javab Manual (version 1.0BETA)

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

Abstract

This paper provides a manual for javab, a prototype tool that supports the automatic detection and exploitation of implicit loop parallelism in JVM bytecode. Implicit parallelism is made explicit by means of the multi-threading mechanism provided by the JVM. Automatically exploiting implicit parallelism at bytecode level can be done independently from the source program and platform from which the bytecode was obtained, and independently from the platform on which the bytecode eventually will run. The parallelized bytecode remains architectural neutral and may exhibit speedup on any platform that supports the true parallel execution of JVM threads.
This project is supported by DARPA under contract ARPA F19628-94-C-0057 through a subcontract from Syracuse University.

1 Introduction

In this paper, we describe our prototype research tool `javab`, which is a bytecode tool (written in C) that supports the automatic detection and automatic exploitation of implicit loop parallelism in bytecode, i.e. the architectural neutral instructions of the JVM (Java Virtual Machine). The tool performs a bytecode to bytecode transformation, where implicit loop parallelism is made explicit by means of the multi-threading mechanism provided by the JVM [12, 16, 15, 18, 20].

In figure 1, we illustrate the automatic exploitation of implicit loop parallelism in JVM bytecode. A binary file `MyClass.class` in the class file format [16] is used as input of `javab`. Thereafter, the compiler transforms the input program into a form that uses the JVM multi-threading mechanism to make all implicit parallelism explicit. Because parallelism is expressed in JVM bytecode itself, the transformed program can still be executed by any implementation of the JVM (`java` in the figure, or, alternatively, an interpreter that is embedded in a browser or appletviewer), and speedup can be obtained on any platform that supports the true parallel execution of threads. All transformations can be done independently from the source program and platform from which the bytecode was obtained, and independently from the platform on which the bytecode eventually will run. Hence, this approach allows the optimization of bytecodes that have been generated by the Java compiler `javac`, bytecodes that have been downloaded over a network, bytecodes generated by bytecode producing compilers for other high level languages, or bytecodes produced by a JVM assembler like Jasmin [17, 18].

Because filenames are essential in Java, the transformed program is stored in the file `MyClass.class` again, after a copy of the original program has been saved in `MyClass.old`.[1] Some auxiliary classes, support the parallel execution of loops, may also be generated by the bytecode parallelization tool `javab`

[1] Our prototype Java restructuring compiler `javar` [4, 6] renames a source file `MyClass.java` into `MyClass.orig`, so that both tools can be used without conflicts.

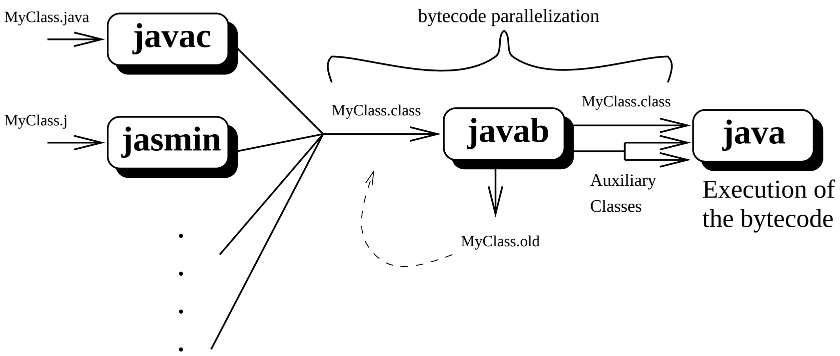

Figure 1: Automatic Bytecode Parallelization by `javab`

To keep compile-time limited, the prototype relies on relatively simple but generally also less expensive analysis, rather than doing, for instance, advanced data dependence analysis (see e.g. [3, 19, 22, 23, 24, 25]). Although currently an off-line bytecode to bytecode transformation is applied, keeping compile-time limited may become more important if the techniques are used for some form of JIT parallelization, i.e. bytecode parallelization directly prior to execution.

