the '`*/`' of a method annotation:

- '`cut_depth = <int>`' : Defines the depth in the method invocation tree until which new threads must be forked to execute self-invocations in parallel (e.g. '`/*par cut_depth = 2 */`'). The default value is 1.
- one of '`no_serial_copy`' or '`serial_copy`' : Defines whether after the cut-depth, a method with some overhead may be used for executing the remaining invocations in a serial fashion (the default), or whether this overhead must be avoided at the expense of some code duplication by executing a copy of the original method at this stage.
- '`( <name> )`' : Enforces the generation of a parallel method that catches any exception assignable to the specified exception that is thrown in one of the threads of the method, and explicitly re-throws this exception in the thread that initiated the parallel method.

Several sequences of parallel self-invocations may be defined in one method (although only the optional constructs of the lexically first sequence are used during code generation). Parallel loops that appear within parallel methods and vice versa are simply ignored.

4.2 Actual Method Parallelization

As for loops, it is *the responsibility of the programmer* to identify parallel methods. The prototype uses the method described in [2] to make this method parallelism explicit by means of multi-threading without ascertaining whether the parallelization is actually valid. Currently, code is generated that uses a simple static allocation scheme to assign parallel method invocations to threads (some initial experimentation with dynamic allocation schemes failed due to the overhead of passing work to other threads).

In this release of `javar`, only straight-line code consisting of self-invocations, possibly assigning to *simple* variables in case of a method with a result, may occur in the sequence of parallel self-invocations, as is illustrated below:

void-method:

```
/*par_inv cut_depth=2*/
{
  myMethod(a11, ..., a1k);
  myMethod(a21, ..., a2k);
}
```

method with result:

```
/*par_inv cut_depth=2*/
{
  r1 = myMethod(a11, ..., a1k);
  r2 = myMethod(a21, ..., a2k);
}
```

Furthermore, the type of each actual argument `aij` must be equal to the type `typej` of the formal argument `fj` of `myMethod()` (because `javar` does currently not account for method invocation conversion [9, ch5], explicit casting may be required to obtain a valid sequence).

5 HTML Representation

If the '`-w`' option is given, an HTML representation of all compilation units that are supplied to the `javar` command is generated. This HTML representation (with root file '`index.html`') is placed in the directory defined by the environment variable `JAVAR_HTML` (or the current directory by default).

Constructing an HTML representation in `/tmp` for all Java compilation units that appear in the current directory and all its sub-directories, for instance, can be done in UNIX as follows:

```
setenv JAVAR_HTML /tmp
javar -w `find . -type f -name "*.java" -print`
```

An HTML representation that is organized into packages is constructed for the source code of all these compilation units, together with an HTML representation of a list of all classes and methods (with signatures) that allows for easy access into the source code representation.

6 Concluding Remarks

We would like to stress that this prototype version of `javar` does *not* provide a full front-end for the Java programming language (unicode escapes are not supported and only a limited semantic analysis of the program is performed). Therefore, we suggest that applying `javar` to a program is postponed until after a program has been thoroughly tested using a full Java compiler (on the other hand, because `javar` usually runs faster than the interpreted execution of `javac`, our prototype can be used as a rapid syntax and limited semantic checker during initial editing of a program). Moreover, the prototype cannot detect implicit parallelism automatically by means of data dependence analysis. Instead it relies completely on the identification of 'implicit' parallelism by means of annotations. Finally, although very unlikely, due to the local character of the transformations, conflicts between new identifiers and other identifiers that have to be resolved by the programmer could arise.

Despite the shortcomings of the prototype, however, we hope that this research tool provides sufficient functionality to make the parallelization of Java programs less complex and less error-prone. The results of some initial experiments with `javar` are reported in [2]

New releases of our prototype Java restructuring compiler `javar` (and a prototype bytecode parallelization tool `javab`) will be made available at the HP-Java page at the Indiana University (`http://www.extreme.indiana.edu/hpjava/`). Please send all your comments, bug reports, experiences, and suggestions to: `ajcbik@extreme.indiana.edu`

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[2] Aart J.C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.

[3] Zoran Budimlic and Ken Kennedy. Optimizing Java – theory and practice. *Concurrency, Practice and Experience*, 9(6):445–463, 1997.

[4] Bryan Carpenter, Yuh-Jye Chang, Geoffrey C. Fox, Donald Leskiw, and Xiaoming Li. Experiments with HP Java. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[5] Michal Cierniak and Wei Li. Optimizing Java bytecodes. *Concurrency, Practice and Experience*, 9(6):427–444, 1997.

[6] H.M. Deitel and P.J. Deitel. *Java, How to Program.* Prentice-Hall, 1997.

[7] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Sebastopol, CA, 1996.

[8] Geoffrey C. Fox and Wojtek Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modelling. *Concurrency, Practice and Experience*, 9(6):415–425, 1997.

[9] James Gosling, Bill Joy, and Guy Steele. *Java Programming Language.* Addison-Wesley, Reading, Massachusetts, 1996.

[10] Brian W. Kernighan. *The UNIX Programming Environment.* Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[12] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, Reading, Massachusetts, 1997.

[13] Samuel P. Midkiff. *The Dependence Analysis and Synchronization of Parallel Programs.* PhD thesis, C.S.R.D., 1993.

[14] Michael Morrison. *Java Unleashed.* Samsnet, Indianapolis, Indiana, 1996.

[15] Patrick Naughton. *The Java Handbook.* McGraw-Hill, New York, 1996.

[16] Patrick Niemeyer and Joshua Peck. *Exploring Java.* O'Reilly & Associates, Sebastopol, CA, 1996.

[17] Scott Oaks and Henry Wong. *Java Threads.* O'Reilly & Associates, Sebastopol, CA, 1997.

[18] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer, Boston, 1988.

[19] Glenn L. Vanderburg et al. *Tricks of the Java Programming Gurus.* Samsnet, Indianapolis, Indiana, 1996.

[20] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London, 1989.

[21] Michael J. Wolfe. *High Performance Compilers for Parallel Computers.* Addison-Wesley, Redwood City, California, 1996.

[22] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.