Contents

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

`javab` – a prototype bytecode parallelization tool

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

Abstract

This paper presents techniques for the automatic detection and automatic exploitation of implicit loop parallelism in bytecode, i.e. the architectural neutral instructions of the JVM. 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. Some initial experiments with a prototype bytecode parallelization tool (that is made freely available) have been included.

1 Introduction

Architectural neutrality of the Java programming language [18] is due to the compilation of Java source programs into bytecode, i.e. instructions of the JVM (Java Virtual Machine) [26], rather than the more traditional compilation of source programs into native machine code. Bytecode can run on any platform that provides an implementation of the JVM. Below, a typical compiled-interpreted execution of a Java program is illustrated. First, a Java compiler (`javac` in the diagram) compiles a Java source program '`MyClass.java`' into bytecode that is embedded in a class file '`MyClass.class`'. Subsequently, this bytecode is interpreted by an implementation of the JVM (`java` in the diagram, or, alternatively, an interpreter that is embedded in a browser):

	(i)		(ii)		(iii)		Interpretation
MyClass.java	$\Longrightarrow$	$\boxed{\texttt{javac}}$	$\Longrightarrow$	MyClass.class	$\Longrightarrow$	$\boxed{\texttt{java}}$	of bytecode

Although the interpretation of bytecode is substantially faster than the interpretation of most high level languages, still a performance penalty must be paid for architectural neutrality. Hence, although Java already has a potential to become a major language in scientific and engineering computing [9, 17], clearly means to speedup execution have to be found before Java can be truly competitive with languages like FORTRAN and C.

The diagram above reveals some obvious ways to speedup Java programs. At source code level (viz. (i)), a source-to-source restructuring can be applied. In earlier work [6], for example, we have shown how a restructuring compiler can make loop parallelism in a Java source program explicit using the multi-threading mechanism of the Java programming language [18, 24, 32]. Eventually, speedup can be obtained on any platform that supports true parallel execution of JVM threads (like the AIX4.2 JDK on an IBM RS/6000 G30). However, other source program transformations that eventually result in the generation of more efficient bytecode can also be applied by a restructuring compiler.

Alternatively, optimizations can be applied at bytecode level, either at compile-time or run-time. Compile-time bytecode optimization (viz. (ii)) can be done by an integrated module of the Java compiler (cf. '`javac -O`') or by a stand-alone bytecode-to-bytecode optimizer [7, 12].

The latter approach makes the optimizer independent of the actual Java compiler, which allows the optimization of bytecodes from alternative sources, such as 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 [28, 29]. The former approach has the advantage, however, that high level information about the program does not have to be recovered prior to the actual optimizations.

Finally, bytecode optimizations can be performed at run-time (viz. (iii)). Again, a bytecode-to-bytecode transformation can be performed at this stage, which now has the advantage that specific properties of the target platform can be accounted for during the optimizations [11]. A more aggressive approach at this stage, however, is to abandon the interpretation of bytecode completely, and to perform a JIT (just-in-time) compilation into native machine code prior to execution (see e.g. [10, 20, 22]). Besides the obvious advantage that native code runs substantially faster than bytecode interpretation, the run-time approach usually also allows more optimizations. For example, even if a Java compiler can proof that particular array references cannot violate subscript bounds, this information cannot be expressed in the bytecode.[1] Hence, without further analysis, the JVM specification requires that run-time checks are performed for each individual reference. Run-time optimization, however, can generate native code (or, in the interpreted approach, a new form of quick pseudo-instruction could be used, cf. [26, ch9]) in which all superfluous checks are omitted.

In this paper, we focus on one particular bytecode-to-bytecode optimization that can be performed at either (ii) or (iii), namely the *automatic detection* of implicit loop parallelism in bytecode and the *automatic exploitation* of this loop parallelism by means of the multi-threading mechanism provided by the JVM. Exploiting parallelism at bytecode level has the advantage that 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. The parallelized bytecode remains architectural neutral and may exhibit speedup on any platform that supports the true parallel execution of JVM threads. The techniques presented in this paper have been actually implemented in a prototype bytecode parallelization tool `javab`, which is made freely available for education, research, and non-profit purposes (for details, see the end of this paper). To keep compile-time limited, the prototype relies on less accurate but generally also less expensive analysis. 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

The rest of this paper is organized as follows. First, in Section 2, we discuss analysis of bytecode. Subsequently, the automatic detection and exploitation of implicit loop parallelism in bytecode is discussed in Section 3. In Section 4, the results of some initial experiments are given. Finally, in Section 5, conclusions are stated.

2 Bytecode Analysis

Bytecode is analyzed by first performing **control flow analysis** to determine the control structure of a program, followed by **data flow analysis** to determine the flow of data through this program. Although control and data flow analysis of more traditional intermediate representations (such as three-address code) are described extensively in the compiler literature (see e.g. [1, 3, 15, 35, 46, 47]), we briefly discuss the analysis specifically for bytecode.

[1]Although the class file format provides a way to supply information about bytecode in so-called attributes [26, ch4], there is no pre-defined way yet to support bytecode optimization. Also, for security reasons, JVM implementations will probably have to reject such optimization attributes from untrusted sources anyway.

2.1 The Java Virtual Machine

The JVM is a stack-based virtual machine that has been designed to support the Java programming language [18, 26]. The input of the JVM consists of platform-independent class files. Each class file is a binary file that contains information about the fields and methods of one particular class, a constant pool (a kind of symbol-table), as well as the actual bytecode for each method.

Each JVM instruction consists of a one-byte **opcode** that defines a particular operation, followed by zero or more byte **operands** that define the data for the operation. For example, instruction '`sipush 500`' (push constant 500 on the stack) is represented by the bytes 17, 1, and 244:

17	opcode for `sipush`
1	these two operands together define the `short` literal `500`
244	(in big-endian order), which is sign-extended to an `int` literal

For most JVM opcodes, the number of corresponding operands is fixed, whereas for the other opcodes (`lookupswitch`, `tableswitch`, and `wide`) this number can be easily determined from the bytecode context. Consequently, once the offset into a class file and the length for the bytecode of a certain method have been determined, it is straightforward to parse the bytecode instructions of this method.

At run-time, the JVM fetches an opcode and corresponding operands, executes the corresponding action, and then continues with the next instruction. At the JVM-level, operations are performed on the abstract notion of **words** [26, ch3]: words have a platform-specific size, but two words can contain values of type `long` and `double`, whereas one word can contain values of all other types. During execution of bytecode, three exceptional situations may arise:

(1) The JVM throws an instance of a subclass of `VirtualMachineError` in case an internal error or resource limitation prevents further execution.

(2) An exception is thrown *explicitly* by the instruction `athrow`.

(3) An exception is thrown *implicitly* by a JVM instruction.

Situation (1) can occur during execution of *any* instruction, because the JVM specification does not mandate precisely when **JVM errors** can be reported [26, ch6]. In contrast, the compiler can easily determine when situation (2) or (3) may occur, because the JVM specification [26] precisely documents which instructions may explicitly or implicitly throw **linking exceptions** or **run-time exceptions**. Our approach to preserve the user-visible state after exceptions 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* (a compiler-switch is provided, however, to preserve the exact handling semantics of linking exceptions as well).

2.2 Flow Graphs

A region of straight-line bytecode that can only be entered via the first instruction, and only be exited via the last instruction is referred to as a **basic block**. We can partition the bytecode for one method into basic-blocks by first finding the set of leaders (cf. [1, ch9]):

- The first instruction of the method and each first instruction of every exception handler for the method are leaders.
- Each statement that is the target of a conditional or unconditional branch (including `jsr` and `jsr_w`) or one of the targets of a `<kind>switch` is a leader.
- Each statement that immediately follows a conditional or unconditional branch (including `jsr` and `jsr_w`), a `ret`, `<T>return`, or `<kind>switch` instruction, or that immediately follows an instruction that may throw an exception (ignoring JVM errors) is a leader.

Now, each individual leader gives rise to a basic block, consisting of all instructions up to the next leader or the end of the bytecode. Furthermore, we enclose each method invocation (`invoke<kind>`) in a basic block of its own.

The **flow graph** $G = \langle V, E, v_0 \rangle$ of a method consists of a directed graph with a set of vertices V that represents the basic blocks, a set of edges $E \subseteq V \times V$ that represents transfer of control between these basic blocks, and a vertex $v_0 \in V$ that represents the entry of the method. There is a directed edge $(v, w) \in E$ if execution of the instructions in w can immediately follow execution of the instructions in v.

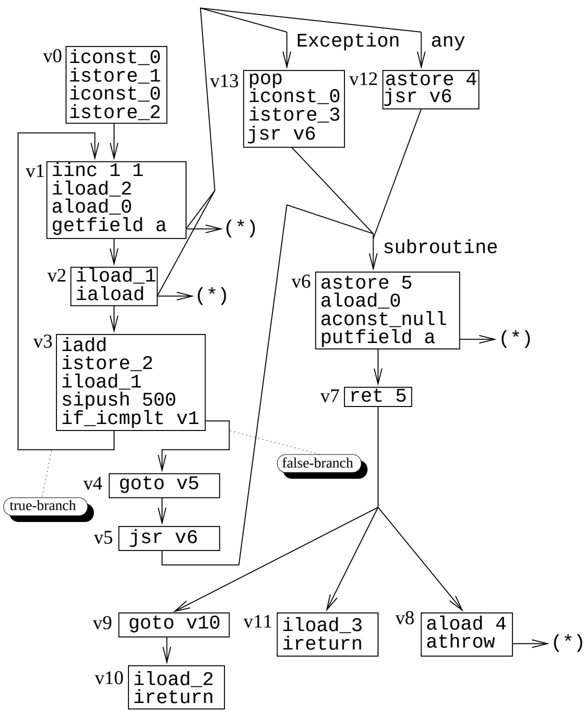
Hence, we add an edge (v, w) to E, if w follows v in the bytecode and v does not end in a unconditional branch, or if the last instruction of v is a conditional or unconditional branch to the first instruction in w. In addition, we add an edge from the basic block of each `tableswitch` or `lookupswitch` to the basic block of each instruction that is defined as target in the switch. For each subroutine call, we add two edges to E: one from the basic block of the `jsr/jsr_w` to the basic block of target, and one from the basic block containing the corresponding `ret` to the basic block of the instruction that immediately follows the call. Finally, we add an edge from the basic block of each instruction that may throw an exception to the entry basic block of every exception handler that covers the region of bytecode in which the instruction appear, and to a dummy basic block that represents **abnormal completion** of the method [26, ch3], i.e. the situation where a thrown exception is not caught by any handler of the method (in the prototype, the types of exceptions are not accounted for during flow graph construction, so that some redundant edges may arise). After the complete flow graph has been constructed, any vertex that cannot be reached from the entry vertex $v_0 \in V$ is discarded from the flow graph.

Consider, as a first example, the following Java source code that provides an implementation of a class `Flow`. The bytecode for this class (as presented by our research tool) together with the corresponding flow graph $G = \langle V, E, v_0 \rangle$, where $V = \{v_0, \ldots, v_{13}\}$, and local variables usage are shown in Figure 1.

```
class Flow {
   int a[];
   int flow() {
     int i = 0, j = 0;
     try {
       do { i++;
            j += a[i];
       } while (i < 500);
     }
     catch(Exception e) { return 0; }
     finally            { a = null; }
     return j;
   }
}
```

First, the access flags, name, and method descriptor [26, ch4] of the method are shown (`'0x00 flow()I'`). 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 (note that reference `this` is implicitly passed to every instance method), and pushed on the operand stack of the caller by the callee on return, respectively. The table of exception handlers is shown next. Eventually, 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 (in Section 2.6, we explain how this stack usage can be computed). Each instruction that may throw an exception (ignoring JVM errors) is marked with an '*'.

Entry $v_0 \in V$ consist of the first four instructions, because the `iinc` at address 4 is used as a target of the conditional branch at address 20. The next basic block $v_1 \in V$ ends at the `getfield` at address 9, because this instruction may implicitly throw an exception. The `pop` at address 26 is a leader because it appears after a conditional branch, and because this instruction is the target of the first exception handler. The other basic blocks are constructed similarly.

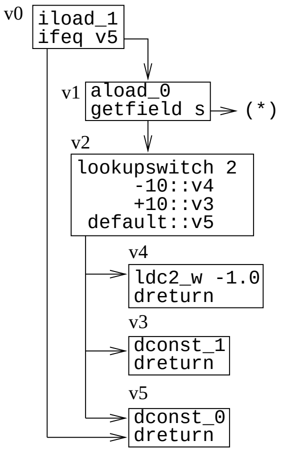



```
*** method 0x00 flow()I 1->1
[4,23) -> 26 java.lang.Exception
[4,34) -> 40 any
----------
[ 0]  0:  iconst_0
[ 1]  1:  istore_1
[ 0]  2:  iconst_0
[ 1]  3:  istore_2
----------
[ 0]  4:  iinc 1 1
[ 0]  7:  iload_2
[ 1]  8:  aload_0
*[ 2]  9:  getfield Flow.a [I
----------
[ 2] 12:  iload_1
*[ 3] 13:  iaload
----------
[ 2] 14:  iadd
[ 1] 15:  istore_2
[ 0] 16:  iload_1
[ 1] 17:  sipush 500
[ 2] 20:  if_icmplt 4
----------
[ 0] 23:  goto 34
----------
[ 1] 26:  pop
[ 0] 27:  iconst_0
[ 1] 28:  istore_3
[ 0] 29:  jsr 48
----------
[ 0] 32:  iload_3
[ 1] 33:  ireturn
----------
[ 0] 34:  jsr 48
----------
[ 0] 37:  goto 57
----------
[ 1] 40:  astore 4
[ 0] 42:  jsr 48
----------
[ 0] 45:  aload 4
*[ 1] 47:  athrow
----------
[ 1] 48:  astore 5
[ 0] 50:  aload_0
[ 1] 51:  aconst_null
*[ 2] 52:  putfield Flow.a [I
----------
[ 0] 55:  ret 5
----------
[ 0] 57:  iload_2
[ 1] 58:  ireturn
----------
```

Locals

	Locals
0:	this
1:	i
2:	j
3:	0
4:	any
5:	return

Figure 1: Flow graph and bytecode of instance method `flow()`




```
*** method 0x00 switcher(Z)D 2->2
----------
[ 0]  0:  iload_1
[ 1]  1:  ifeq 42
----------
[ 0]  4:  aload_0
*[ 1]  5:  getfield Switcher.s I
----------
[ 1]  8:  lookupswitch 2
        -10::36
        +10::40
     default::42
----------
[ 0] 36:  ldc2_w -1.0
[ 2] 39:  dreturn
----------
[ 0] 40:  dconst_1
[ 2] 41:  dreturn
----------
[ 0] 42:  dconst_0
[ 2] 43:  dreturn
----------
```

	Locals
0:	this
1:	b

Figure 2: Flow graph and bytecode of instance method `switcher()`

Both the edges (v_3, v_1) and (v_3, v_4) are added to E, because of the conditional branch at address 20. The edges (v_5, v_6), (v_{12}, v_6), and (v_{13}, v_6), together with the edges (v_7, v_8), (v_7, v_9), and (v_7, v_{11}), are due to the three subroutine calls in this fragment. Vertex v_2 is connected to v_3 because of the normal flow of execution. Moreover, because the `iaload` at address 13 may implicitly throw an exception, v_2 is also connected to both the exception handlers v_{12} and v_{13} that covers address 13, as well as to a dummy vertex representing abnormal completion of the method, which is indicated as an edge to an '`(*)`' in the figure (note that the latter two edges are redundant in this case). Vertex v_8, on the other hand, is only connected to the vertex of abnormal completion. The rest of the control flow graph construction is straightforward.

Consider, as another example, the Java source code shown below. The bytecode, the corresponding flow graph $G = \langle V, E, v_0 \rangle$, where $V = \{v_0, \ldots, v_5\}$, and the local variables usage are shown in Figure 2. Here, the edges (v_2, v_3), (v_2, v_4), and (v_2, v_5) are included in E because of the `lookupswitch` instruction at address 8 that is used to implement the Java `switch` statement.

```
class Switcher {
  int s;
  double switcher(boolean b) {
    if (b)
      switch(s) {
        case -10: return -1.0;
        case +10: return +1.0;
      }
    return 0.0;
  }
}
```

Analysis that is only concerned with an individual basic block is referred to as **local analysis**, whereas analysis of all the basic blocks belonging to one method is usually called **global analysis** (or intra-procedural analysis). In addition, analysis that deals with a program as a whole is called **inter-procedural analysis** (where a call-graph [40] is used to represent possible transfer of control between methods). To enable transformations on a single class file (rather than only on a complete bytecode program which is not always available beforehand), in this paper we will focus on local and global analysis, and resort to worst-case assumptions (the default) or user interaction (which can be enabled with a compiler-switch) if inter-procedural information is required.

```
procedure comp_dominators() {
  Dom(v0) := {v0};
  foreach v ∈ V − {v0}
    Dom(v) := V;

  do {
    busy := false;
    foreach v ∈ V − {v0} {

      O := {v} ∪   ⋂        Dom(p);
                p∈Pred(v)

      if (Dom(v) ≠ O) {
        busy := true;
        Dom(v) := O;
      }
    }
  } while (busy);
}
```

```
procedure comp_natloop(v) {
  if v ∉ L {
    L := L ∪ {v};
    foreach p ∈ Pred(v)
      make_natloop(p);
  }
}
```

Figure 3: Computing dominator sets and natural loops

2.3 Dominators and Natural Loops

For a flow graph $G = \langle V, E, v_0 \rangle$ and vertices $w \in V$ and $v \in V$, we say that w **dominates** v if every directed path from v_0 to v contains vertex w. In Figure 3, where $\text{Pred}(v) = \{p \in V \mid (p, v) \in E\}$, we give an iterative algorithm `comp_dominators()` to compute the set of dominators $\text{Dom}(v) \subseteq V$ for every vertex $v \in V$ [1, p670–671].

