

A Strategy for Exploiting Implicit Loop Parallelism in Java Programs

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

In this paper, we explore a strategy that can be used by a source to source restructuring compiler to exploit implicit loop parallelism in Java programs. First, the compiler must identify the parallel loops in a program. Thereafter, the compiler explicitly expresses this parallelism in the transformed program using the multi-threading mechanism of Java. Finally, after a single compilation of the transformed program into Java byte-code, speedup can be obtained on any platform on which the Java byte-code interpreter supports actual concurrent execution of threads, whereas threads only induce a slight overhead for serial execution. In addition, this approach can enable a compiler to explicitly express the scheduling policy of each parallel loop in the program.

1 Introduction

One of the important design goals of the Java programming language was to provide a truly *architectural neutral* language, so that Java applications could run on any platform. To achieve this objective, a Java program is compiled into architectural neutral instructions (byte-code) of an abstract machine (the Java Virtual Machine), rather than compiled into native machine code. In this manner, a compiled Java program will run on any platform on which a Java byte-code interpreter is available.

This idea has already proven to be successful in the past. Some of first implementers of Pascal compilers [6], for example, used so-called p-code as target language, forming the language of an abstract ma-

chine (the Virtual Stack Machine). The compiler itself, translating Pascal program into p-code, was entirely written in p-code. Hence, to install Pascal on a particular platform, only a simple p-code interpreter had to be written. Likewise, the Java compiler itself is written in Java, and the byte-code version of this compiler can run immediately on any platform that provides a Java byte-code interpreter. Because many Java utilities are written in Java as well, a complete Java environment becomes available after this relatively simple byte-code interpreter together with some native methods have been implemented on a particular platform. Without any doubt, this has contributed to the rapid spread of the language.

Although interpretation of byte-code is, in general, faster than the interpretation of high level languages, it is definitely slower than executing native machine code. Clearly, for particular interactive applications this is not a major drawback, but still other situations remain in which performance is critical. In these cases, so-called ‘just-in-time compilation’ may be useful, where *at run-time* byte-code is compiled into native machine code. In fact, Sun claims that with this approach the performance of Java can come close to the performance of compiled languages. However, because the demand for more computing power is likely to remain, other means to speedup Java programs have to be found.

In this paper, we explore a strategy that can be used by a source restructuring compiler to exploit implicit loop parallelism in Java programs. While not yet incorporated in a working compiler, we describe the required transformations and illustrate the potential speedup with a series of experiments.

Within our framework, the compiler must first identify the parallel loops in a Java program, either by means of advanced data dependence analysis, or simply by means of user annotations. Thereafter, the compiler uses the multi-threading mechanism of the Java programming language (see e.g. [2, 12, 20, 21]) to explicitly express these parallel loops in the transformed program. In this manner, after a single compilation of the transformed program into Java byte-code, speedup can be obtained on any platform on which the Java byte-code interpreter supports actual concurrent execution of threads. Furthermore, threads only induce a slight overhead for serial execution. Finally, this approach enables the compiler to express the scheduling policy of each parallel loop in the program. However, since threads are lightweight processes sharing one address-space, the actual concurrent execution of threads is typically only supported on shared-address-space architectures [15]. Hence, the focus of this paper is to obtain speedup on such architectures. Future research, however, will focus on letting a restructuring compiler use the networking capabilities of Java in a message-passing like manner to take advantage of computing power that is available over a network.

The rest of this paper is organized as follows. First, in section 2, we give some preliminaries related to exploiting implicit loop parallelism. Thereafter, in section 3, we discuss how a compiler can explicitly implement parallel loops in Java using multi-threading. In section 4, we present the results of experiments. Finally, we state conclusions in section 5.

2 Preliminaries

Some preliminaries related to exploiting implicit loop parallelism and Java restructuring are given.

2.1 Parallel Loops

If all iterations of a loop are independent, i.e. *if no data dependence is carried by the loop*, then this loop can be converted into a truly parallel loop, frequently referred to as a DO-ALL-loop. Because data dependence theory, data dependence analysis and loop parallelization are discussed extensively in the literature (see e.g. [3, 4, 5, 7, 10, 14, 23, 22, 25, 26, 27, 28, 29]), we do not further elaborate on these issues in this paper.