2 `javab` Command

The bytecode tool `javab` is invoked as follows:

```
javab [-bcdlnopqsv] [file ...]
```

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

The options have the following meaning for all specified class files:

- `-b` : Enforces `javab` to show information about the basic blocks in the JVM bytecode.
- `-c` : Enforces `javab` to summarize constant pool contents.
- `-d` : Enforces `javab` to print the JVM bytecode instructions for each method.
- `-l` : Enforces `javab` to preserve the exact handling semantics of *linking* exceptions.
- `-n` : Enables loop parallelization of *single* loops.
- `-o` : Enforces `javab` to write back each class file.
- `-p` : Enforces `javab` to automatically detect implicit parallel loops in the JVM bytecode.
- `-q` : Enables user interaction if inter-procedural information is required.
- `-s` : Enforces `javab` to show the stack states associated with each bytecode instruction.
- `-v` : Enforces `javab` to print messages about its activities.

A nonzero exit status is returned by `javab` if an error is encountered in *any* of the specified class files, and a zero exit status otherwise. All errors, messages, user queries, and output enforced by the options '`-bcdsv`' are written to standard error.

With only the '`-p`' option, `javab` merely attempts to detect implicit loop parallelism without actually transforming the input class files.

With the '-o' option, `javab` renames each class file `MyClass.class` into `MyClass.old` and generates the output for this file in `MyClass.class` again. If `javab` cannot rename a file `MyClass.class` into `MyClass.old` because this latter file already exists, `javab` assumes that the conversion already has been done. In this case, the programmer must first rename `MyClass.old` into `MyClass.class` if the former file indeed contains the original bytecode program, or, otherwise, the programmer must first deal with the conflicting file `MyClass.old` in some other manner. If no files are specified, then `javab` reads a class file from standard input and, if the option '-o' is also given, writes the class file back to standard output, which enables the use of `javab` in Unix pipes or input-output redirections [13] (e.g. `javab -op < filein > fileout`). Clearly, the '-o' option is only useful when combined with the parallelization '-p' option.

3 Automatic Loop Parallelization of JVM Bytecode

If the '-p' flag is given, `javab` examines the JVM bytecode of every method in each class file to automatically detect and exploit implicit parallelism in the natural loops that are defined by back edges in the flow graph (see e.g. [1, 2, 10, 21, 24, 25]). The default approach taken by `javab` to preserve the user-visible state after exceptions [16] as much as possible (without being over restrictive with respect to transformations) is to only transform regions of code for which the compiler can prove that **run-time exceptions** cannot occur, but to allow semantical changes with respect to the handling of **JVM errors** and **linking exceptions**. The compiler-switch '-l' can be used, however, to preserve the exact handling semantics of linking exceptions as well.

In some cases, the tool can detect implicit loop parallelism completely automatically. For example, below we show a small Java fragment together with the corresponding JVM bytecode (as presented by our research tool):

```
static double[] loop1(int n) {
  double[] a = new double[n];
  for (int i = 0; i < n; i++)
     a[i] = 1.0;
  return a;
}
```

```
*** method 0x08 loop1(I)[D 1->1
    max_stack   = 4
    max_locals  = 3
    code_length = 23
 ----------
 [ 0]  0:  iload_0
*[ 1]  1:  newarray double
 ----------
 [ 1]  3:  astore_1
 [ 0]  4:  iconst_0
 [ 1]  5:  istore_2
 [ 0]  6:  goto 16
 ----------
 [ 0]  9:  aload_1
 [ 1] 10:  iload_2
 [ 2] 11:  dconst_1
*[ 4] 12:  dastore
 ----------
 [ 0] 13:  iinc 2 1
 ----------
 [ 0] 16:  iload_2
 [ 1] 17:  iload_0
 [ 2] 18:  if_icmplt 9
 ----------
 [ 0] 21:  aload_1
 [ 1] 22:  areturn
 ----------
```

First, the access flags, name, and method descriptor [16, ch4] of the method are shown ('`0x08 loop1(I)[D`'). The next descriptor $w \rightarrow w'$ denotes the corresponding number of words moved from the operand stack of the caller into locals of the callee, and pushed on the operand stack of the caller by the callee on return, respectively. Finally, the bytecode is presented, where different basic blocks are separated by horizontal lines. For each instruction, the byte offset is shown together with the number of words that reside on the stack prior to execution of this instruction in between square brackets.