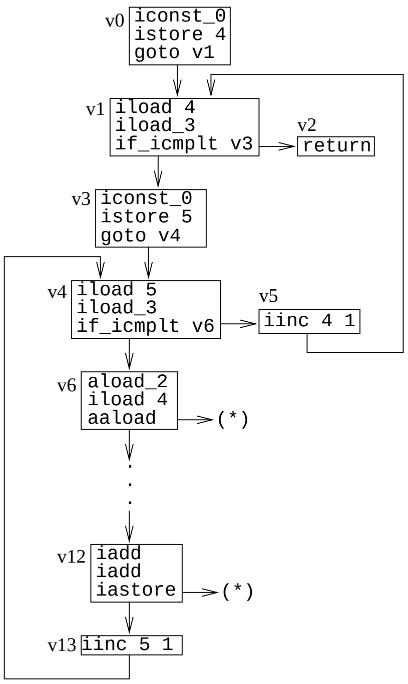
An edge $(g, h) \in E$ where $h \in \text{Dom}(g)$ is called a **back edge**. Such a back edge defines a **natural loop** $L \subseteq V$, consisting of all vertices that can reach vertex g without going through h. Given a back edge $(g, h) \in E$, the corresponding natural loop can be constructed by invoking the algorithm in Figure 3 as `comp_natloop`(g) for an initial set $L = \{h\}$ [1, p604]. Given a natural loop $L \subseteq V$, an edge $(v, e) \in E$ where $v \in L$ and $e \notin L$ is called a **loop-exit**. We further distinguish between **normal loop-exits** and **abnormal loop-exits**, depending on whether the edge can be taken during normal program execution or only directly after an instruction throws an exception (leading to either an exception handler or abnormal completion of the method).

In the flow graph $G = \langle V, E, v_0 \rangle$ of Figure 1, for instance, we have $\text{Dom}(v_3) = \{v_0, v_1, v_2, v_3\}$, and $(v_3, v_1) \in E$ forms a back edge that defines the natural loop $L = \{v_1, v_2, v_3\}$. For this loop, $(v_3, v_4) \in E$ forms a normal loop-exit, and the edges from vertices v_1 and v_2 to the vertices v_{12}, v_{13}, and the dummy vertex representing abnormal completion of method `flow()` form abnormal loop-exits.

Consider, as another example, the following Java instance method:

```java
static void mat(int[][] a, int[][] b, int[][] c, int n) {
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      c[i][j] += a[i][j] + b[i][j];
}
```

In Figure 4, the bytecode for this method and corresponding flow graph $G = \langle V, E, v_0 \rangle$ are shown. Here we have, for example, $\text{Dom}(v_5) = \{v_0, v_1, v_3, v_4, v_5\}$. There are two back edges, namely $(v_5, v_1) \in E$ and $(v_{13}, v_4) \in E$, with corresponding natural loops $L = \{v_1, v_3, \ldots, v_{13}\}$ and $L' = \{v_4, v_6, \ldots, v_{13}\}$, respectively. For the former loop, edge $(v_1, v_2) \in E$ forms a normal loop-exit, whereas all the edges to the dummy vertex that represents abnormal completion of the method form abnormal loop-exits. For the latter loop, edge $(v_4, v_5) \in E$ forms a normal loop-exit, whereas, again, all edges representing abnormal completion of `mat()` form abnormal loop-exits.




```
*** method 0x08 mat([[I[[I[[II)V 4->0
----------
[ 0]  0:  iconst_0
[ 1]  1:  istore 4
[ 0]  3:  goto 49
----------
[ 0]  6:  iconst_0
[ 1]  7:  istore 5
[ 0]  9:  goto 40
----------
[ 0] 12:  aload_2
[ 1] 13:  iload 4
*[ 2] 15:  aaload
----------

  ...

----------
[ 5] 34:  iadd
[ 4] 35:  iadd
*[ 3] 36:  iastore
----------
[ 0] 37:  iinc 5 1
----------
[ 0] 40:  iload 5
[ 1] 42:  iload_3
[ 2] 43:  if_icmplt 12
----------
[ 0] 46:  iinc 4 1
----------
[ 0] 49:  iload 4
[ 1] 51:  iload_3
[ 2] 52:  if_icmplt 6
----------
[ 0] 55:  return
----------
```

	Locals
0:	a
1:	b
2:	c
3:	n
4:	i
5:	j

Figure 4: Flow graph and bytecode of class method `mat()`

2.4 Reaching Definitions of Local Variables

On a method invocation, a fixed-sized frame consisting of a fixed-sized operand stack and a set of local variables is allocated [26, ch3]. Effectively, this latter set consists of an array of words in which locals are addressed as word offsets from the array base.

A bytecode instruction that assigns a value to a local variable in this frame forms a **definition** of that local. Hence, the instructions `istore_<l>`, `istore`, `iinc`, `fstore_<l>`, `fstore`, `astore_<l>`, and `astore` form definitions of the local that is defined either implicitly in the opcode or explicitly in the next operand byte (or two bytes if combined with a `wide` instruction).

Similarly, because the instructions `dstore_<l>`, `dstore`, `lstore_<l>`, and `lstore` effectively operate on two locals (viz. a data item of type `long` or `double` at `l` effectively occupies locals `l` and `l+1`), we let each such instruction form *two* definitions of locals. For example, '`iinc 5 1`' forms a definition of local 5, while `dstore_0` forms definitions of both locals 0 and 1.

The parameter passing mechanism of bytecode gives rise to another source of definitions of local variables: if w words of parameters are passed to a particular method, then invoking that method forms initial definitions of the first w locals. The types of these parameters can be easily determined from the bytecode context. For an instance method, the first parameter is a reference `this` to an instance of the class in which the method is defined. Types of all other arguments are defined by the corresponding method descriptor [26, ch4].

At implementation level, each definition of a local in a particular method can be represented by two words: the address of the defining instruction (set to $\perp$ for definitions arising from parameter passing), and the offset of the local. We say that such a definition **reaches** a particular bytecode instruction, if there is an execution path from the definition to the instruction along which there are no definitions of the same local. The compiler can determine global reaching definitions information

```
procedure comp_reaching_defs() {
  foreach v ∈ V
    rd_out[v] := rd_gen[v];

  do {
    busy := false;
    foreach v ∈ V {

      rd_in[v] :=      ⋃        rd_out[p];
                  p∈Pred(v)

      O := rd_gen[v] ∪ (rd_in[v] − rd_kill[v]);

      if (rd_out[v] ≠ O) {
        busy := true;
        rd_out[v] := O;
      }
    }
  } while (busy);
}
```

Figure 5: Computing reaching definitions of local variables

for a method with control flow graph $G = \langle V, E, v_0 \rangle$ by solving a forward, any-path (may) data-flow problem [1, 3, 15, 35, 46, 47]. First, for each basic block $v \in V$, the sets rd_gen[v] and rd_kill[v] are constructed. The former set consists of all definitions in v that also reach the end of this basic block (i.e. if a basic block contains several definitions of the *same* local variable, then only the last definition appears in the gen[v]). For each local defined in v, the rd_kill[v] contains all definitions of this local in the method other than the one appearing in rd_gen[v].

Thereafter, global reaching definitions information can be found by solving the following data flow equations for sets rd_in[v] and rd_out[v] over all $v \in V$:

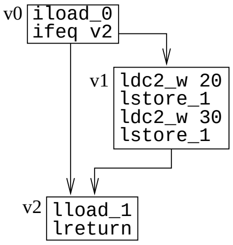
$$\begin{cases} \text{rd\_in}[v] &= \displaystyle\bigcup_{p \in \text{Pred}(v)} \text{rd\_out}[p] \\ \\ \text{rd\_out}[v] &= \text{rd\_gen}[v] \cup (\text{rd\_in}[v] - \text{rd\_kill}[v]) \end{cases} \tag{1}$$

The iterative algorithm `comp_reaching_defs()` shown in Figure 5 is used to solve these data flow equations (statistics gathered in [21] reveal that visiting the vertices in the flow graph according to topological sorting of the dominance relation tends to reduce the number of passes required to solve forward data flow equations iteratively [1, 47]). Here, we assume that the entry vertex $v_0 \in V$ has a dummy predecessor w, where rd_out[w] consists of all definitions that are formed by invoking the current method.

Some subtleties of computing global reaching definitions information for locals are illustrated using the bytecode for the following Java class method `retLong()`:

```
static long retLong(boolean b, long l) {
  if (b) {
    l = 20;
    l = 30;
  }
 return l;
}
```

The bytecode and flow graph $G = \langle V, E, v_0 \rangle$ are shown in Figure 6. Method descriptor '`(ZJ)J`' indicates that a parameter of type `boolean` and a parameter of type `long` are passed to this method. Since a `long` takes two words (shown as `l-1` and `l-2` in the illustration of the local variables usage), invoking the method forms definitions of the locals 0, 1, and 3. The instructions `lstore_1` at address 7 and 11 form two definitions of the locals 1 and 2, respectively:




```
*** method 0x08 retLong(ZJ)J 3->2
----------
[ 0]   0:  iload_0
[ 1]   1:  ifeq 12
----------
[ 0]   4:  ldc2_w 20
[ 2]   7:  lstore_1
[ 0]   8:  ldc2_w 30
[ 2]  11:  lstore_1
----------
[ 0]  12:  lload_1
[ 2]  13:  lreturn
----------
```

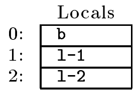


Figure 6: Flow graph and bytecode of class method `retLong()`

	d_1	d_2	d_3	d_4	d_5	d_6	d_7
local	0	1	2	1	2	1	2
address	$\perp$	$\perp$	$\perp$	7	7	11	11
par. type	Z	J	–				

The rd_gen[v] and rd_kill[v] set for each $v \in V$, where $V = \{v_0, v_1, v_2\}$, are derived from examination of the bytecode. Note that because the instruction at address 11 kills the (useless) definition that is formed by the instruction at 7, only d_6 and d_7 appear in the rd_gen[v_1].[2] Eventually, the following sets are computed:

v	rd_gen[v]	rd_kill[v]	rd_in[v]	rd_out[v]
v_0	$\emptyset$	$\emptyset$	$\{d_1, d_2, d_3\}$	$\{d_1, d_2, d_3\}$
v_1	$\{d_6, d_7\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_1, d_2, d_3\}$	$\{d_1, d_6, d_7\}$
v_2	$\emptyset$	$\emptyset$	$\{d_1, d_2, d_3, d_6, d_7\}$	$\{d_1, d_2, d_3, d_6, d_7\}$

Consider, as another small example, the following Java class method `reach()` in which a `do-while`-loop and three local variables `i`, `j`, and `k` are used (cf. [1, p619]):

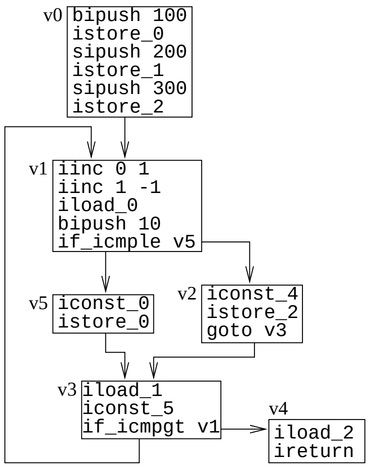
```
static int reach() {
  int i = 100, j = 200, k = 300;
  do { i++; j--;
       if (i > 10) k = 4; else i = 0;
  } while (j > 5);
  return k;
}
```

The bytecode and corresponding flow graph $G = \langle V, E, v_0 \rangle$, where $V = \{v_0, \ldots, v_5\}$, for this method are shown in Figure 7. Because the method descriptor '`()I`' of this class method indicates that no parameters are passed to `reach()`, all definitions in this fragment are formed by bytecode instructions:

	d_1	d_2	d_3	d_4	d_5	d_6	d_7
local	0	1	2	0	1	2	0
address	2	6	10	11	14	24	29

The rd_gen[v] and rd_kill[v] set for $v \in V$ are derived from examination of the bytecode. Subsequently, given the flow graph of Figure 7, the rd_in[v] and rd_out[v] sets are computed by the iterative algorithm `comp_reaching_defs()` of Figure 5:

[2]Consequently, although according to our definition, d_4 and d_5 belong to the rd_kill[v_1], we could equally well omit these definitions from this set because d_4 and d_5 cannot reach any other basic block.




```
*** method 0x08 reach()I 0->1
----------
[ 0]  0:  bipush 100
[ 1]  2:  istore_0
[ 0]  3:  sipush 200
[ 1]  6:  istore_1
[ 0]  7:  sipush 300
[ 1] 10:  istore_2
----------
[ 0] 11:  iinc 0  1
[ 0] 14:  iinc 1 -1
[ 0] 17:  iload_0
[ 1] 18:  bipush 10
[ 2] 20:  if_icmple 28
----------
[ 0] 23:  iconst_4
[ 1] 24:  istore_2
[ 0] 25:  goto 30
----------
[ 0] 28:  iconst_0
[ 1] 29:  istore_0
----------
[ 0] 30:  iload_1
[ 1] 31:  iconst_5
[ 2] 32:  if_icmpgt 11
----------
[ 0] 35:  iload_2
[ 1] 36:  ireturn
----------
```

Locals

0:	i
1:	j
2:	k

Figure 7: Flow graph and bytecode of class method `reach()`

v	rd_gen[v]	rd_kill[v]	rd_in[v]	rd_out[v]
v_0	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6, d_7\}$	$\emptyset$	$\{d_1, d_2, d_3\}$
v_1	$\{d_4, d_5\}$	$\{d_1, d_2, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_3, d_4, d_5, d_6\}$
v_2	$\{d_6\}$	$\{d_3\}$	$\{d_3, d_4, d_5, d_6\}$	$\{d_4, d_5, d_6\}$
v_3	$\emptyset$	$\emptyset$	$\{d_3, d_4, d_5, d_6, d_7\}$	$\{d_3, d_4, d_5, d_6, d_7\}$
v_4	$\emptyset$	$\emptyset$	$\{d_3, d_4, d_5, d_6, d_7\}$	$\{d_3, d_4, d_5, d_6, d_7\}$
v_5	$\{d_7\}$	$\{d_1, d_4\}$	$\{d_3, d_4, d_5, d_6\}$	$\{d_3, d_5, d_6, d_7\}$

2.5 UD-Chains and DU-Chains of Local Variables

A bytecode instruction that reads the value of a local variable forms a **use** of that local. Hence, the instructions `iload_<l>`, `iload`, `iinc`, `fload_<l>`, `fload`, `aload_<l>`, and `aload` form uses of a single local defined either implicitly in the opcode or explicitly in the next operand byte (or two bytes if combined with a `wide` instruction). Similarly, instructions `dload_<l>`, `dload`, `lload_<l>`, and `lload` effectively form uses of two locals. At implementation level, each use in a particular method may be represented by two words: the address of the using instruction and the offset of the used local.

In a **ud-chain** (use/definition-chain), each use u has an associated set UD(u) consisting of all definitions that reach that use. Given the global reaching definitions information of the previous section, it is straightforward to construct local reaching definitions information (i.e. for each individual instruction) and, hence, ud-chains [1, ch10]. Starting with the initial set rd_in[v] of a basic block $v \in V$, we step through the instructions of this basic block in order. At each instruction i, we first associate definitions in the current set with all corresponding uses formed by i. Subsequently, we delete all definitions that are killed by i, add all definitions generated by i, and continue with the next instruction.

Likewise, in a **du-chain** (definition/use-chain), each definition d has an associated set DU(d) consisting of all uses that are reached by this particular definition. Although the problem of finding du-chains can be formulated as another data flow problem (see e.g. [1, p632-633]), we can also use the following obvious property to construct du-chains from ud-chains:

$$u \in \mathrm{DU}(d) \Leftrightarrow d \in \mathrm{UD}(u) \tag{2}$$

If $\mathrm{DU}(d) = \emptyset$, then definition d does not reach any use, and the corresponding instruction in the bytecode can be eliminated as **dead code** [1, ch9,10]. Likewise, if $\mathrm{UD}(u) = \emptyset$, an uninitialized variable is referenced, which indicates a programming error in the bytecode.

For example, in method `retLong()` of the previous section, there are effectively three uses:

	u_1	u_2	u_3
local	0	1	2
address	0	12	12

Since there are no killing definitions prior to these uses in the same basic block, the ud-chains can be directly constructed from the set rd_in$[v]$ computed in the previous section:

u	u_1	u_2	u_3
$\mathrm{UD}(u)$	$\{d_1\}$	$\{d_2, d_6\}$	$\{d_3, d_7\}$

Subsequently, property (2) can be used to construct the du-chain for this method:

d	d_1	d_2	d_3	d_4	d_5	d_6	d_7
$\mathrm{DU}(d)$	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\emptyset$	$\emptyset$	$\{u_2\}$	$\{u_3\}$

Because the JVM specifications does not allow operations on the individual words of `long` or `double` data items, definition- and use-pairs of such locals (like $u_2 - u_3$ and $d_2 - d_3$) are always associated with each other in a pair-wise fashion.

2.6 Stack Sizes

As stated earlier, the frame that is allocated on a method invocation also consists of a fixed-size operand stack. Using this stack, however, slightly differs from using stacks in more traditional stack-based architectures: the JVM specification requires that for each bytecode instruction, the operand stack contains the same number of words prior to the execution of this instruction, regardless of the execution path taken to reach the instruction [26, ch4].