Instead, we simply assume that our restructuring compiler has some (conservative) mechanism to determine whether data dependences are carried by a particular loop.

Even if data dependences are carried by a loop, some parallelism may result from executing the loop as a DO-ACROSS-loop, where a partial execution of some (parts) of the iterations is enforced using synchronization to satisfy the data dependences. In [8, 9], this synchronization is modeled under the assumption that processors operate synchronously by using a particular delay $d \geq 0$ between consecutive iterations of the DO-ACROSS-loop: the i th iteration is executed after a delay of $(i - 1) \cdot d$. Alternatively, synchronization can be enforced using the primitives **testset/test** (or **advance/await**) [17, ch6][18, 19][28, p393-395], which can be implemented with a scalar synchronization variable [27, p84-86].

A more general form of synchronization in a DO-ACROSS-loop is provided by **random synchronization** with primitives **post/wait** [27, p75-83][29, p289-295]. If data dependences are carried by the loop, a non-blocking **post**-statement that sets a bit corresponding to the current iteration is placed directly *after* the source statement of each *static* data dependence. Directly *before* the sink statement of this data dependence, we place a blocking **wait**-statement that tests the iteration on which the current source instance depends. Different synchronization variables, implemented as bit-arrays, are used to synchronize the different static data dependences in the loop. The automatic generation of synchronization is addressed in [16, 17, 18, 19, 29].

On shared-address space architectures, parallel loops can be executed using fork/join-like parallelism. In this approach, a master thread executes the serial parts of a program, and initiates a number of slave threads, or workers, when a parallel loop is reached [28, 385-387]. After all iterations of this loop have been executed, the workers synchronize using barrier synchronization.

Whether all threads are actually executed on different physical processors or whether threads are scheduled competitively depends on the operating system. The way in which iterations of the parallel loop are assigned to the workers, on the other hand, is dependent on the **scheduling policy** [25, ch4][27, p73-74][28, p387-392][29, 296-298] that is used.

In **pre-scheduling**, either a block of consecutive iterations is assigned statically to each worker (block-scheduling), or iterations are assigned statically in a cyclic fashion to the workers (cyclic scheduling). To reduce the potential of load imbalance, we can also use **self-scheduling**. Here, workers enter a critical section to dynamically obtain a next chunk of iterations to execute. A small chunk size yields better load balance at the expense of synchronization overhead, whereas a large chunk size trades synchronization overhead for potential load imbalance. A good compromise is to vary the chunk size dynamically, such as assigning $1/p$ of the remaining iterations to each next worker, where p denotes the number workers (guided self-scheduling).

2.2 Java Restructuring

In figure 1, we illustrate our approach to exploiting implicit loop parallelism in Java programs.

A Java program `MyClass.java` is used as input of a source to source Java restructuring compiler (referred to as `javar` in the figure). This compiler applies data dependence analysis to program, and, depending on the outcome of this analysis, transforms the program to take advantage of implicit loop parallelism. Parallel loops (both DO-ALL- and DO-ACROSS-like loops) are explicitly coded using the multi-threading mechanism of Java. This implies that the transformed program can still be compiled into byte-code by any Java compiler (`javac`), and interpreted by any byte-code interpreter (`java` or, alternatively, an interpreter that is embedded in a browser or appletviewer). Because filenames are essential in Java, the transformed program is stored in the file `MyClass.java` again, although a copy of the original java program can be saved in `MyClass.old` if desired.

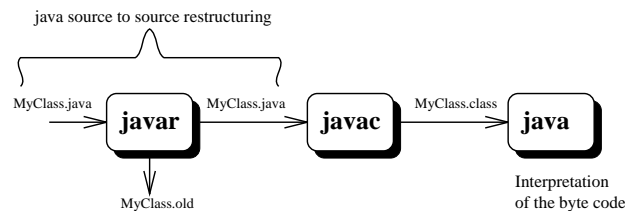


Figure 1: Restructuring, compiling, and interpreting

In this paper, we focus on the parallelization of stride-1 loops that have the following form:

```
L1:   for (int i = low; i < high; i++)
        body(i);
```

Note that conventional compiler techniques, like constant propagation, scalar forward substitution, induction variable substitution, and loop normalization [1, 11, 24, 29] may be useful to convert some other loop constructs into this form.