Each instruction that may throw a run-time or linking exception (ignoring JVM errors) is marked with an '*'. For this particular loop, javab can determine that the abnormal loop-exit from the instruction at address 12 will not be taken, because a non-null array reference is used, and the index is always within bounds of the declared array. Furthermore, javab can determine that all iterations of the bytecode loop corresponding to the i-loop are completely independent. Hence, the prototype proceeds with parallelization of this loop.

If the method loop1() appears in a class file MyClass.class that defines a class with the fully qualified name package.MyClass, then javab subsequently constructs a new class file MyClass_loop1_x.class that defines an auxiliary class package.MyClass_loop1_x (where '_x' denotes some compiler-generated number), and places this new class file in the same directory as where the original class file resided. In this manner, the auxiliary class belongs to the same package as the original class. The auxiliary class is a sub-class of 'java.lang.Thread', and contains a run()-method that can be used to let different threads executed the loop-body of the parallel loop concurrently. The number of threads T, where $2 \leq T \leq 16$, that must be used for the parallel execution can be defined by the environment variable 'JAVAB_THREADS' (the default value is $T = 4$). The original loop in MyClass.class is replaced by a construct that initiates these T threads. After all loops in the original class file have been examined, the (possibly) modified class is written back into the file MyClass.class, after a copy of the original program has been saved in a file MyClass.old.

In other cases, user interaction may be required. For example, below we show another small Java fragment together with the corresponding JVM bytecode:

```
static void loop2(double a[], int n) {
  for (int i = 0; i <= n+30; i++)
     a[i] = 1.0;
}
```

```
*** method 0x08 loop2([DI)V 2->0
    max_stack   = 4
    max_locals  = 3
    code_length = 21
 ----------
 [ 0]  0:  iconst_0
 [ 1]  1:  istore_2
 [ 0]  2:  goto 12
 ----------
 [ 0]  5:  aload_0
 [ 1]  6:  iload_2
 [ 2]  7:  dconst_1
*[ 4]  8:  dastore
 ----------
 [ 0]  9:  iinc 2 1
 ----------
 [ 0] 12:  iload_2
 [ 1] 13:  iload_1
 [ 2] 14:  bipush 30
 [ 3] 16:  iadd
 [ 2] 17:  if_icmple 5
 ----------
 [ 0] 20:  return
 ----------
```

Here, although javab can determine that the iterations of the bytecode loop corresponding to the i-loop are independent, without inter-procedural analysis the compiler cannot determine whether an abnormal loop-exit due to a run-time exception thrown by the dastore at address 8 may occur. In such cases, either worst-case assumptions are made by resorting to keeping the loop serial (the default), or, if the compiler-switch '-q' is given, user-interaction is initiated to obtain inter-procedural analysis:

```
*** MyClass.class:: Queries for method 'loop2([DI)V'
> Queries for the 1-dimensional array parameter '[D' in local 0
      * dim-1: is range [0, 31+n) valid where n = parameter in local 1?
      (y/n/q) =>
```

The following responses can be used:

- 'y': confirms the query
- 'n': denies the query (the loop will be kept serial)
- 'q': denies the query, and disables further user interaction

Furthermore, to prevent unforeseen loop-carried data dependences, for each array parameter that is involved in a `<t>astore` instruction within the candidate parallel loop, the compiler also inquiries the user whether this parameters will always refer to non-overlapping (sub) arrays in memory with respect to *all* other array parameters that are referred to within the loop.