Consider, for example, the following Java method:

```
static int myMethod(int n) {
  int i, acc = 0;
  for (i = 1; i <= n; i++) acc += i;
  return acc;
}
```

The left fragment shown below presents an assembler representation of the bytecode that is generated by the Java compiler for this method. Here we see that the stack contains the same number of words prior to executing each individual bytecode instruction. The right fragment shown below presents an alternative hand-written bytecode assembler implementation for this method. Here, first all values of `i` are pushed on the stack in one loop. Subsequently, these values are added using a second loop. Although this implementation could be used on a traditional stack-based architecture, this fragment is invalid for the JVM because the number of words residing on the stack prior to executing the instruction at label `Test1`, for example, is not a fixed constant:

```
procedure comp_sp(v, sp) {
  foreach i ∈ Instr(v) in order {
    if (i.sp = ⊥) { // first visit

      i.sp := sp;          // record for instruction

      pre := sp_pre(i);
      pos := sp_pos(i);

      if (sp < pre)
        report("stack underflow");

      sp += (pos - pre);   // update stack pointer

      if (sp > max_stack)
        report("stack overflow");
    }
    else {  // subsequent visit
      if (i.sp != sp)
        report("inconsistent stack size");
      else
        return;
    }
  }

  foreach s ∈ Succ(v)
    if (s is an exception handler for v) then
      comp_sp(s, 1);  // single exception reference on stack
    else
      comp_sp(s, sp);
}
```

Figure 8: Computing stack usage

javac-generated bytecode:

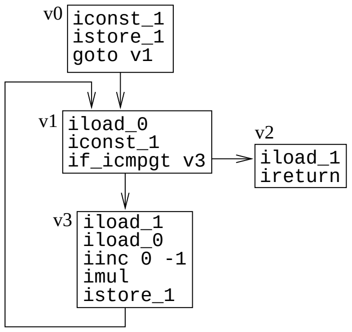
```
      iconst_0
      istore_2       ; acc = 0
      iconst_1
      istore_1       ; i = 1
      goto Test
Loop: iload_2
      iload_1
      iadd           ; acc += i
      istore_2
      iinc 1 1       ; i++
Test: iload_1
      iload_0
      if_icmple Loop ; if (i <= n) goto Loop
      iload_2
      ireturn        ; return acc
```

hand-written bytecode (invalid for JVM):

```
       iconst_0        ; push 0
       iconst_1
       istore_1        ; i = 1
       goto Test1
Loop1: iload_1         ; push i
       iinc 1 1        ; i++
Test1: iload_1
       iload_0
       if_icmple Loop1 ; if (i <= n) goto Loop1
       iconst_1
       istore_1        ; i = 1
       goto Test2
Loop2: iadd
       iinc  1 1       ; i++
Test2: iload_1
       iload_0
       if_icmple Loop2 ; if (i <= n) goto Loop2
       ireturn
```

The number of words residing on the operand stack prior to executing each instruction in a method with flow graph $G = \langle V, E, v_0 \rangle$ is easily determined by invoking the algorithm shown in Figure 8 as `comp_sp`$(v_0, 0)$. Here, set $\mathrm{Instr}(v)$ denotes all the bytecode instructions in a basic block $v \in V$ and $\mathrm{Succ}(v) = \{s \in V \mid (v, s) \in E\}$.

Function `sp_pre`(i) yields the number of words popped from the operand stack by i. For most instructions, this number is fixed, whereas for `putfield`, `putstatic`, `invoke<kind>`, and `multianewarray`, this number can be determined from the bytecode context (i.e. a field or method descriptor in the constant pool, or an operand of the opcode [26]). Likewise, `sp_post`(i) yields the number of words that are pushed back on the operand stack by i. This number is usually fixed, or can be determined from the bytecode context for the instructions `getfield`, `getstatic`, and `invoke<kind>` (for invocations, `sp_post`(i), in fact, yields the number of words that are pushed back onto the stack by the corresponding `<T>return` instruction in the callee).




```
*** method 0x08 iterativeFac(I)I 1->1
----------
[ 0]   0:  iconst_1
[ 1]   1:  istore_1
[ 0]   2:  goto 12
----------
[ 0]   5:  iload_1
[ 1]   6:  iload_0
[ 2]   7:  iinc 0 -1
[ 2]  10:  imul
[ 1]  11:  istore_1
----------
[ 0]  12:  iload_0
[ 1]  13:  iconst_1
[ 2]  14:  if_icmpgt 5
----------
[ 0]  17:  iload_1
[ 1]  18:  ireturn
----------
```

Locals	
0:	n
1:	fac

Figure 9: Flow graph and bytecode of class method `iterativeFac()`

For each instruction i in a basic block, an associated attribute i`.sp` is used to store the operand stack size in words prior to executing the instruction. This attribute is initialized to the value '$\perp$', so that the algorithm can test whether the instruction is visited for the first time.

During the first visit, the current stack size is stored in i`.sp`, and the effects of executing i on the stack size are determined. In addition, a test for stack underflow or overflow are performed, where we assume that `max_stack` denotes the maximum number of words that may reside on the operand stack size for the current method. After all instructions in a basic block $v \in V$ have been handled successfully, the algorithm is applied recursively to all successors $s \in \mathrm{Succ}(v)$ with either the current stack size, or stack size 1 in case s is an exception handler for v (i.e. edge $(v, s) \in E$ is only taken if the last instruction in v throws an exception).

On the other hand, if an instruction already has been visited, we simply compare the current stack size with the previously recorded stack size. If these sizes are consistent, further processing of the basic block is terminated, while an error is reported otherwise.

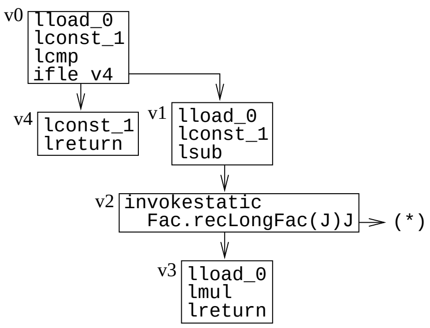
Consider, for instance, the following Java class method `iterativeFac()` that computes the factorial of parameter `n`:

```
static int iterativeFac(int n) {
  int fac = 1;
  while (n > 1)
    fac *= (n--);
  return fac;
}
```

In Figure 9, the bytecode and corresponding flow graph $G = \langle V, E, v_0 \rangle$ for this method are shown. The number of words that reside on the stack prior to executing each instruction are summarized in between square brackets before each instruction. Initially, in basic block v_0, zero words reside on the stack. After `iconst_1` has been executed, 1 word resides on the stack, which is popped again by `istore_1`. Hence, the stack is empty on entry of basic block v_1. Because on exit of v_1, the stack is empty again, basic blocks v_2 and v_3 are visited with an initial empty stack. Basic block v_3 *as a whole* also leaves the stack empty. Following the edge from v_3 to v_1, we see that the latter basic block already has been visited, and that the current stack size is consistent with a previously computed size. Finally, v_2 is visited, which returns a single integer that is loaded on the stack.

Consider, as another example, a recursive version `recLongFac()` of the previous method with a `long` parameter that appears in the following class `Fac`:

```
class Fac {
  static long recLongFac(long n) {
    return (n > 1) ? recLongFac(n-1) * n : 1;
  }
}
```




```
*** method 0x08 recLongFac(J)J 2->2
----------
[ 0]   0:  lload_0
[ 2]   1:  lconst_1
[ 4]   2:  lcmp
[ 1]   3:  ifle 15
----------
[ 0]   6:  lload_0
[ 2]   7:  lconst_1
[ 4]   8:  lsub
----------
*[ 2]  9:  invokestatic Fac.recLongFac (J)J
----------
[ 2] 12:  lload_0
[ 4] 13:  lmul
[ 2] 14:  lreturn
----------
[ 0] 15:  lconst_1
[ 2] 16:  lreturn
----------
```

Locals	
0:	n

Figure 10: Flow graph and bytecode of class method `recLongFac()`

Stack size information for the corresponding byte code is summarized in Figure 10. In this case, the method descriptor '`(J)J`' is used to determine that `invokestatic` takes 2 words from the stack and that 2 words reside on the stack upon completion of this instruction (i.e. after the corresponding `lreturn` has been executed).

2.7 Partial Stack States

In order to provide the compiler with more opportunities for bytecode transformations, it is desirable to have some compile-time information about the actual contents of the operand stack prior to executing each individual instruction. In this section, we present our approach to the problem of computing stack state information as a forward, all-paths (must) data-flow problem. To keep analysis time limited, we will trace stack states partially, namely only with respect to some constants, locals, and array references.

Given a bytecode method where the maximum number of words on the operand stack is k, we define the **stack state** $\overline{s}$ as a k-tuple of the following form, where c denotes an integer literal, D is a set of definitions, and a is an address of a bytecode instruction:

$$\overline{s} = \langle s_0, \ldots, s_{k-1} \rangle \qquad \text{where} \quad s_j = \begin{cases} \top \\ \mathbf{exp}(c, D) \\ \mathbf{ref}(a) \\ \bot \end{cases}$$

Here, $s_j = \top$, $s_j = \mathbf{exp}(c, D)$, $s_j = \mathbf{ref}(a)$, and $s_j = \bot$ denotes that that word j on the operand stack is uninitialized, contains an addition of a constant c and a local defined by a definition $d \in D$, an array reference generated by an instruction at address a, or an unknown or 'irrelevant' value, respectively.[3] In particular, under the assumption that no local is used before defined [26, ch8], we let $\mathbf{exp}(c, D)$ represent a single constant for $D = \emptyset$, which will be denoted as $\mathbf{con}(c)$, or a single local defined by a $d \in D$ for $c = 0$, which will be denoted as $\mathbf{var}(D)$.

We define a component-wise meet operation $\overline{s} \wedge \overline{s}\,' = \langle s_0 \wedge s'_0, \ldots, s_{k-1} \wedge s'_{k-1} \rangle$, where $s_j \wedge s'_j$ is defined by the following table and condition τ as $(c = c') \wedge [(D \cup D' = \emptyset) \vee (D \neq \emptyset \wedge D' \neq \emptyset)]$:

[3] In a more advanced scheme, we could also record *sets* of constants c and addresses a.

$s_j \wedge s'_j$	$s'_j = \top$	$s'_j = \mathbf{exp}(c', D')$	$s'_j = \mathbf{ref}(a')$	$s'_j = \bot$
$s_j = \top$	$\top$	$\mathbf{exp}(c', D')$	$\mathbf{ref}(a')$	$\bot$
$s_j = \mathbf{exp}(c, D)$	$\mathbf{exp}(c, D)$	$\tau\,?\,\mathbf{exp}(c, D \cup D') : \bot$	$\bot$	$\bot$
$s_j = \mathbf{ref}(a)$	$\mathbf{ref}(a)$	$\bot$	$(a = a')\,?\,\mathbf{ref}(a) : \bot$	$\bot$
$s_j = \bot$	$\bot$	$\bot$	$\bot$	$\bot$

For a method with flow graph $G = \langle V, E, v_0 \rangle$ and maximum stack size k, we first compute the **stack state modification** s_mod$[v]$ for each basic block by invoking the algorithm shown in Figure 11 as `comp_mod_tup`(v) for each $v \in V$. In this method, we record the effects on the stack state for a subset of bytecode instructions, and simply set all affected stack words to '$\bot$' for the other instructions. In the branch that implements a simple form of constant folding [1, ch10], operator $\mathcal{C}_i$ performs the appropriate operation on stack state in case the result of the operation can be expressed as a new stack state component. Here, we use one of the following constant folding rules (recall that $\mathbf{exp}(c, \emptyset) = \mathbf{con}(c)$ and $\mathbf{exp}(0, D) = \mathbf{var}(D)$), or set the resulting component to '$\bot$' if none of these rules is applicable:

$$\begin{array}{llcll}
\mathbf{exp}(c, D) & + & \mathbf{con}(c') & = & \mathbf{exp}(c + c', D) \\
\mathbf{con}(c) & + & \mathbf{exp}(c', D') & = & \mathbf{exp}(c + c', D') \\
\mathbf{exp}(c, D) & - & \mathbf{con}(c') & = & \mathbf{exp}(c - c', D) \\
\mathbf{con}(c) & * & \mathbf{con}(c') & = & \mathbf{con}(c * c') \\
 & - & \mathbf{con}(c) & = & \mathbf{con}(-c')
\end{array}$$

Finally, we define the operation $\Psi(\overline{s}, \overline{s}\,')$ to model the effects of a whole basic block as follows:

$$\Psi(\overline{s}, \overline{s}\,') \;=\; \langle\; \psi(s_0, s'_0), \ldots, \psi(s_{k-1}, s'_{k-1}) \;\rangle \qquad \text{where } \psi(s_j, s'_j) = \begin{cases} s'_j & \textbf{if } s_j = \top \\ s_j & \textbf{otherwise} \end{cases}$$

Now, global stack state information can be found by solving the following data flow equations for sets s_in$[v]$ and s_out$[v]$ over all $v \in V$:

$$\begin{cases} \text{s\_in}[v] & = & \bigwedge_{p \in \mathrm{Pred}(v)} \text{s\_out}[p] \\ & & \\ \text{s\_out}[v] & = & \Psi(\text{s\_mod}[v], \text{s\_in}[v]) \end{cases} \tag{3}$$

The iterative algorithm `comp_stack_states()` shown in Figure 12 is used to solve these data flow equations. Eventually, the global stack state information computed by this algorithm can be converted into local stack state information (i.e. stack state information for a particular instruction) by applying an algorithm that is similar to the method of Figure 11 to the derived s_in$[v]$ tuple up to the instruction in a basic block $v \in V$ for which the stack state information is desired.

Consider the following Java class method `init()` that initializes the first `n` elements of a one-dimensional float array `a` with either the value `1.0f` of `2.0f`, as indicated by a one-dimensional boolean array `b`:

```
static void init(float[] a, boolean b[], int n) {
  for (int i = 0; i < n; i++)
     a[i] = (b[i]) ? 1.0f : 2.0f;
}
```

In Figure 13, we present the bytecode for this method, as well as the corresponding flow graph $G = \langle V, E, v_0 \rangle$. Method descriptor '`([F[ZI)V`' indicates that three definitions d_1 d_2, and d_3 arise from invoking this method. The other definitions are formed by instructions at addresses 1 and 19:

	d_1	d_2	d_3	d_4	d_5
local	0	1	2	3	3
address	$\bot$	$\bot$	$\bot$	1	19
loc. par	`[F`	`[Z`	`I`		

```
procedure comp_mod_tup(v) {

  ⟨s_0, ..., s_{k-1}⟩ = ⟨⊤, ..., ⊤⟩;

  foreach i ∈ Instr(v) in order {

     sp    := i.sp;
     minsp := sp    - sp_pre(i);
     newsp := minsp + sp_pos(i);

     if i ∈ { iconst_<c>, bipush c, sipush c, ldc c, ldc_w c } {
       s_sp := con(c);
     }
     else if i ∈ { iload_<l>, iload l, aload_<l>, aload l } {
       let u denote this use of local l;
       s_sp := var(UD(u));           // note that UD(u) ≠ ∅
     }
     else if i ∈ { aaload, anewarray, newarray, multianewarray } {
       let a denote address of i;
       for j := minsp to max(newsp, sp) - 1
         s_j := ⊥;
       s_newsp-1 := ref(a);
     }
     else if i ∈ { dup, dup<kind>, swap } {
       modify s̄ accordingly;
     }
     else if i ∈ { iadd, isub, imul, ineg } {
       for j := minsp to max(newsp, sp) - 1
         s_j := ⊥;
       s_newsp-1 := C_i(s̄);          // constant folding
     }
     else {
       for j := minsp to max(newsp, sp) - 1
         s_j := ⊥;
     }
  }

  s_mod[v] := ⟨s_0, ..., s_{k-1}⟩;
}
```

Figure 11: Computing stack state modification

Because $v_0 \in V$ pushes and pops an integer 0 on an initially empty stack, for example, we have s_mod$[v_0] = \langle \perp, \top, \top, \top \rangle$ (because sp=0 after v_0, the $\perp$ means that the first word on the stack is killed). As another example, because v_8 pushes a floating point constant on a stack that already contains two words, we have s_mod$[v_8] = \langle \top, \top, \perp, \top \rangle$ (because sp=3 after v_8, the $\perp$ now means that something 'irrelevant' resides in the third stack word). After all stack state modification tuples have been computed, the algorithm of Figure 12 is used to determine the global stack state information. The resulting tuples are shown below.

For example, s_in$[v_6] = \langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$ denotes that prior to execution of v_6, two locals reside on operand stack. The first local is reached by d_1 only, while the second local is reached by definitions d_4 and d_5.

v	s_mod$[v]$	s_in$[v]$	s_out$[v]$
v_0	$\langle \perp, \top, \top, \top \rangle$	$\langle \top, \top, \top, \top \rangle$	$\langle \perp, \top, \top, \top \rangle$
v_1	$\langle \perp, \perp, \top, \top \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$
v_2	$\langle \top, \top, \top, \top \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$
v_3	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$
v_4	$\langle \top, \top, \perp, \top \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$
v_5	$\langle \top, \top, \perp, \top \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$
v_6	$\langle \perp, \perp, \perp, \top \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$
v_7	$\langle \top, \top, \top, \top \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$	$\langle \perp, \perp, \perp, \perp \rangle$
v_8	$\langle \top, \top, \perp, \top \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$

```
procedure comp_stack_states() {
  foreach v ∈ V
    s_out[v] := s_mod[v]

  do {
    busy := false;
    foreach v ∈ V {

      s_in[v] :=    ⋀         s_out[p];          // yields  ⟨⊤,...,⊤⟩  if Pred(v) = ∅
                 p∈Pred(v)

      O := Ψ(s_mod[v],s_in[v]);

      if (s_out[v] ≠ O) {
        busy := true;
        s_out[v] := O;
      }
    }
  } while (busy);
}
```

Figure 12: Computing partial stack states

2.8 Copy and Constant Propagation

Local stack state information can frequently be made slightly more accurate by using the following bytecode variants of **copy** and **constant propagation** [1, 3, 15, 35, 46, 47] on stack states. Under the assumption that no local is used before defined [26, ch8], the following replacements can be applied repetitively to the stack states associated with all instructions of a bytecode method.