In essence, the parallelization of the a loop merely consists of adding a new method `runL1` to the class in which this loop occurs:

```
void runL1(int l, int h, int s) {
    for (int i = l; i < h; i += s)
        body(i);
}
```

If the loop occurs in an instance method, then `runL1` is also made an *instance method*. Otherwise, `runL1` made a *class (or static) method*. In this manner, all variables remain visible in the new loop-body (except for some local variables, which will be handled differently). Thereafter, a new class `LoopL1Worker` is constructed, having an instance method `run` that repetitively fetches new iterations from a pool and executes the `runL1` method for these iterations. In this manner, a number of threads can be used to concurrently execute the iterations of loop L1. Moreover, by providing a new entry method (viz. `runL1`) for each parallel loop, an arbitrary number of loops can be parallelized in each class.

In the next sections, we will further elaborate on this idea and provide a class hierarchy in which loop workers can be implemented.

3 Loop Parallelization in Java

In figure 2, we present our class hierarchy for implementing parallel loops in Java. The top layer of this hierarchy is completely independent of the source program. Hence, these classes may be provided in a separate, immutable package. The particular classes in the second layer of this hierarchy (viz. `LoopL1Worker..LoopLnWorker`) are constructed explicitly by the restructuring compiler and are added to the transformed Java program.

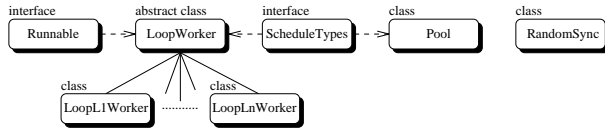


Figure 2: Class Hierarchy

3.1 Schedules

The interface `Schedules` is used to provide the classes `LoopWorker` and `Pool` with symbolic constants that represent different scheduling policies:

```

interface Schedules {
    static final int SCHED_BLOCK = 0;
    static final int SCHED_CYCLIC = 1;
    static final int SCHED_GUIDED = 2;
}
  
```

In this paper, we focus on the implementation of three scheduling policies, namely block, cyclic, and guided-self scheduling. Other scheduling policies, however, can be easily incorporated in the framework.

3.2 LoopWorker

The abstract class `LoopWorker` provides an abstraction of a worker that executes certain iterations of a parallel loop. Instance variables `l`, `h`, and `s` represent an execution set $[l, h)$ and stride for the iterations that the worker currently must execute. Instance variable `pool` and `sync[]` are used to store a pool of iterations and a number of synchronization variables that will be shared amongst all workers that execute the same parallel loop:

```

abstract class LoopWorker implements Runnable, Schedules {
    // Variables
    int l, h, s;
    Pool pool;
    RandomSync sync[];
    // Methods
    boolean nextWork() { ... }
    static void parloop(...) { ... }
} // End of Loop Worker
  
```

Because `LoopWorker` implements the interface `Runnable`, it must provide an instance method `run`. This method, however depends on the particular parallel loop for which a worker is required. As illustrated in figure 2, the restructuring compiler will explicitly construct a class description for the workers of a particular parallel loop (with label `Li`) by

extending the class `LoopWorker` with another class (viz. `LoopLiWorker`) that provides the appropriate `run` method. Hence, `LoopWorker` remains abstract.

Instance method `nextWork`, fetching the next amount of work from the shared pool, can already be provided for all workers. This method simply consists of calling a method `next` on the pool. The value returned by `next`, indicating whether more work has to be done, is also returned by `nextWork`:

```

boolean nextWork() {
    return pool.next(this);
}
  