Consider, for example, the following Java static method `query()`:

```
static void query(int[][] a, int[] b, int x[], int k, int l) {
  for (int i = 0; i <= k+100; i++)
     for (int j = 0; j < l-25; j++)
        b[i] += a[i][j] * x[j];
}
```

If user interaction is enabled with the flag '`-q`', then eventually the following inquiries will be made to the programmer:

```
*** ManQuery.class:: Queries for method 'query([[I[I[III)V'
  > Queries for the 2-dimensional array parameter '[[I' in local 0
    * dim-1: is range [0,101+n) valid where n = parameter in local 3?
    * dim-2: is range [0,-25+n) valid where n = parameter in local 4?
  > Queries for the 1-dimensional array parameter '[I' in local 2
    * dim-1: is range [0,-25+n) valid where n = parameter in local 4?
  > Queries for the 1-dimensional array parameter '[I' in local 1
    * dim-1: is range [0,101+n) valid where n = parameter in local 3?
Is storage of array parameters 0 '[[I' and 1 '[I' non-overlapping?
Is storage of array parameters 2 '[I'  and 1 '[I' non-overlapping?
```

We realize that resorting to user interaction as soon as inter-procedural information is required may place a heavy burden on the user, especially if source code is not available. However, these problems will be alleviated by the incorporation of inter-procedural analysis in environments where complete bytecode programs are available.

4 Concluding Remarks

For a more detailed presentation of the techniques used by the prototype bytecode parallelization tool `javab`, we refer to [5]. Please keep in mind that `javab` is a prototype tool, and not a robust commercial product. We hope, however, that this bytecode parallelization tool provides an environment with sufficient functionality to enable rapid prototyping of new automatic JVM bytecode parallelization methods. We encourage users to modify or extend `javab` for further experimentation with bytecode parallelization (please refer to the `LICENSE` file provided with `javab` for details on the licensing terms). With the '`-v`' flag, `javab` provides some information on why particular loops cannot be parallelized. We hope that using this option may gain insight in the kind of improvements required to increase the number of loops in JVM bytecodes that can be parallelized automatically.

New releases of `javab` (and the Java restructuring compiler `javar` [4, 6]) are made available at our HP-Java page (`http://www.extreme.indiana.edu/hpjava/`). Please send us all your comments, bug reports, experiences, and suggestions (`ajcbik@extreme.indiana.edu`).

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, New York, 1997.

[3] Utpal Banerjee. *Dependence Analysis*. Kluwer, Boston, 1997. A Book Series on Loop Transformations for Restructuring Compilers.

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

[5] Aart J.C. Bik and Dennis B. Gannon. `javab` – a prototype bytecode parallelization tool. Technical Report 489, Computer Science Department, Indiana University, 1997. This report and the complete source of `javab` are made available at `http://www.extreme.indiana.edu/hpjava/`.

[6] Aart J.C. Bik and Dennis B. Gannon. `javar` – a prototype java restructuring compiler. Technical Report 487, Computer Science Department, Indiana University, 1997. This manual and the complete source of `javar` are made available at `http://www.extreme.indiana.edu/hpjava/`.

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

[8] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with 'HP Java'. *Concurrency, Practice and Experience*, 9(6):633–648, 1997.

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

[10] C.N. Fischer and R.J. LeBlanc. *Crafting a Compiler*. Benjamin-Cummings, Menlo Park, California, 1988.

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

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

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

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

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

[16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[17] Jon Meyer. *Jasmin: a Java Assembler Interface*. This JVM bytecode assembler is made available at `http://www.cat.nyu.edu/meyer/jasmin/`.

[18] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly & Associates, Sebastopol, CA, 1997.

[19] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36:1485–1495, 1987.

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

[21] Thomas W. Parsons. *Introduction to Compiler Construction*. Computer Science Press, New York, 1992.

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

[23] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 1992.

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

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