Given an instruction i and an arbitrary component $s_j = \mathbf{exp}(c, D)$ of the stack state $\overline{s}$ associated with this instruction, then:

- If $c = 0$ and for a fixed address a', each $d \in D$ is formed by a store instruction i' with $\mathbf{ref}(a')$ on top of the stack state $\overline{s}\,'$ associated with i', then we may replace s_j by $\mathbf{ref}(a')$.
- If for a fixed c' and D', each $d \in D$ is formed by a store instruction i' with $\mathbf{exp}(c', D')$ on top of the stack state $\overline{s}\,'$ associated with i', then we may replace s_j by $\mathbf{exp}(c + c', D')$.

After the last replacement, it may be useful to further propagate this more accurate stack state information to determine whether any subsequent constant folding now becomes applicable (in our prototype, this is only done locally, i.e. within the same basic block).

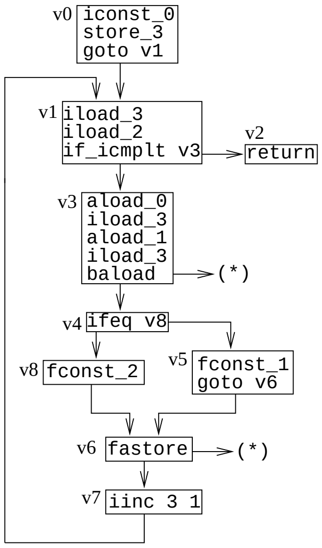
The bytecode and flow graph $G = \langle V, E, v_0 \rangle$, where $V = \{v_0, v_1, v_2, v_3\}$, of the following Java class method `copy()`, for example, are shown in Figure 14:

```
static int copy(boolean b) {
  int i = 1, j = i, k = j;
  return ((b) ? j : k) + 1;
}
```

In the bytecode, the following definitions of locals can be distinguished. Definition d_1 arises from passing the boolean parameter `b` to this method, as indicated by method descriptor '`(Z)I`':

	d_1	d_2	d_3	d_4
local	0	1	2	3
address	$\perp$	1	3	5
par. type	Z			

First, the stack state modification tuple s_mod[v] is computed for each $v \in V$. For example, because instruction `iload_3` in $v_3 \in V$ loads local 3 on an initially empty stack and $\mathrm{UD}(u) = \{d_4\}$ for this use u (viz. rd_in[v_3] $= \{d_1, d_2, d_3, d_4\}$), we have s_mod[v_3] $= \langle \mathbf{var}(\{d_4\}), \top \rangle$.




```
*** method 0x08 init([F[ZI)V 3->0
----------
[ 0]   0:  iconst_0
[ 1]   1:  istore_3
[ 0]   2:  goto 22
----------
[ 0]   5:  aload_0
[ 1]   6:  iload_3
[ 2]   7:  aload_1
[ 3]   8:  iload_3
*[ 4]   9:  baload
----------
[ 3]  10:  ifeq 17
----------
[ 2]  13:  fconst_1
[ 3]  14:  goto 18
----------
[ 2]  17:  fconst_2
----------
*[ 3]  18:  fastore
----------
[ 0]  19:  iinc 3 1
----------
[ 0]  22:  iload_3
[ 1]  23:  iload_2
[ 2]  24:  if_icmplt 5
----------
[ 0]  27:  return
----------
```

	Locals
0:	a
1:	b
2:	n
3:	i

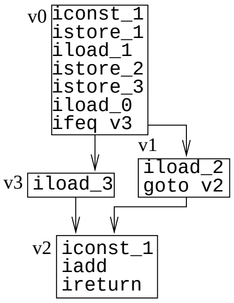
Figure 13: Flow graph and bytecode of class method `init()`

All other stack state modifications tuples are computed in a similar way. Subsequently, the algorithm of Figure 12 yields the following global stack state information:

v	s_mod$[v]$	s_in$[v]$	s_out$[v]$
v_0	$\langle\bot,\top\rangle$	$\langle\top,\top\rangle$	$\langle\bot,\top\rangle$
v_1	$\langle\mathbf{var}(\{d_3\}),\top\rangle$	$\langle\bot,\top\rangle$	$\langle\mathbf{var}(\{d_3\}),\top\rangle$
v_2	$\langle\bot,\bot\rangle$	$\langle\mathbf{var}(\{d_3,d_4\}),\top\rangle$	$\langle\bot,\bot\rangle$
v_3	$\langle\mathbf{var}(\{d_4\}),\top\rangle$	$\langle\bot,\top\rangle$	$\langle\mathbf{var}(\{d_4\}),\top\rangle$

This global stack state information can be converted into local stack state information by using an algorithm similar to algorithm `comp_mod_tup()` of Figure 11 to convert s_in$[v]$ into the stack state for every individual instruction of a basic block $v \in V$. The resulting local stack state information is shown in the following table, where the stack state prior to executing an instruction is associated with this instruction. Moreover, the improved local stack state information after applying copy and constant propagation are shown:

code	original stack states	improved stack states
`d1: *** method 0x08 copy(Z)I 1->1`		
`    ----------`		
`    [ 0]   0:  iconst_1`	$\langle\top,\top\rangle$	
`d2: [ 1]   1:  istore_1`	$\langle\mathbf{con}(1),\top\rangle$	
`    [ 0]   2:  iload_1`	$\langle\bot,\top\rangle$	
`d3: [ 1]   3:  istore_2`	$\langle\mathbf{var}(\{d_2\}),\top\rangle$	$\langle\mathbf{con}(1),\top\rangle$
`    [ 0]   4:  iload_2`	$\langle\bot,\top\rangle$	
`d4: [ 1]   5:  istore_3`	$\langle\mathbf{var}(\{d_3\}),\top\rangle$	$\langle\mathbf{con}(1),\top\rangle$
`    [ 0]   6:  iload_0`	$\langle\bot,\top\rangle$	
`    [ 1]   7:  ifeq 14`	$\langle\mathbf{var}(\{d_1\}),\top\rangle$	
`    ----------`		
`    [ 0]  10:  iload_2`	$\langle\bot,\top\rangle$	
`    [ 1]  11:  goto 15`	$\langle\mathbf{var}(\{d_3\}),\top\rangle$	$\langle\mathbf{con}(1),\top\rangle$
`    ----------`		
`    [ 0]  14:  iload_3`	$\langle\bot,\top\rangle$	
`    ---------`		
`    [ 1]  15:  iconst_1`	$\langle\mathbf{var}(\{d_3,d_4\}),\top\rangle$	$\langle\mathbf{con}(1),\top\rangle$
`    [ 2]  16:  iadd`	$\langle\mathbf{var}(\{d_3,d_4\}),\mathbf{con}(1)\rangle$	$\langle\mathbf{con}(1),\mathbf{con}(1)\rangle$
`    [ 1]  17:  ireturn`	$\langle\mathbf{exp}(1,\{d_3,d_4\}),\bot\rangle$	$\langle\mathbf{con}(2),\bot\rangle$




```
*** method 0x08 copy(Z)I 1->1
----------
[ 0]   0:  iconst_1
[ 1]   1:  istore_1
[ 0]   2:  iload_1
[ 1]   3:  istore_2
[ 0]   4:  iload_2
[ 1]   5:  istore_3
[ 0]   6:  iload_0
[ 1]   7:  ifeq 14
----------
[ 0]  10:  iload_2
[ 1]  11:  goto 15
----------
[ 0]  14:  iload_3
----------
[ 1]  15:  iconst_1
[ 2]  16:  iadd
[ 1]  17:  ireturn
----------
```

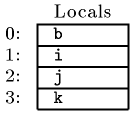


Figure 14: Flow graph and bytecode of class method `copy()`

For example, let i denote the instruction at address 5 with associated stack state $\langle \mathbf{var}(\{d_3\}), \top \rangle$, where the only definition d_3 is formed by instruction i' at address 3. Since i' denotes `istore_2` with $\mathbf{var}(\{d_2\})$ on top of the stack state (viz. `sp=1` holds prior to executing i'), we may replace component $\mathbf{var}(\{d_3\})$ with $\mathbf{var}(\{d_2\})$. This corresponds to propagating copy '`j=i`' in statement '`k=j`' into '`k=i`'. Subsequently, the component may be replaced by $\mathbf{con}(1)$, because the only definition d_2 is formed by a store instruction with the constant 1 on top of the operand stack. Alternatively, if copy propagation is first applied to the instruction at 3, then copy propagation becomes directly applicable to the original component $\mathbf{var}(\{d_3\})$ of the stack state associated with the instruction at 5. As another example, component $\mathbf{exp}(1, \{d_3, d_4\})$ of the `ireturn` instruction at address 17 may be replaced by $\mathbf{con}(2)$ after constant propagation has been applied to both the definitions d_3 and d_4. Anyway, independent of the order in the replacements are done, eventually the improved stack state information shown above results.

In this case, the replacements of this section directly reveal that this method always returns the constant 2, because one of the constant folding rules of Section 2.7 could combine components $\mathbf{var}(\{d_3, d_4\})$ and $\mathbf{con}(1)$ into another component $\mathbf{exp}(1, \{d_3, d_4\})$ for the `iadd` instruction. For an `imul` instruction, however, it would be useful to propagate the improved stack state locally, in order to apply a constant folding rule that was not formerly applicable:

code	original stack states	locally propagated improved stack states	constant folding
...			
`[ 1] 15: iconst_1`	$\langle \mathbf{var}(\{d_3, d_4\}), \top \rangle$	$\langle \mathbf{con}(1), \top \rangle$	
`[ 2] 16: imul`	$\langle \mathbf{var}(\{d_3, d_4\}), \mathbf{con}(1) \rangle$	$\langle \mathbf{con}(1), \mathbf{con}(1) \rangle$	
`[ 1] 17: ireturn`	$\langle \bot, \bot \rangle$	$\langle \bot, \bot \rangle$	$\langle \mathbf{con}(1), \bot \rangle$

2.9 Array Reference Chasing

After partial stack states have been computed for a method, and after copy and constant propagation have been accounted for, the compiler can use this information to partially trace the usage of arrays in a bytecode program.

First, each $\mathbf{var}(\{d\})$ component, where the single definition d is formed by an m-dimensional array parameter in local l, is replaced by a new component $\mathbf{apar}(l, m)$ (the number of leading '`[`' characters in an individual parameter descriptor in a method descriptor defines the number of dimensions for each array parameter [26, ch4]). Subsequently, the compiler attempts to chase each $\mathbf{ref}(a)$ component in the local stack state information back to the corresponding parameters or declarations using algorithm `chase_array_refs()` of Figure 15. If $p \geq n$ holds at the end of a chase

```
procedure chase_array_refs() {
  foreach v ∈ V {
    foreach i ∈ Instr(v) {
    let s̄ denote stack state associated with i;
      for j := 0, k-1
         if (s_j = ref(a))
            s_j := chase(s_j, 0);
    }
  }
}
```

```
function chase(r, p) returns state {
  if (r = apar(l, n)) {
    return (p < n) ? apar(l, n − p) : ⊥;
  }
  else if (r = adecl(a, n)) {
    return (p < n) ? adecl(a, n − p) : ⊥;
  }
  else if (r = ref(a)) {
    let i denote instruction at a;
    switch(i) {
      case aaload:
          let s̄ denote stack state associated with i;
          return chase(s_{i_sp − 2}, p + 1);
      case anewarray:
      case newarray:
      case multianewarray:
          let n denote dimension defined by i;
          return (p < n) ? adecl(a, n − p) : ⊥;
    }
  }
  return ⊥;
}
```

Figure 15: Array reference chasing

in this algorithm, we cannot fully chase the effects of `aaload` instructions, and the corresponding component is replaced by a '$\perp$' in the stack state. Likewise, '$\perp$' results if the chase fails due to lack of more accurate stack state information. In all other cases, $s_j = \mathbf{apar}(l, n)$ or $s_j = \mathbf{adecl}(a, n)$ indicates that word j of the operands stack contains a reference to the last n-dimensions of some m-dimensional array ($n \leq m$) that is defined by either the parameter in the lth local of the method, or the instruction at address a, respectively.

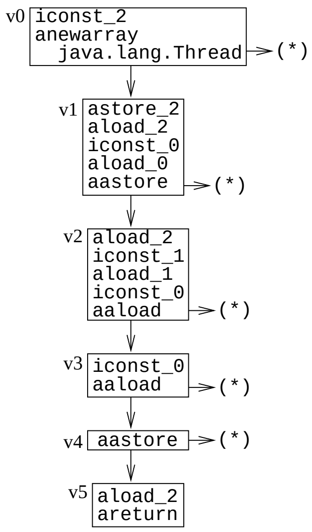
Consider the following Java instance method `decl()` in a class '`Decl`' that is a subclass of the class '`java.lang.Thread`':

```
class Decl extends Thread {
  Thread[] decl(Thread[][] t) {
    Thread[] nt = new Thread[2];
    nt[0] = this;
    nt[1] = t[0][0];
    return nt;
  }
}
```

The bytecode, corresponding flow graph $G = \langle V, E, v_0 \rangle$, and local variables usage are shown in Figure 16. Because `decl()` is an instance method, the first definition d_1 arises from passing reference `this` of type $t_1 =$'`LDecl;`' to the method (here we assume that the class declaration appears in the default package). Furthermore, the method descriptor of `decl()` indicates that there is another two-dimensional array parameter that gives rise to definition d_2 of type $t_2 =$'`[[Ljava/lang/Thread;`'. Finally, a definition is formed by the bytecode instruction at address 4:

	d_1	d_2	d_3
local	0	1	2
address	$\perp$	$\perp$	4
par. type	t_1	t_2	

Below, we show the stack states associated with each bytecode instruction after copy and constant propagation (initial stack states) and after array reference chasing (chased stack states). For example, the component $\mathbf{ref}(1)$ associated with the store at address 4 is replaced by $\mathbf{adecl}(1, 1)$ to indicate that this instruction effectively stores a reference to the whole one-dimensional array defined by the `anewarray` at address 1. Likewise, each $\mathbf{var}(\{d_2\})$ component is replaced by $\mathbf{apar}(1, 2)$, indicating a reference to the two-dimensional array passed as parameter into local 1. The `aaload`




```
*** method 0x00 decl([[Ljava/lang/Thread;)
                     [Ljava/lang/Thread; 2->1
 ----------
 [ 0]   0:  iconst_2
*[ 1]   1:  anewarray java.lang.Thread
 ----------
 [ 1]   4:  astore_2
 [ 0]   5:  aload_2
 [ 1]   6:  iconst_0
 [ 2]   7:  aload_0
*[ 3]   8:  aastore
 ----------
 [ 0]   9:  aload_2
 [ 1]  10:  iconst_1
 [ 2]  11:  aload_1
 [ 3]  12:  iconst_0
*[ 4]  13:  aaload
 ----------
 [ 3]  14:  iconst_0
*[ 4]  15:  aaload
 ----------
*[ 3]  16:  aastore
 ----------
 [ 0]  17:  aload_2
 [ 1]  18:  areturn
 ----------
```

Locals	
0:	this
1:	t
2:	nt

Figure 16: Flow graph and bytecode of instance method `decl()`

instruction at address 13 effectively converts this reference into $\mathbf{apar}(1,1)$, i.e. a reference to the last dimension of this array. Finally, note that the component $\mathbf{ref}(15)$ associated with the `aastore` at address 16 becomes a '$\perp$' after it has been chased back to the `aaload` at 15:

code	initial stack states	chased stack states
`d1,2:*** method 0x00 decl...`		
`----------`		
`[ 0]  0:  iconst_2`	$\langle \top, \top, \top, \top \rangle$	
`*[ 1]  1:  anewarray ...`	$\langle \mathbf{con}(2), \top, \top, \top \rangle$	
`----------`		
`d3:  [ 1]  4:  astore_2`	$\langle \mathbf{ref}(1), \top, \top, \top \rangle$	$\langle \mathbf{adecl}(1,1), \top, \top, \top \rangle$
`[ 0]  5:  aload_2`	$\langle \perp, \top, \top, \top \rangle$	
`[ 1]  6:  iconst_0`	$\langle \mathbf{ref}(1), \top, \top, \top \rangle$	$\langle \mathbf{adecl}(1,1), \top, \top, \top \rangle$
`[ 2]  7:  aload_0`	$\langle \mathbf{ref}(1), \mathbf{con}(0), \top, \top \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(0), \top, \top \rangle$
`*[ 3]  8:  aastore`	$\langle \mathbf{ref}(1), \mathbf{con}(0), \mathbf{var}(\{d_1\}), \top \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(0), \mathbf{var}(\{d_1\}), \top \rangle$
`----------`		
`[ 0]  9:  aload_2`	$\langle \perp, \perp, \perp, \top \rangle$	
`[ 1] 10:  iconst_1`	$\langle \mathbf{ref}(1), \perp, \perp, \top \rangle$	$\langle \mathbf{adecl}(1,1), \perp, \perp, \top \rangle$
`[ 2] 11:  aload_1`	$\langle \mathbf{ref}(1), \mathbf{con}(1), \perp, \top \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(1), \perp, \top \rangle$
`[ 3] 12:  iconst_0`	$\langle \mathbf{ref}(1), \mathbf{con}(1), \mathbf{var}(\{d_2\}), \top \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(1), \mathbf{apar}(1,2), \top \rangle$
`*[ 4] 13:  aaload`	$\langle \mathbf{ref}(1), \mathbf{con}(1), \mathbf{var}(\{d_2\}), \mathbf{con}(0) \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(1), \mathbf{apar}(1,2), \mathbf{con}(0) \rangle$
`----------`		
`[ 3] 14:  iconst_0`	$\langle \mathbf{ref}(1), \mathbf{con}(1), \mathbf{ref}(13), \perp \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(1), \mathbf{apar}(1,1), \perp \rangle$
`*[ 4] 15:  aaload`	$\langle \mathbf{ref}(1), \mathbf{con}(1), \mathbf{ref}(13), \mathbf{con}(0) \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(1), \mathbf{apar}(1,1), \mathbf{con}(0) \rangle$
`----------`		
`*[ 3] 16:  aastore`	$\langle \mathbf{ref}(1), \mathbf{con}(1), \mathbf{ref}(15), \perp \rangle$	$\langle \mathbf{adecl}(1,1), \mathbf{con}(1), \perp, \perp \rangle$
`----------`		
`[ 0] 17:  aload_2`	$\langle \perp, \perp, \perp, \perp \rangle$	
`[ 1] 18:  areturn`	$\langle \mathbf{ref}(1), \perp, \perp, \perp \rangle$	$\langle \mathbf{adecl}(1,1), \perp, \perp, \perp \rangle$

3 Bytecode Parallelization

After the bytecode of a method has been analyzed, all natural loops are examined by the compiler to automatically detect and exploit implicit parallelism in these loops. In this section, we first define a simple form of natural loop: trivial loops. Thereafter, we discuss how implicit parallelism in trivial loops can be automatically detected and exploited. Rather than using advanced data dependence analysis (see e.g. [2, 5, 8, 23, 25, 30, 31, 33, 34, 36, 37, 39, 38, 45, 46, 47]), our prototype relies on simple, but generally also less expensive data dependence tests to automatically determine whether the different iterations of a natural loop are independent and, hence, can be executed in parallel.

3.1 Trivial Loops

Given a flow graph $G = \langle V, E, v_0 \rangle$, we call a natural loop $L \subseteq V$ defined by a back-edge $(g, h) \in E$ a **trivial loop** with a local `i` as loop index, if the following constraints are satisfied:

- There is exactly one *normal* loop-exit $(h, e) \in E$. Examination of the bytecode and local stack state information reveals that this loop-exit is a conditional branch that compares index `i` with a stack component $\mathbf{exp}(c, D)$, where no instruction in a basic block $v \in L$ forms a definition $d \in D$.

- The last (non-branching) instruction in basic block v consists of '`iinc i` p', where $p > 0$, and this instruction forms the only definition of `i` in the loop. Furthermore, there is only one definition $d \in \text{rd\_in[h]}$ of this index `i`, and this definition (of the lower bound) is formed by a store instruction.

Conceptually, a trivial loop consists of any positive stride `for`- or `while`-like construct with a single reaching definition of the lower bound and a loop-invariant upper bound. If the loop-body is not executed for the upper bound, then this upper bound is called strict. Otherwise, the upper bound is called non-strict. Conventional loop transformations, such as loop-normalization [47, p174-177], either at bytecode level, or already at source code level, could be used to increase the number of trivial loops in a program.

Consider, for example, the following two bytecode assembler fragments, where we assume that local 0 contains a reference to a one-dimensional float array `a`:

```
        iconst_0                                       iconst_0
        istore_1        ; i = 0                        istore_1        ; i = 0;
        goto Test                               Test:  iload_1
Loop:   aload_0                                        iload_2
        iload_1                                        if_icmpge Exit  ; if (i >= n) goto Exit
        fconst_0                                       aload_0
        fastore         ; a[i] = 0.0f                  iload_1
        iinc 1 1        ; i++                          fconst_0
Test:   iload_2                                        fastore         ; a[i] = 0.0f
        iload_1                                        iinc 1 1        ; i++
        if_icmpgt Loop  ; if (n > i) goto Loop         goto Test
        ...                                     Exit:  ...
```

In Figure 17, the flow graphs for these two fragments are shown. Examination of local stack state information reveals that both `if_icmp<cond>` instructions compare loop index `i` to a loop-invariant variable `n`, and that both loops are exited conditionally as soon as `i` $\geq$ `n`. Since for both loops, the only outer-loop definition of `i` that reaches the loop corresponds to a `istore_1` of constant `0`, both loops are stride-1 trivial loops with a strict upper bound `n`. For comparing conditions '`ge`' and '`gt`', respectively, the upper bounds would be non-strict. Obviously, loops with `iload_2` and `iload_1` interchanged, and comparing conditions '`lt`' and '`le`', or '`le`' and '`lt`', respectively, would form similar trivial loops with strict or non-strict upper bounds. Note that the looping construct of the left fragment is typically generated by the `javac` compiler to reduce the number of overhead instructions around the loop [26, ch7].

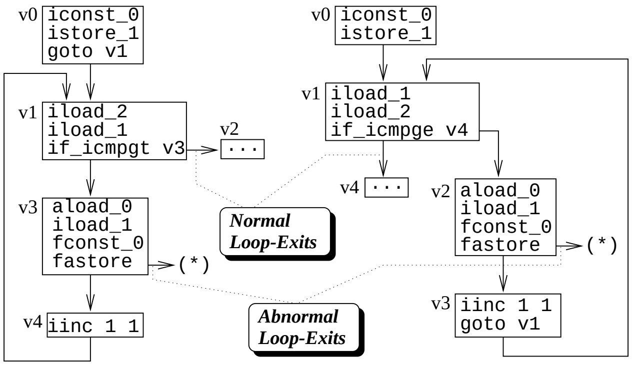


Figure 17: Flow graphs of two trivial loops

3.2 Detection of Implicit Loop Parallelism

A trivial loop $L \subseteq V$ in a method with flow graph $G = \langle V, E, v_0 \rangle$ is a candidate for parallelization if (i) the operand stack is empty on entry and exit of the loop, (ii) the loop-body does not contain a `invoke<kind>`,[4] `putfield`, or `putstatic` instruction, (iii) the compiler can ascertain that *abnormal* loop-exits due to run-time exceptions will not be taken, and (iv) all iterations are independent. Since (i) and (ii) are easily checked, in this section we focus on checking constraints (iii) and (iv).

3.2.1 Analysis of Exceptions

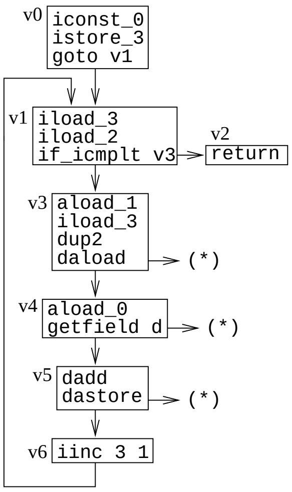
To ascertain that abnormal loop-exits due to run-time exceptions of a candidate parallel loop cannot be taken at run-time, our prototype uses some simple rules to determine that particular instructions that generally may throw run-time exceptions cannot do so within that loop. The following instructions will not throw a linking or run-time exception if the listed constraints are met: (a) `idiv` or `irem`, if a $\textbf{con}(c)$ component, where $c \neq 0$, resides on top of the operand stack prior to executing this instruction, (b) `getfield`, if a $\textbf{var}(\{d\})$ component resides on top of the operand stack prior to executing the instruction, and d is due to passing `this` to an instance method, and (c) `<t>aload` or `<t>astore`, if the instruction refers to a non-`null` array reference and the index on the operand stack is within the bounds of this array (this verification is further discussed in combination with data dependence analysis of arrays in Section 3.2.3).

For all other situations where an instruction that may throw a run-time exception (or a linking exception in case preserving the exact handling semantics of these exceptions is also desired) occurs in a candidate parallel loop, the prototype simply resorts to disabling parallelization of this loop. Moreover, since parallelization of a loop is also be disabled in the presence of method invocations (`invoke<kind>`) or field modifications (`putstatic` or `putfield`), checking whether all iterations of a loop are independent consist of testing if loop-carried data dependences can be caused by the usage of arrays and scalar locals.

Consider, for instance, the Java instance method shown below:

```
class Field {
  double d = 3.0;
  void add_field(double[] a, int n) {
    for (int i = 0; i < n; i++)
      a[i] += d;
  }
}
```

[4]In fact, this constraint can be slightly relaxed by allowing invocations of *side-effect free static methods* such as '`java.lang.Math.cos()`'.




```
*** method 0x00 add_field([DI)V 3->0
 ----------
 [ 0]  0:  iconst_0
 [ 1]  1:  istore_3
 [ 0]  2:  goto 18
 ----------
 [ 0]  5:  aload_1
 [ 1]  6:  iload_3
 [ 2]  7:  dup2
*[ 4]  8:  daload
 ----------
 [ 4]  9:  aload_0
*[ 5] 10:  getfield Field.d D
 ----------
 [ 6] 13:  dadd
*[ 4] 14:  dastore
 ----------
 [ 0] 15:  iinc 3 1
 ----------
 [ 0] 18:  iload_3
 [ 1] 19:  iload_2
 [ 2] 20:  if_icmplt 5
 ----------
 [ 0] 23:  return
 ----------
```

	Locals
0:	this
1:	a
2:	n
3:	i

Figure 18: Flow graph and bytecode of instance method `field()`

The bytecode, corresponding flow graph $G = \langle V, E, v_0 \rangle$, and local variables usage are shown in Figure 18. The following definitions occur in this bytecode, where $t =$‘`LField;`’ holds:

	d_1	d_2	d_3	d_4	d_5
local	0	1	2	3	3
address	$\perp$	$\perp$	$\perp$	1	15
par. type	t	`[D`	`I`		

Local stack state information for instructions in the trivial loop $L = \{v_1, v_3, \ldots, v_6\}$ is shown below:

code	stack states
`...`	
`[ 0]  5:  aload_1`	$\langle \perp, \perp, \perp, \perp, \perp, \perp \rangle$
`[ 1]  6:  iload_3`	$\langle \mathbf{apar}(1,1), \perp, \perp, \perp, \perp, \perp \rangle$
`[ 2]  7:  dup2`	$\langle \mathbf{apar}(1,1), \mathbf{var}(\{d_4, d_5\}), \perp, \perp, \perp, \perp \rangle$
`*[ 4]  8:  daload`	$\langle \mathbf{apar}(1,1), \mathbf{var}(\{d_4, d_5\}), \mathbf{apar}(1,1), \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$
`----------`	
`[ 4]  9:  aload_0`	$\langle \mathbf{apar}(1,1), \mathbf{var}(\{d_4, d_5\}), \perp, \perp, \perp, \perp \rangle$
`*[ 5] 10:  getfield d`	$\langle \mathbf{apar}(1,1), \mathbf{var}(\{d_4, d_5\}), \perp, \perp, \mathbf{var}(\{d_1\}), \perp \rangle$
`----------`	
`[ 6] 13:  dadd`	$\langle \mathbf{apar}(1,1), \mathbf{var}(\{d_4, d_5\}, \perp, \perp, \perp, \perp \rangle$
`*[ 4] 14:  dastore`	$\langle \mathbf{apar}(1,1), \mathbf{var}(\{d_4, d_5\}, \perp, \perp, \perp, \perp \rangle$
`----------`	
`[ 0] 15:  iinc 3 1`	$\langle \perp, \perp, \perp, \perp, \perp, \perp \rangle$
`...`	

Prior to executing instruction `getfield`, component $\mathbf{var}(\{d_1\})$ resides on top of the operand stack. Since this only definition d_1 is due to implicitly passing reference `this` to the instance method, the loop-exit from `getfield` will not be taken. The abnormal loop-exits due to the instructions `daload` at address 8 and `dastore` at address 14 will be analyzed using the method described in 3.2.3.

3.2.2 Analysis of Scalar Local Variables

The different iterations of a trivial loop $L \subseteq V$ defined by a back edge $(g, h) \in E$ can be executed in parallel with respect to local scalars, if the following constraints are satisfied:

(I) For each instruction i in every $v \in L$ forming a use u of a local (other than the loop-index), either:

(a) for all $d' \in \mathrm{UD}(u)$, d' is formed by an instruction i' in a basic block $v' \notin L$, or

(b) there exists a $d' \in \mathrm{UD}(u)$ formed by an instruction i' in a basic block $v' \in L$, such that either $v' \in \mathrm{Dom}(v)$ for $v \neq v'$, or i' appears before i in basic block $v' = v$.

(II) For each instruction in every $v \in L$ that forms a definition d of a local (including the loop-index), there is no instruction in a basic block $v' \notin L$ that forms a use $u' \in \mathrm{DU}(d)$.

Constraint (I) states that every use of a local scalar must receive a value either from instructions that appear outside the loop, or always from an instruction that is executed earlier in the same iteration (note that because for all $v \in L$, $h \in \mathrm{Dom}(v)$ holds for the loop-entry $h \in L$, $v' \in \mathrm{Dom}(v)$ for $v \neq v'$ implies that v' is executed before v in each iteration). This provides a *necessary* (but not *sufficient*) test to ensure that there are no *loop-carried flow dependences* caused by local scalars. *Loop-carried output and anti dependences* caused by local scalars will be resolved in the parallel loop by letting each thread operate on a separate set of locals. Constraint (II) states that all local scalars defined in the loop must be **dead** [1, ch10] on exit of the loop.

Consider, for instance, the following Java class method `scalardep()`, the corresponding bytecode and flow graph $G = \langle V, E, v_0 \rangle$ of which are presented in Figure 19:

```
static int scalardep(boolean[] b, int[] a) {
  int i, k = 5;
  for (i = 0; i < 100; i++) {
    if (b[i]) k = 2;
    a[i] = k;
  }
  return k;
}
```

In the bytecode, there are the following definitions of locals, where method descriptor '`([Z[I)I`' indicates that d_1 and d_2 correspond to passing parameters `b` and `a`:

	d_1	d_2	d_3	d_4	d_5	d_6
local	0	1	3	2	3	2
address	$\perp$	$\perp$	1	3	14	19
par. type	`[Z`	`[I`				

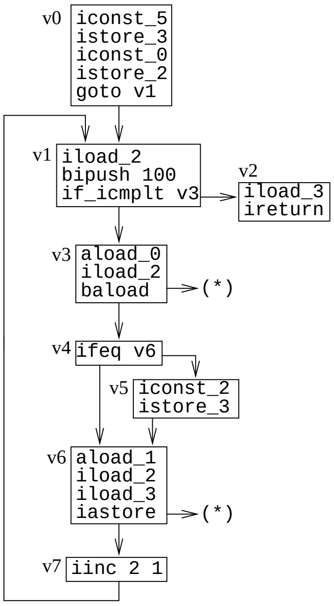
Furthermore, the following uses of local variables can be distinguished:

	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
local	0	2	1	2	3	2	2	3
address	7	8	15	16	17	19	22	28

Global reaching definitions information is computed using the algorithm of Figure 5. The resulting information for the basic blocks $V = \{v_0, \ldots, v_7\}$ is summarized in the following table, where $D = \{d_1, d_2, d_3, d_4, d_5, d_6\}$:

v	rd_gen$[v]$	rd_kill$[v]$	rd_in$[v]$	rd_out$[v]$
v_0	$\{d_3, d_4\}$	$\{d_5, d_6\}$	$\{d_1, d_2\}$	$\{d_1, d_2, d_3, d_4\}$
v_1	$\emptyset$	$\emptyset$	D	D
v_2	$\emptyset$	$\emptyset$	D	D
v_3	$\emptyset$	$\emptyset$	D	D
v_4	$\emptyset$	$\emptyset$	D	D
v_5	$\{d_5\}$	$\{d_3\}$	D	$\{d_1, d_2, d_4, d_5, d_6\}$
v_6	$\emptyset$	$\emptyset$	D	D
v_7	$\{d_6\}$	$\{d_4\}$	D	$\{d_1, d_2, d_3, d_5, d_6\}$

The different iterations of the trivial loop $L = \{v_1, v_3, \ldots, v_7\}$ cannot be executed in parallel with respect to its local scalars because definition d_5 in basic block $v_5 \in L$ reaches the use u_5 in $v_6 \in L$ and $v_5 \notin \mathrm{Dom}(v_6)$ (thereby possibly causing a loop-carried flow-dependence). Moreover, the same definition d_5 in $v_5 \in L$ reaches the use u_8 in basic block $v_2 \notin L$ (thereby causing a loop-exiting flow dependence).




```
*** method 0x08 scalardep([Z[I)I 2->1
----------
[ 0]  0:  iconst_5
[ 1]  1:  istore_3
[ 0]  2:  iconst_0
[ 1]  3:  istore_2
[ 0]  4:  goto 22
----------
[ 0]  7:  aload_0
[ 1]  8:  iload_2
*[ 2]  9:  baload
----------
[ 1] 10:  ifeq 15
----------
[ 0] 13:  iconst_2
[ 1] 14:  istore_3
----------
[ 0] 15:  aload_1
[ 1] 16:  iload_2
[ 2] 17:  iload_3
*[ 3] 18:  iastore
----------
[ 0] 19:  iinc 2 1
----------
[ 0] 22:  iload_2
[ 1] 23:  bipush 100
[ 2] 25:  if_icmplt 7
----------
[ 0] 28:  iload_3
[ 1] 29:  ireturn
----------
```

	Locals
0:	b
1:	a
2:	i
3:	k

Figure 19: Flow graph and bytecode of instance method `scalardep()`

3.2.3 Analysis of Arrays

As a final step in the automatic detection of implicit parallelism, the compiler verifies if none of the array instructions `<t>aload` and `<t>astore` in a candidate parallel loop give rise to loop-carried data dependences or a transfer of control along abnormal loop-exits. For this purpose, the following steps are taken for each candidate parallel loop $L \subseteq V$.

In order to record data dependences, an **access tuple** $\overline{x} = \langle x_0, \ldots, x_{m-1} \rangle$ is associated with each m-dimensional array parameter and array declaration that occurs in the method of the candidate parallel loop. Here, $x_j = \top$, $x_j = [l, u]$, or $x_j = \bot$ means that with respect to the candidate parallel loop with index `i`, the $(j+1)$th dimension of the array is either not referenced, only referenced by subscript expressions of the form '`i + c`' where $c \in [l, u]$, or referenced by arbitrary subscript expressions, respectively. Before examination of a candidate parallel loop, all access tuples are initialized to $\overline{x} = \langle \top, \ldots, \top \rangle$.

Given an arbitrary `<t>aload` or `<t>astore` instruction within the loop-body, let r and s denotes the components on the operand stack that correspond to the array reference and subscript expression, respectively (for a `<t>aload` instruction, r and s are on top of the stack, whereas for a `<t>astore` instruction, r and s reside below the data item to be stored). A lower bound and strict upper bound for subscript expression s can be determined by invoking the auxiliary functions of Figure 20 as `get_lbnd`(a, s) and `get_strict_ubnd`(a, s), where a denotes the address of the `<t>aload` or `<t>astore` instruction under consideration. In these algorithms, we assume that another auxiliary function `get_loop`(a, D) yields either the trivial loop that contains the instruction at a and where the definitions in D are formed by the store instruction of the lower bound and increment instruction of the index of this loop (cf. Section 3.1), or the value '$\bot$' otherwise. In the former case, constructs '`lp.lbnd`' and '`lp.ubnd`' are used to obtain the stack components that correspond to the lower bound and strict upper bound, respectively. Construct '`lp.strict`' denotes whether the upper bound of the trivial loop is strict.

These function can be used, for example, to determine that within the three trivial loops in the bytecode equivalent of the following Java source code, a subscript expression '`k+4`' only takes values

```
function get_lbnd(i, s) returns state {
  if (s = exp(c, D)) {
    lp := get_loop(a, D);
    if (lp = ⊥)
      return exp(c, D);
    else {
      lbnd = get_lbnd(a, lp.lbnd);
      if (lbnd = exp(c′, D′))
        return exp(c + c′, D′);
    }
  }
  return ⊥;
}
```

```
function get_strict_ubnd(a, s) returns state {
  if (s = exp(c, D)) {
    lp := get_loop(a, D);
    if (lp = ⊥)
      return exp(c + 1, D);
    else {
      ubnd := get_strict_ubnd(a, lp.ubnd);
      if (ubnd = exp(c′, D′))
        return (lp.strict) ? exp(c + c′ − 1, D′)
                           : exp(c + c′, D′)
    }
  }
  return ⊥;
}
```

Figure 20: Subscript bounds computation

in the interval $[9, 102)$ (viz. $\texttt{i} \in [1, 500), \texttt{j} \in [4, 101), \texttt{k} \in [5, 98)$):

```
static void bound(int a[]) {
  for (int i = 1; i < 500; i++)
     for (int j = i+3; j <= 100; j++)
       for (int k = j+1; k < j-2; k++)
          a[k+4] = 1;
}
```

code	stack states
...	
[0] 12: iload_2	$\langle \perp, \perp, \perp \rangle$
[1] 13: iconst_1	$\langle \mathbf{var}(\{d_3, d_6\}), \perp, \perp \rangle$
d4: [2] 14: iadd	$\langle \mathbf{var}(\{d_3, d_6\}), \mathbf{con}(1), \perp \rangle$
[1] 15: istore_3	$\langle \mathbf{exp}(1, \{d_3, d_6\}), \perp, \perp \rangle$
[0] 16: goto 28	$\langle \perp, \perp, \perp \rangle$
----------	
[0] 19: aload_0	$\langle \perp, \perp, \perp \rangle$
[1] 20: iload_3	$\langle \mathbf{apar}(0,1), \perp, \perp \rangle$
[2] 21: iconst_4	$\langle \mathbf{apar}(0,1), \mathbf{var}(\{d_4, d_5\}), \perp \rangle$
[3] 22: iadd	$\langle \mathbf{apar}(0,1), \mathbf{var}(\{d_4, d_5\}), \mathbf{con}(4) \rangle$
[2] 23: iconst_1	$\langle \mathbf{apar}(0,1), \mathbf{exp}(4, \{d_4, d_5\}), \perp \rangle$
*[3] 24: iastore	$\langle \mathbf{apar}(0,1), \mathbf{exp}(4, \{d_4, d_5\}), \mathbf{con}(1) \rangle$
----------	
d5: [0] 25: iinc 3 1	$\langle \perp, \perp, \perp \rangle$
----------	
[0] 28: iload_3	$\langle \perp, \perp, \perp \rangle$
[1] 29: iload_2	$\langle \mathbf{var}(\{d_4, d_5\}), \perp, \perp \rangle$
[2] 30: iconst_2	$\langle \mathbf{var}(\{d_4, d_5\}), \mathbf{var}(\{d_3, d_6\}), \perp \rangle$
[3] 31: isub	$\langle \mathbf{var}(\{d_4, d_5\}), \mathbf{var}(\{d_3, d_6\}), \mathbf{con}(2) \rangle$
[2] 32: if_icmplt 19	$\langle \mathbf{var}(\{d_4, d_5\}), \mathbf{exp}(-2, \{d_3, d_6\}), \perp \rangle$
...	

Since the upper bound of the k-loop is strict, a strict upper bound of $\mathbf{exp}(4, \{d_4, d_5\})$ (viz. '`k+4`') is given by $\mathbf{exp}(4 + c' - 1, D')$, where $\mathbf{exp}(c', D')$ is a strict upper bound of $\mathbf{exp}(-2, \{d_3, d_6\})$ (viz. '`j-2`'). Because the j-loop has a non-strict upper bound, a strict upper bound of the latter expression is given by $\mathbf{exp}(-2 + c'', D'')$, where $\mathbf{exp}(c'', D'')$ is a strict upper bound of $\mathbf{con}(100)$. Since this implies that $c'' = 101$ and $D'' = \emptyset$, and hence $c' = 99$ and $D' = \emptyset$, we obtain $\mathbf{con}(102)$ as strict upper bound of the expression '`k+4`'. The lower bound is found similarly.

An array reference is verified by invoking the algorithm of Figure 21 as $\texttt{verify\_ref}(a, r, s)$ for its address a, array reference r, and subscript expression s. If lower bound l is a known non-negative constant, and u is an expression consisting of a constant and possibly a parameter (function $\texttt{is\_par}(D_u)$ yields true in case $D_u = \{d\}$, and this only definition is due to parameter passing), then the array reference r is examined. If $r = \mathbf{adecl}(a', n)$, then the compiler records access in the $(m - n + 1)$th dimension of this array by calling procedure $\texttt{record\_access}(b, \overline{x}, m - n, c)$, where boolean b denotes if subscript s involves the candidate parallel loop currently considered, and $\overline{x}$ is the access tuple of the m-dimensional array declared at a'. Furthermore, the compiler verifies whether u does not exceed the strict upper bound in the $(m - n + 1)$th dimension of the declared m-dimensional array, which is defined by component $s_{i.\texttt{sp}-n}$ on the stack state $\overline{s}$ associated with the declaration instruction i. If $r = \mathbf{apar}(l, n)$, where l is a m-dimensional array parameter, then the compiler also first records access in the $(m - n + 1)$th dimension of this array. For further verification, however, inter-procedural information would be required. In such cases, the compiler

```
function verify_ref(a, r, s) returns boolean {
  if (s = exp(c, D)) {
    l  := get_lbnd(s);
    u  := get_strict_ubnd(s);
    b  := (get_loop(a, D) = candidate parallel loop);

    if ( l = con(c_l), where c_l ≥ 0 and
         u = exp(c_u, D_u), where (D_u = ∅ or is_par(D_u)) ) {

      // known bounds on subscript

      if (r = adecl(a', n)) {  // array declaration
        let s̄ denote stack state associated
        with instruction i at a' and let x̄ denote the access
        tuple of the m-dimensional array declared by i;

        record_access(b, x̄, m − n, c);

        return ( s_{i.sp−n} = exp(c', D') and
                 c_u ≤ c' and D_u = D' );
      }
      else if (r = apar(l, n)) {   // array parameter
        let x̄ denote stack state associated
        with the m-dimensional array parameter l;

        record_access(b, x̄, m − n, c);
        record_query(l, m − n, u);

        return true;
      }
    }
  }
  return false;
}
```

```
procedure record_access(b, x̄, j, c) {
  if (b) {
    if (x_j = ⊤)
      x_j  := [c, c];
    else if (x_j = [l, u])
      x_j  := [min(l, c), max(u, c)];
  }
  else
    x_j  := ⊥;
}
```

Figure 21: Array reference verification

simply records a query: is the strict upper bound in the $(m - n + 1)$th dimension of the array parameter l at least $u = \mathbf{exp}(c_u, D_u)$?

In all cases where subscript bounds may be violated, or where sufficiently accurate stack states information is lacking, function `verify_ref()` returns the value '`false`'. Hence, if a `verify_ref()` invocation fails for a `<t>aload` or `<t>astore` instruction in a candidate parallel stride-p loop, then this loop is kept serial, because abnormal loop-exits may be taken at run-time. Otherwise, the compiler tests for data dependences as follows. For each array parameter or array declaration in the method of this loop that is involved in at least one `<t>astore` instruction (viz. appearing at the left-hand side of an assignment), the compiler verifies whether $x_j = [l, u]$, where $u - l < p$ holds for a component of the access tuple associated with the array. If such a component does not exists, loop-carried data dependences may hold, and parallelization of the loop is disabled.

For example, with respect to the trivial `i`-loop in the bytecode equivalent of the following Java fragment, access tuple $\langle [0, 2], \perp \rangle$ will eventually be associated with the array parameter `a`. Likewise, with respect to the `j`-loop, access tuple $\langle \perp, [-1, +1] \rangle$ will eventually be associated with `a`. Since array `a` appears at the left-hand side of an assignment, the compiler concludes that data dependences may be carried by the stride-2 `i`-loop (viz. $2 - 0 \geq 2$), but that no data dependences can be carried by the stride-3 `j`-loop (viz. $1 - (-1) < 3$):

```
static void datadep(int[][] a) {
  for (int i = 0; i < 100; i += 2)
    for (int j = 1; j < 100; j += 3)
       a[i][j] = a[i+1][j-1] * a[i+2][j+1];
}
```

Finally, if parallelization of the candidate parallel loop still has not been disabled, the compiler examines if any queries were recorded during the verification of array references. If user interaction is allowed, the compiler inquires the programmer whether the recorded queries will be satisfied for all possible invocations of the currently considered method. 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. Clearly, if user interaction is not enabled (the default), or if a query is not confirmed by the programmer, then worst-case assumptions have to be made, and the compiler resorts to disabling parallelization of the candidate parallel loop. If no queries have to be made, however, or if all queries are confirmed, then this loop can be actually executed in parallel.

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

```
static void query(int[][] a, int[] b, int x[], int n) {
  for (int i = 0; i <= n+100; i++)
     for (int j = 0; j < n-20; j++)
        b[i] += a[i][j] * x[j];
}
```

If the bytecode parallelization method discussed in this section is applied to the bytecode equivalent of this method, and if user interaction is enabled, then eventually the following inquiries will be made to the programmer:

```
Queries in method 'query([[I[I[I)V'
  Queries for the 2-dimensional array parameter '[[I' in local 0:
    * 1-dim: is range [0,101+n) valid, where n = parameter in local 3?
    * 2-dim: is range [0,n-20)  valid, where n = parameter in local 3?
  Queries for the 1-dimensional array parameter '[I' in local 1:
    * 1-dim: is range [0,101+n) valid, where n = parameter in local 3?
  Queries for the 1-dimensional array parameter '[I' in local 2:
    * 1-dim: is range [0,n-20)  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?
```

3.2.4 An Elaborate Example

Consider, as an elaborate example of the automatic detection of implicit loop parallelism, the following Java class method `par()`:

```
class Par {
  static int[][] par(int m, int n) {
    try {
      int[][] a = new int[m][n];
      int     mc = m, nc = n;
      for (int i = 0; i < mc; i++)  {
        int   k = i;
        int[] p = a[i], q = a[k];
        for (int j = 0; j < nc; j++)
           p[j] = q[j] + 1;
      }
      return a;
    }
    catch (Exception e) { return null; }
  }
}
```

The bytecode, flow graph $G = \langle V, E, v_0 \rangle$, and local variables usage for `par()` are shown in Figure 22. In this fragment, the following definitions can be distinguished:

	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}
local	0	1	2	3	4	5	6	7	8	9	9	5
address	$\perp$	$\perp$	6	8	10	13	20	26	32	35	52	62
par. type	I	I										

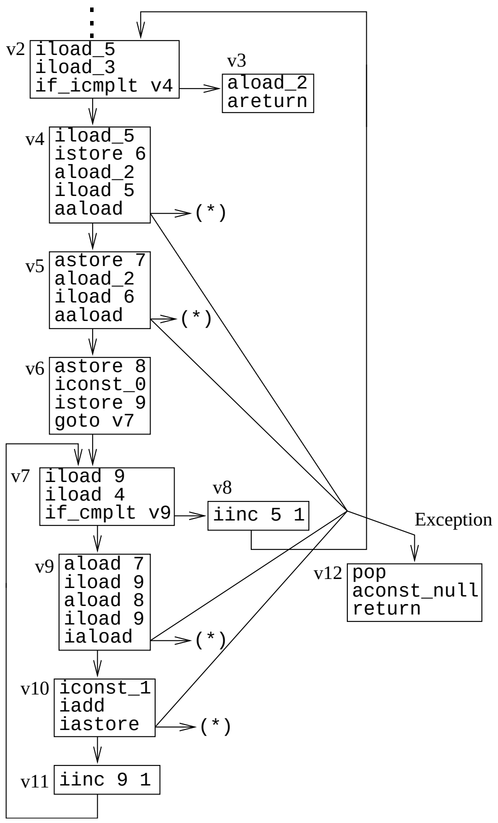
Local stack state information for some of the instructions in this method is shown below. Note that although the functionality of the loop is somewhat obscured by copy statements, our bytecode analysis eventually reveals that the method performs the operation '`a[i][j] += 1;`' for all elements of an `m`×`n` array, where `m` and `n` are parameters of the method:

code	stack states
`*** method 0x08 par(II)[[I 2->1`	
`----------`	
`[ 0]  0:  iload_0`	$\langle \top, \top, \top, \top \rangle$
`[ 1]  1:  iload_1`	$\langle \mathbf{var}(\{d_1\}), \top, \top, \top \rangle$
`*[ 2]  2:  multianewarray [[I`	$\langle \mathbf{var}(\{d_1\}), \mathbf{var}(\{d_2\}), \top, \top \rangle$
`----------`	
`[ 1]  6:  astore_2`	$\langle \mathbf{adecl}(2,2), \top, \top, \top \rangle$
`...`	
`----------`	
`[ 0] 40:  aload 7`	$\langle \bot, \bot, \bot, \bot \rangle$
`[ 1] 42:  iload 9`	$\langle \mathbf{adecl}(2,1), \bot, \bot, \bot \rangle$
`[ 2] 44:  aload 8`	$\langle \mathbf{adecl}(2,1), \mathbf{var}(\{d_{10}, d_{11}\}), \bot, \bot \rangle$
`[ 3] 46:  iload 9`	$\langle \mathbf{adecl}(2,1), \mathbf{var}(\{d_{10}, d_{11}\}), \mathbf{adecl}(2,1), \bot \rangle$
`*[ 4] 48:  iaload`	$\langle \mathbf{adecl}(2,1), \mathbf{var}(\{d_{10}, d_{11}\}), \mathbf{adecl}(2,1), \mathbf{var}(\{d_{10}, d_{11}\}) \rangle$
`----------`	
`[ 3] 49:  iconst_1`	$\langle \mathbf{adecl}(2,1), \mathbf{var}(\{d_{10}, d_{11}\}), \bot, \bot \rangle$
`[ 4] 50:  iadd`	$\langle \mathbf{adecl}(2,1), \mathbf{var}(\{d_{10}, d_{11}\}), \bot, \mathbf{con}(1) \rangle$
`*[ 3] 51:  iastore`	$\langle \mathbf{adecl}(2,1), \mathbf{var}(\{d_{10}, d_{11}\}), \bot, \bot \rangle$
`----------`	
`[ 0] 52:  iinc 9 1`	$\langle \bot, \bot, \bot, \bot \rangle$
`----------`	
`[ 0] 55:  iload 9`	$\langle \bot, \bot, \bot, \bot \rangle$
`[ 1] 57:  iload 4`	$\langle \mathbf{var}(\{d_{10}, d_{11}\}), \bot, \bot, \bot \rangle$
`[ 2] 59:  if_icmplt :40`	$\langle \mathbf{var}(\{d_{10}, d_{11}\}), \mathbf{var}(\{d_2\}), \bot, \bot \rangle$
`----------`	
`...`	

In the flow graph, the two back edges $(v_8, v_2) \in E$ and $(v_{11}, v_7) \in E$ define the natural loops $L_1 = \{v_2, v_4, \ldots, v_{10}\}$ and $L_2 = \{v_7, v_9, v_{10}, v_{11}\}$. Since both loops satisfy the constraints given in Section 3.1, L_1 and L_2 are trivial loops. Now, suppose that the compiler starts with examination of the outermost loop L_1. Clearly, the only abnormal loop-exits are due to <t>aload or <t>astore instructions. The loop can be executed in parallel with respect to local scalars, since there are no loop-carried flow dependences caused by locals, and all locals are dead on exit of the loop (cf. Section 3.2.2). Hence, subsequently, array analysis is performed.

Consider, for instance, the `iaload` instruction at address $a = 48$ (corresponding to reference '`q[j]`' at source code level). Here, we have $r = \mathbf{adecl}(2,1)$ and $s = \mathbf{var}(\{d_{10}, d_{11}\})$ for the array reference and subscript expression, respectively. If `verify_ref`(a, r, s) is called, the lower bound $\mathbf{con}(0)$ and strict upper bound $\mathbf{var}(\{d_2\})$ result for s, since the subscript expression corresponds to the inner trivial loop with index 9 as local. Because $r = \mathbf{adecl}(2, n)$, where $n = 1$, represents a reference to the last dimension of the two-dimensional array declared at address 2, and because $s_{i.\texttt{sp}-n} = \mathbf{var}(\{d_2\})$ holds for the stack state $\overline{s}$ associated with the declaration instruction i at 2, the compiler concludes that the abnormal loop-exit at address 48 cannot be taken at run-time.

After all other <t>aload and <t>astore instructions are handled similarly, access tuple $\overline{x} = \langle [0,0], \bot \rangle$ is associated with the array declared at address 2. This indicates that references to this array do not cause any data dependences that are carried by the outer trivial loop, which implies that the `i`-loop can be executed in parallel.




```
*** method 0x08 par(II)[[I 2->1
[0,73) -> 73 java.lang.Exception
----------
 [ 0]  0:  iload_0
 [ 1]  1:  iload_1
*[ 2]  2:  multianewarray [[I
----------
 [ 1]  6:  astore_2
 [ 0]  7:  iload_0
 [ 1]  8:  istore_3
 [ 0]  9:  iload_1
 [ 1] 10:  istore 4
 [ 0] 12:  iconst_0
 [ 1] 13:  istore 5
 [ 0] 15:  goto 65
----------
 [ 0] 18:  iload 5
 [ 1] 20:  istore 6
 [ 0] 22:  aload_2
 [ 1] 23:  iload 5
*[ 2] 25:  aaload
----------
 [ 1] 26:  astore 7
 [ 0] 28:  aload_2
 [ 1] 29:  iload 6
*[ 2] 31:  aaload
----------
 [ 1] 32:  astore 8
 [ 0] 34:  iconst_0
 [ 1] 35:  istore 9
 [ 0] 37:  goto 55
----------
 [ 0] 40:  aload 7
 [ 1] 42:  iload 9
 [ 2] 44:  aload 8
 [ 3] 46:  iload 9
*[ 4] 48:  iaload
----------
 [ 3] 49:  iconst_1
 [ 4] 50:  iadd
*[ 3] 51:  iastore
----------
 [ 0] 52:  iinc 9 1
----------
 [ 0] 55:  iload 9
 [ 1] 57:  iload 4
 [ 2] 59:  if_icmplt 40
----------
 [ 0] 62:  iinc 5 1
----------
 [ 0] 65:  iload 5
 [ 1] 67:  iload_3
 [ 2] 68:  if_icmplt 18
----------
 [ 0] 71:  aload_2
 [ 1] 72:  areturn
----------
 [ 1] 73:  pop
 [ 0] 74:  aconst_null
 [ 1] 75:  areturn
----------
```

	Locals
0:	m
1:	n
2:	a
3:	mc
4:	nc
5:	i
6:	k
7:	p
8:	q
9:	j

Figure 22: Flow graph and bytecode of class method `par()`

3.3 Exploitation of Implicit Loop Parallelism

If a trivial loop $L \subseteq V$ in a method with flow graph $G = \langle V, E, v_0 \rangle$ satisfies all the constraints of the previous sections, implicit loop parallelism is made explicit using the JVM multi-threading mechanism. The actual parallelization resembles the Java source code transformations described in [6], although now all transformations are directly applied at bytecode level. Because there is usually little advantage of having nested parallel loops, we disable the parallelization of candidate parallel loops within another candidate parallel loop. Moreover, because parallelization of outer loops is generally preferred over the parallelization of inner loops, the compiler attempts to parallelize the natural loops of each method in decreasing order of size, where the size of a loop consists of the number of enclosed basic blocks.

In this section, all new identifiers that may conflict with other identifiers are denoted with a suffix '`_x`'. In reality, however, an appropriate suffix must be generated by the compiler.

3.3.1 Modification of the Original Loop

Given a stride-p candidate parallel loop $L \subseteq V$ defined by a back edge $(g, h) \in E$ with loop index `i` in a method with l locals, the compiler first determines the set $W = \{l_1, \ldots, l_p\}$ of locals (including the loop-index) such that an outer-loop definition of local $l_j \in [0..l)$ reaches an inner-loop use of l_j. Furthermore, for each l_j, the corresponding type t_j is determined by examination of the reaching definitions. For a `long` or `double` definition of locals l and $l+1$, only l is included in W.

Parallel execution of the loop in class '`MyClass`', is implemented by letting T different threads invoke an appropriate `run()`-method in a new auxiliary class '`MyClass_Worker_x`' for different subsets of iterations.[5] Consequently, locals in the set W must be passed to this method (constraint (II) of Section 3.2.2 avoids the need to pass locals back to the original method after execution of the parallel loop). For this purpose, the following bytecode fragment is added to the method in which the parallel loop appears. Here, `PARS` denote the concatenation of the parameter descriptors for each type t_j, label `OldExit` denotes the address of the first instruction in the basic block $e \notin L$ of the normal loop exit $(h, e) \in E$, and '<t_j>`load`' denotes the appropriate load instruction (e.g. `aload` for a reference type t_j):

```
NewCode:
         new MyClass_Worker_x                          \
         dup                                           |
         <t1>load l1                                   |
         ...                                           }  T ×
         <tp>load lp                                   |
         invokespecial MyClass_Worker_x.<init> (PARS)V |
         iinc i p                                      /

         invokevirtual java.lang.Thread.join ()V       }  T ×

         goto OldExit


Handler: pop
         goto OldExit
```

In this code, T new workers are constructed with the appropriate parameters, where each worker receives a subsequent iteration. As discussed in the next section, on construction, each worker will start a new thread to execute iterations of the original loop with a new stride '$p \times T$', thereby effectively implementing a cyclic scheduling policy for the parallel loop. In the code above, the workers are eventually joined again by invoking T times method `join()`, using the references that are still on the operand stack.

[5] Since '`MyClass`' appears as fully qualified class name in the class file format, the auxiliary worker class is automatically placed in the same package if a similar prefix is used in the worker class name and the corresponding class file is created in the directory of the original class file.

Finally, the compiler changes the maximum operand stack size of the method in case the maximum stack size in this new fragment exceeds the old size, replaces the first instruction in the basic block $d \in V$ of the loop by the instruction `goto NewCode`, and adds an exception handler at `Handler` for '`java.lang.InterruptedException`', covering the region from label `NewCode` up to label `Handler`. All other inner-loop bytecode instructions are replaced by `nop` (these instructions are unreachable in the resulting code), and attributes that are nested within the Code attribute [26, ch4] are stripped from the resulting class file, because the meaning of these attributes may have become obsoleted.

3.3.2 Construction of a new Loop Worker Class

In the next step, the compiler generates a class file for the auxiliary class '`MyClass_Worker_x`' that corresponds to the following Java definition (but keep in mind that this class is directly generated in the class file format [26]):

```
class MyClas_Worker_x extends java.lang.Thread {
  private t1 loc_1;
  ...
  private tp loc_p;

  public MyClass_Worker_x(t1 loc_1, ..., tp loc_p) {
    this.loc_1 = loc_1;
    ...
    this.loc_p = loc_p;
    start();
  }

  public void run() {
    ...
  }
}
```

In the constructor, the values of the p locals that are passed as parameter are temporarily stored in p fields of appropriate type, as described by field descriptors [26, ch4] in the class file. Moreover, a new thread is started by calling `start()`.

The `run()`-method in this class has the form shown below. First, the values of all locals are restored, where we assume that the locals in W are sorted in decreasing order, so that in case local 0 appears in the set W, the implicit `this` parameter is destroyed last. Subsequently, control is transferred to a modified version of the original loop:

```
              aload_0
              getfield MyClass_Worker_x.loc_1
              <tp>store 1
              ...
              aload_0
              getfield MyClass_Worker_x.loc_p
              <t1>store p

              goto LoopEntry

LoopEntry:
              ... modified loop ...

NewLoopExit: return
```

Modifications to the original loop consists of replacing the old loop exit $(h, e) \in E$ by a transfer of control to `NewLoopExit` (either an implicit fall-through or an explicit transfer in an `if_icmp<cond>`), and replacing the original '`iinc i` p' instruction by '`iinc i` $p \times T$', where T denotes the number of threads used for parallel execution. As stated earlier, this effectively implements a cyclic scheduling policy. In a future implementation, however, more advanced scheduling policies (see e.g. [36, ch4][45, p73-74][46, p387-392] [41, 42][47, 296-298]) with better load balancing or data locality properties could be incorporated.

If an instruction within the loop (indirectly) refers to entries in the constant pool of the original class file [26], these entries must be copied into the constant pool of the new auxiliary class.

For example, consider the following class (placed in the default package):

```
class Pool {
  static void pool(double dc) {
    double[][] d = new double[1000][1000];
    for (int i = 0; i < 1000; i++)
      for (int j = 0; j < 1000; j++)
         d[i][j] = 42.42d * java.lang.Math.sin(dc);
  }
}
```

The constant pool in the corresponding class file and the bytecode for the method `pool` are shown below. If loop parallelization is applied to the trivial loop that corresponds to the `i`-loop, then some of the constant pool entries must be copied into the constant pool of the auxilary class. For instance, the `ldc2_w` instruction at 27 refers to the `CONSTANT_Double` entry 7 (making entry 8 invalid). Moreover, the method invocation instruction at 31 refers to the `CONSTANT_Methodref` entry 5 and, hence, indirect to entries 1, 10, 11, 24, and 25:

```
--> constant_pool[ 1]: <Class> java/lang/Math
    constant_pool[ 2]: <Class> java/lang/Object
    constant_pool[ 3]: <Class> [[D
    constant_pool[ 4]: <Class> Pool
--> constant_pool[ 5]: <Class> java/lang/Math.sin (D)D
    constant_pool[ 6]: <Class> java/lang/Object.<init> ()V
--> constant_pool[ 7]: <Double> 42.42
    constant_pool[ 9]: <init> ()V
--> constant_pool[10]: sin (D)D
--> constant_pool[11]: (D)D
    constant_pool[12]: pool
    constant_pool[13]: ConstantValue
    constant_pool[14]: Exceptions
    constant_pool[15]: Pool.java
    constant_pool[16]: LineNumberTable
    constant_pool[17]: SourceFile
    constant_pool[18]: LocalVariables
    constant_pool[19]: Code
    constant_pool[20]: java/lang/Object
    constant_pool[21]: [[D
    constant_pool[22]: Pool
    constant_pool[23]: <init>
--> constant_pool[24]: sin
--> constant_pool[25]: java/lang/Math
    constant_pool[26]: (D)V
    constant_pool[27]: ()V
```

```
*** method 0x08 pool(D)V 2->0
 ----------
 [ 0]  0:  sipush 1000
 [ 1]  3:  sipush 1000
*[ 2]  6:  multianewarray #3 <Class> [[D
 ----------
 [ 1] 10:  astore_2
 [ 0] 11:  iconst_0
 [ 1] 12:  istore_3
 [ 0] 13:  goto 50
 ----------
 [ 0] 16:  iconst_0
 [ 1] 17:  istore 4
 [ 0] 19:  goto 39
 ----------
 [ 0] 22:  aload_2
 [ 1] 23:  iload_3
*[ 2] 24:  aaload
 ----------
 [ 1] 25:  iload 4
 [ 2] 27:  ldc2_w #7
           <Double> 0x404535c28f5c28f6
 [ 4] 30:  dload_0
 ----------
*[ 6] 31:  invokestatic #5
           <Class> java/lang/Math.sin (D)D
 ----------
 [ 6] 34:  dmul
*[ 4] 35:  dastore
 ----------
 [ 0] 36:  iinc 4 1
 ----------
 [ 0] 39:  iload 4
 [ 1] 41:  sipush 1000
 [ 2] 44:  if_icmplt 22
 ----------
 [ 0] 47:  iinc 3 1
 ----------
 [ 0] 50:  iload_3
 [ 1] 51:  sipush 1000
 [ 2] 54:  if_icmplt 16
 ----------
 [ 0] 57:  return
 ----------
```

If constant pool entries must be copied into the constant pool of the auxiliary class, then currently `javab` copies the whole old constant pool into the new class file and simply obsoletes all entries that are not required. This approach avoids the necessity to change references to the constant pool within the loop. Thereafter, new entries are added to the end of the old constant pool. In a future implementation, however, a compression should be applied to the new constant pool to reduce the size of the resulting class file.

3.3.3 An Elaborate Example (Continued)

Consider, for example, the Java class method `par()` of the previous section again. After the compiler has detected that the trivial loop $L = \{v_2, \ldots, v_{10}\}$ can be executed in parallel, the set $W = \{2,3,4,5\}$ is constructed, where local 2 contains a two-dimensional integer array, and the other locals integer scalars. Thereafter, the following code is added to the `par()`-method in case $T = 2$. In addition, the instruction at address 65 is replaced by `goto 76`, and an exception handler starting at 121 for code region `[76,121)` and exception '`java.lang.InterruptedException`' is added to the exception table of this method:

```
  ...
  ----------
  [ 1] 73:  pop
  [ 0] 74:  aconst_null
  [ 1] 75:  areturn
  ----------
 *[ 0] 76:  new Par_Worker_par_0
  ----------
  [ 1] 79:  dup
  [ 2] 80:  aload 2
  [ 3] 82:  iload 3
  [ 4] 84:  iload 4
  [ 5] 86:  iload 5
  ----------
 *[ 6] 88:  invokespecial Par_Worker_par_0.<init> ([[IIII)V
  ----------
  [ 1] 91:  iinc 5 1
 *[ 1] 94:  new Class Par_Worker_par_0
  ----------
  [ 2] 97:  dup
  [ 3] 98:  aload 2
  [ 4]100:  iload 3
  [ 5]102:  iload 4
  [ 6]104:  iload 5
  ----------
 *[ 7]106:  invokespecial Par_Worker_par_0.<init> ([[IIII)V
  ----------
  [ 2]109:  iinc 5 1
  ----------
 *[ 2]112:  invokevirtual java.lang.Thread.join ()V
  ----------
 *[ 1]115:  invokevirtual java.lang.Thread.join ()V
  ----------
  [ 0]118:  goto 71
  ----------
  [ 1]121:  pop
  [ 0]122:  goto 71
  ----------
```

Subsequently, the compiler constructs a new class file that defines the `Par_par_worker` class with fields `local_2` (with field descriptor '`[[I`'), `local_3`, `local_2`, and `local_3` (with field descriptor '`I`'). The bytecode for the constructor and `run()`-method of this class are shown below:

```
*** method 0x01 <init>([[IIII)V 5->0
----------
 [ 0]  0:  aload_0
----------
*[ 1]  1:  invokespecial java.lang.Thread.<init> ()V
----------
 [ 0]  4:  aload_0
 [ 1]  5:  aload 1
*[ 2]  7:  putfield Par_Worker_par.loc_2 [[I
----------
 [ 0] 10:  aload_0
 [ 1] 11:  iload 2
*[ 2] 13:  putfield Par_Worker_par.loc_3 I
----------
 [ 0] 16:  aload_0
 [ 1] 17:  iload 3
*[ 2] 19:  putfield Par_Worker_par.loc_4 I
----------
 [ 0] 22:  aload_0
 [ 1] 23:  iload 4
*[ 2] 25:  putfield Par_Worker_par.loc_5 I
----------
 [ 0] 28:  aload_0
----------
*[ 1] 29:  invokevirtual java.lang.Thread.start ()V
----------
 [ 0] 32:  return
----------
```

```
*** method 0x01 run()V 1->0
----------
 [ 0]  0:  aload_0
*[ 1]  1:  getfield Par_Worker_par.loc_5 I
----------
 [ 1]  4:  istore 5
 [ 0]  6:  aload_0
*[ 1]  7:  getfield Par_Worker_par.loc_4 I
----------
 [ 1] 10:  istore 4
 [ 0] 12:  aload_0
*[ 1] 13:  getfield Par_Worker_par.loc_3 I
----------
 [ 1] 16:  istore 3
 [ 0] 18:  aload_0
*[ 1] 19:  getfield Par_Worker_par.loc_2 [[I
----------
 [ 1] 22:  astore 2
 [ 0] 24:  goto 74      ; goto NewLoopEntry
----------
 [ 0] 27:  iload 5
 [ 1] 29:  istore 6
 [ 0] 31:  aload_2
 [ 1] 32:  iload 5
*[ 2] 34:  aaload
----------
 [ 1] 35:  astore 7
 [ 0] 37:  aload_2
 [ 1] 38:  iload 6
*[ 2] 40:  aaload
----------
 [ 1] 41:  astore 8
 [ 0] 43:  iconst_0
 [ 1] 44:  istore 9
 [ 0] 46:  goto 64
----------
 [ 0] 49:  aload 7
 [ 1] 51:  iload 9
 [ 2] 53:  aload 8
 [ 3] 55:  iload 9
*[ 4] 57:  iaload
----------
 [ 3] 58:  iconst_1
 [ 4] 59:  iadd
*[ 3] 60:  iastore
----------
 [ 0] 61:  iinc 9 1
----------
 [ 0] 64:  iload 9
 [ 1] 66:  iload 4
 [ 2] 68:  if_icmplt 49
----------
 [ 0] 71:  iinc 5 2
----------
 [ 0] 74:  iload 5
 [ 1] 76:  iload_3
 [ 2] 77:  if_icmplt 27 ; fall-through to return
----------
 [ 0] 80:  return
----------
```

4 Initial Experiments

The techniques presented in this paper have been actually implemented in a prototype bytecode parallelization tool `javab` which is made freely available (for details, see the end of this paper). In this section, we present some initial experiments that have been conducted with this prototype.

4.1 Timings

Consider, for instance, the following Java class method `mat_mat()`:

```
static void mat_mat(double[][] a, double b[][], double c[][], int n) {
  for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)  {
        c[i][j] = 0.0d;
        for (int k = 0; k < n; k++)
          c[i][j] += a[i][k] * b[k][j];
     }
}
```

When our prototype is applied to the bytecode equivalent of this fragment, the compiler first inquires the programmer whether array parameters `a`, `b`, and `c` will always refer to matrices with at least size $\mathtt{n} \times \mathtt{n}$, and whether the storage of array parameters `a` or `b` will not overlap with the storage of array parameter `c`. If these queries are affirmed, the compiler proceeds with parallelization of the trivial loop that corresponds to the `i`-loop.

In Figures 23, we show the serial execution times T_s and the parallel execution times T_p for varying values of `n`, with and without JIT compilation on an IBM RS/6000 G30 with four PowerPC 604 processors using the AIX4.2 JDK1.0.2d programming environment. The bytecode is executed using 'java -noasyncgc' and the parallel execution of threads enabled. In Figure 24, the corresponding speedup $S = T_p/T_s$ is shown. With JIT compilation enabled, parallelization becomes useful for this particular loop and target platform when `n` exceeds 30, and an efficiency over 85% is obtained.

4.2 Statistics

In the following table, we present the results of applying `javab` to the class files of a number of packages: (i) the complete Java 1.0.2API, (ii) Pendragon Software's CaffeineMark(tm) version 2.5 [14], (iii) a wavelet transformation package written by David Wanqian Liu [27], and (iv) a Java implementation of some routines of BLAS and LINPACK [44] written by Steve Verrill. In addition, because the last package contains a number of test programs where the user must supply a value $n \leq 100$ for the size of test matrices, we also consider the results of applying `javab` to the class files that result when we compile the package with a fixed size for n.

In the table, we present the total number of class files and natural loops. In addition, we show how many of the natural loops are trivial loops, and divide the trivial loops into three categories: serial trivial loops, parallel trivial loops, and trivial loops that appear within a parallel trivial loop.

	Class Files	Loops Natural	Trivial	Serial	Parallel	Nested	
Java 1.0.2 API	224	308	53	50	3	0	
CaffeineMark 2.5	17	99	54	47	3	4	
Wavelets	21	83	65	59	4	2	
				41	15	9	(queries)
BLAS/LINPACK	23	368	322	318	4	0	
				306	14	2	(queries)
BLAS/LINPACK (fixed n)	23	368	322	290	17	15	
				278	27	17	(queries)

From the table it becomes clear that a reasonable amount of natural loops in bytecode programs can be classified as trivial loops. Furthermore, despite the fact that are analysis method have been

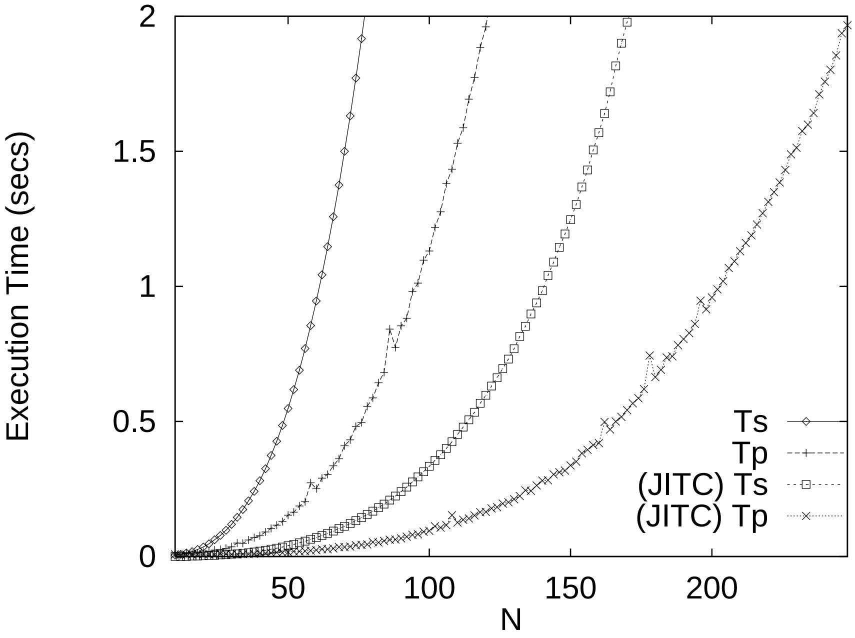


Figure 23: Execution time of class method mat_mat()

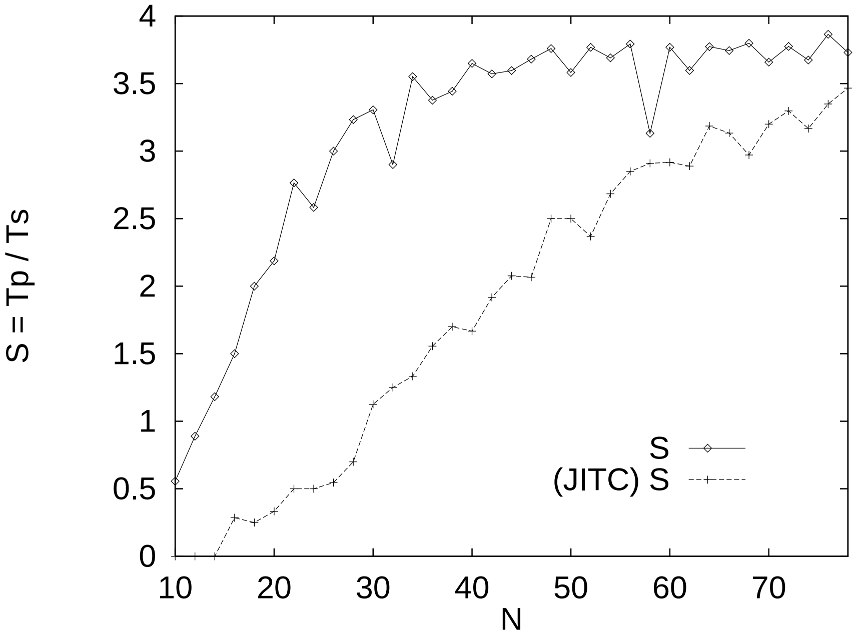


Figure 24: Speedup of class method mat_mat()

kept relatively simple, already a reasonable number of parallel loops can be detected automatically by our tool (whether any speedup may be actually expected from executing these loops in parallel heavily depends on the amount of work done in the loops and the startup overhead of threads in the JVM implementation). In some cases, this number can be increased by means of user interaction, which implies that more loops will be detected automatically if inter-procedural analysis is incorporated in environments where complete bytecode programs are available (such as the JIT parallelization approach we have alluded to earlier). Because our prototype can inform the user about the reasons why a particular loop cannot be parallelized, we hope that more experiments will provide us with insights in the kind of improvements that are required to increase the number of parallel loops that can be automatically detected.

The three parallel loops in the Java API appear in the class initialization method '<clinit>' of the 'java.lang.Character' class. One of these parallel trivial loops is illustrated below:

code	stack states
*** method 0x08 <clinit>()V 0->0	
----------	
[0] 0: sipush 256	$\langle \top, \top, \top, \top, \top, \top \rangle$
*[1] 3: newarray char	$\langle \mathbf{con}(256), \top, \top, \top, \top, \top \rangle$
----------	
[1] 5: astore_0	$\langle \mathbf{adecl}(3,1), \top, \top, \top, \top, \top \rangle$
[0] 6: sipush 1 0	$\langle \bot, \top, \top, \top, \top, \top \rangle$
*[1] 9: newarray char	$\langle \mathbf{con}(256), \top, \top, \top, \top, \top \rangle$
----------	
[1] 11: astore_1	$\langle \mathbf{adecl}(9,1), \top, \top, \top, \top, \top \rangle$
[0] 12: iconst_0	$\langle \bot, \top, \top, \top, \top, \top \rangle$
[1] 13: istore_2	$\langle \mathbf{con}(0), \top, \top, \top, \top, \top \rangle$
[0] 14: goto 29	$\langle \bot, \top, \top, \top, \top, \top \rangle$
----------	
[0] 17: aload_0	$\langle \bot, \bot, \bot, \bot, \bot, \bot \rangle$
[1] 18: iload_2	$\langle \mathbf{adecl}(3,1), \bot, \bot, \bot, \bot, \bot \rangle$
[2] 19: aload_1	$\langle \mathbf{adecl}(3,1), \mathbf{var}(\{d_3, d_4\}), \bot, \bot, \bot, \bot \rangle$
[3] 20: iload_2	$\langle \mathbf{adecl}(3,1), \mathbf{var}(\{d_3, d_4\}), \mathbf{adecl}(9,1), \bot, \bot, \bot \rangle$
[4] 21: iload_2	$\langle \mathbf{adecl}(3,1), \mathbf{var}(\{d_3, d_4\}), \mathbf{adecl}(9,1), \mathbf{var}(\{d_3, d_4\}), \bot, \bot \rangle$
[5] 22: i2c	$\langle \mathbf{adecl}(3,1), \mathbf{var}(\{d_3, d_4\}), \mathbf{adecl}(9,1), \mathbf{var}(\{d_3, d_4\}), \mathbf{var}(\{d_3, d_4\}), \bot \rangle$
[5] 23: dup_x2	$\langle \mathbf{adecl}(3,1), \mathbf{var}(\{d_3, d_4\}), \mathbf{adecl}(9,1), \mathbf{var}(\{d_3, d_4\}), \bot, \bot \rangle$
*[6] 24: castore	$\langle \mathbf{adecl}(3,1), \mathbf{var}(\{d_3, d_4\}), \bot, \mathbf{adecl}(9,1), \mathbf{var}(\{d_3, d_4\}), \bot \rangle$
----------	
*[3] 25: castore	$\langle \mathbf{adecl}(3,1), \mathbf{var}(\{d_3, d_4\}), \bot, \bot, \bot, \bot \rangle$
----------	
[0] 26: iinc 2 1	$\langle \bot, \bot, \bot, \bot, \bot, \bot \rangle$
----------	
[0] 29: iload_2	$\langle \bot, \bot, \bot, \bot, \bot, \bot \rangle$
[1] 30: sipush 256	$\langle \mathbf{var}(\{d_3, d_4\}), \bot, \bot, \bot, \bot, \bot \rangle$
[2] 33: if_icmplt 17	$\langle \mathbf{var}(\{d_3, d_4\}), \mathbf{con}(256), \bot, \bot, \bot, \bot \rangle$

...	

All three loops in Pendragon Software's CaffeineMark(tm) version 2.5 appear in the method fpmark() that is used to test floating point performance. One of these loops is a single loop, and the other two loops are loops that essentially perform the following operation:

```
for (int i = 0; i < 50; i++) {
  c[0][i] = 0.0; c[1][i] = 0.0; c[2][i] = 0.0;
  for (int j = 0; j < 3; j++)
    for (int k = 0; k < 3; k++)
        c[j][i] += a[k]j] * c[j][i];
}
```

Unfortunately, if loop parallelization is applied to the triple loops (by default, parallelization of single loops in javab is disabled), the Floating Point Score drops from 279 to 112 on an IBM RS/6000 G30 with four PowerPC 604 processors using the AIX4.2 JDK1.1 programming environment (JIT compilation is not yet supported).[6] In this case, the startup-time of the parallel loop is too high in comparison with the time required to execute the $50 \times 3 \times 3$ iterations. On the same machine the execution time of a matrix times vector in the BLAS/LINPACK package, for instance, drops from 0.51 sec. to 0.17 sec. for a matrix of size $n = 500$, so that a speedup of 3 is obtained.

[6]These tests were performed without independent verification by Pendragon Software and Pendragon Software makes no representations or warranties as to the result of the test.

5 Conclusions

In this paper, we have presented techniques for the automatic detection and automatic exploitation of implicit loop parallelism in bytecode. This work extends our earlier work on Java source code restructuring [6] in two ways: first, all transformations are now directly applied at bytecode level and, second, loop parallelism is now also detected automatically (rather than relying on annotations in the Java source code). Because implicit loop parallelism is made explicit by means of the JVM multi-threading mechanism, the parallelization can be done independently from the source program and platforms from which the bytecode was obtained and 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. The techniques have been implemented in a prototype bytecode parallelization tool, and some initial experiments with automatic bytecode parallelization have been included. More elaborate experiments are planned in a follow-up paper.

Our prototype focuses on the parallelization of loops that operate on arrays, and this parallelization is done by means of an off-line bytecode to bytecode conversion. Because exceptions have to be dealt with precisely [18, 26], the parallelization of loops is only applied to regions of code for which the compiler can prove that run-time exceptions cannot be thrown. A potential change in semantics with respect to JVM errors and (possibly) linking-exceptions is allowed, however. Alternatively, we could have generated tests that at run-time determine whether exceptions may be thrown in a loop and, based on the outcome of these tests, decide between serial or parallel execution of the loop. If the start-up time of a parallel loop is substantial, a similar multi-version approach can be used to make the decision between serial or parallel execution of the loop dependent on the actual number of iterations seen at run-time. In the current implementation, cyclic scheduling is used for all parallel loops, but more advanced scheduling policies could be incorporated in future versions. Another possible future extension is the incorporation of the `arraylength` instruction in the analysis of arrays.

Compile-time has been kept limited by letting the compiler rely on less accurate but generally also less expensive analysis. For example, operand stack states are only traced partially, rather than doing a more accurate conversion of stack usage into expressions trees, as is discussed in, for example, the paper [7]. Although this implies that currently only relatively simple loop bounds and subscripts can be dealt with, preliminary experiments indicate that this simple analysis already suffices to handle a reasonable number of loops in numerical programs. In addition, although we have presented stack state analysis specifically in the context of loop parallelization, this simple stack state analysis may also provide a source for a vast amount of other compiler optimizations. Simple data dependence tests for arrays and scalars have been used, rather than advanced data dependence analysis, and parallelization of a loop in which field modifications occur is disabled, rather than using e.g. the side-effect analysis presented in [12]. However, although using more advanced analysis will increase the opportunities for loop parallelization, 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 or JIT compilation of the bytecode.

Finally, to allow the transformations to be applied to a single class file, our prototype resorts to user interaction as soon as inter-procedural information is required. Although this currently may place a heavy burden on the user, especially if source code is not available, these problems will be alleviated by the incorporation of inter-procedural analysis in environments where complete bytecode programs are available. Hereby, techniques that incrementally update inter-procedural information after changes of parts of a program [13] will be very useful.

Obtaining Java Research Tools

Documentation and the complete source of the prototype bytecode parallelization tool `javab`, together with a prototype Java restructuring compiler `javar` described in [6], are made available for education, research, and non-profit purposes at: `http://www.extreme.indiana.edu/hpjava/`

References

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

[2] J.R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

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

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

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

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

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

[8] David Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Department of Computer Science, Rice University, 1987.

[9] 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.

[10] Michal Cierniak and Wei Li. Just-in-time optimizations for high-performance Java programs. *To Appear Concurrency, Practice and Experience*, 1997.

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

[12] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[13] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8:491–523, 1986.

[14] Pendragon Software Corporation. *Pendragon Software's CaffeineMark(tm) version 2.5 – Java Benchmark.* This Java benchmark is made available at `http://www.webfayre.com/cm.html`.

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

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

[17] 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.

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

[19] Jonathan C. Hardwick and Jay Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, 1996.

[20] Joe Hummel, Ana Azevedo, David Kolson, and Alex Nicolau. Annotating the Java bytecodes in support of optimization. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[21] Donald E. Knuth. *An Empirical Study of FORTRAN Programs.* US Department of Commerce, Stanford University, 1971.

[22] Andreas Krall and Reinhard Grafl. A Java just-in-time compiler that transcends Java-VM's 32 bit barrier. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[23] David J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, New York, 1978. Volume 1.

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

[25] Zhiyuan Li and Walid Abu-Sufah. On reducing data synchronization in multi-processed loops. *IEEE Transactions on Computers*, C-36:105–109, 1987.

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

[27] David Wanqian Liu. *Wavelet Transformation Package.* This package is made available at `http://reality.sgi.com/davidliu/`.

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

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

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

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

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

[33] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29:763–776, 1980.

[34] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, 1986.

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

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

[37] Constantine D. Polychronopoulos, David J. Kuck, and David A. Padua. Execution of parallel loops on parallel processor systems. In *Proceedings of the International Conference on Parallel Processing*, pages 519–527, 1986.

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

[39] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[40] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5:216–226, 1979.

[41] Rizos Sakellarious and John R. Gurd. Compile-time minimisation of load imbalance in loop nests. In *Proceedings of the International Conference on Supercomputing*, Vienna, 1997.

[42] Rizos Sakellarious and John R. Gurd. Compile-time schemes for mapping loop parallelism. In A. Bakkers, editor, *Parallel Programming and Java*, pages 252–260. IOS Press, 1997. (WoTUG-20).

[43] Shawn Silverman. *D-Java: a Class File Disassembler.* This JVM bytecode disassembler is made available at `http://home.cc.umanitoba.ca/~umsilve1/djava/`.

[44] Steve Verrill. *Linear Algebra for Statistics Java Package.* This numerical package is made available at `http://www1.fpl.fs.fed.us/linear_algebra.html`.

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

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

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