```

Finally, the class `LoopWorker` provides a class method `parloop` that can be used to start the execution of a parallel loop. This method expects the lower and upper bound of a stride-1 parallel loop in `low` and `high`, some workers in array `worker` (which, in fact, will always be workers of a specific `LoopLiWorker` class), the number of synchronization variables required in `numS` and a scheduling policy in `tp`. First, a pool and the appropriate number of synchronization variables, both shared amongst all workers, are obtained. Thereafter, a thread is started for each worker. After the fork, the method performs a join by simply waiting for all threads to finish:

```

static void parloop(int low, int high, LoopWorker worker[],
                   int numS, int tp) {
    int numW = worker.length;
    Thread thread[] = new Thread[numW];
    Pool pool = new Pool(low, high, numW, tp);
    RandomSync sync[] = new RandomSync[numS];

    for (int i = 0; i < numS; i++)
        sync[i] = new RandomSync(low, high);

    // FORK
    for (int i = 0; i < numW; i++) {
        worker[i].pool = pool;
        worker[i].sync = sync;
        thread[i] = new Thread(worker[i]);
        thread[i].start();
    }
    // JOIN
    for (int i = 0; i < numW; i++) {
        try thread[i].join(); catch (Exception e) ;
    }
}
  
```

3.3 Pool

The class `Pool` defines the structure of a pool, which will become instantiated for each parallel loop. Instance variables `low` and `high` represent the execution set $[low, high)$ of a particular stride-1 parallel loop.

Instance variables `tp` and `numW` are used to record the scheduling policy and the number of workers. Instance variable `size` is used for several purposes:

```
class Pool implements Schedules {
    // Variables
    int low, high, tp, numW, size;
    // Methods
    Pool(int low, int high, int numW, int tp) { ... }
    synchronized boolean next(LoopWorker worker) { ... }
} // End of Pool
```

In the only constructor of this class, initial values are assigned to all the instance variables:

```
Pool(int low, int high, int numW, int tp) {
    this.low = low;
    this.high = high;
    this.numW = numW;
    this.tp = tp;

    // Initialization for Different Scheduling Policies
    switch (tp) {
    case SCHED_BLOCK:
        this.size = (int) Math.ceil(((double) high-low)
            / numW);
        break;
    case SCHED_CYCLIC:
        this.size = numW;
        break;
    case SCHED_GUIDED: // No initialization required
        break;
    }
}
```

Instance method `next` is used to fetch the next amount of work to be done. As illustrated in figure 3, during execution of a parallel loop, all the workers will compete for work in the shared pool. Therefore, the next amount of work must be obtained in a *critical section*. In Java, this mutual exclusion is obtained by making `next` a synchronized method:

```
synchronized boolean next(LoopWorker worker) {
    boolean more = false;

    switch (tp) {
    case SCHED_GUIDED:
        size = (int) Math.ceil(((double) (high-low))
            / numW);
        // FALL THROUGH
    case SCHED_BLOCK:
        more = (low < high);
        worker.l = low;
        worker.h = Math.min(low + size, high);
        worker.s = 1;
        low += size;
        break;
    case SCHED_CYCLIC:
        more = (size-- > 0);
        worker.l = low++;
        worker.h = high;
        worker.s = numW;
        break;
    }
    return more;
}
```

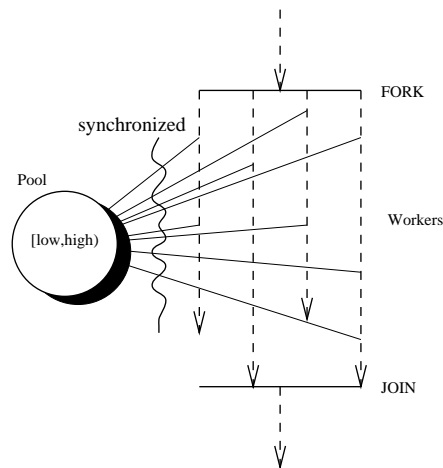


Figure 3: Execution of a Parallel Loop

Depending on the scheduling policy used, the method `next` sets the new values of the instance variables `l`, `h`, and `s` of the worker. Using cyclic scheduling, each worker exclusively starts with one of the iterations `[low, numW]` and steps through the *original* execution set with stride `numW`. Using block or guided-self scheduling, each worker obtains a block of iterations, where the block size varies for the latter policy (note that `low` becomes updated during each call to `next`). The return value of `next` indicates whether more work has to be done.

Note that in order to obtain a uniform interface between workers and the pool, the *pre-scheduled* policies have in fact been implemented as special versions of self-scheduled policies, where each worker directly obtain all its work in the first call to `next`, and terminates after the second call to this method. Other scheduling policies can be easily incorporated into this framework by making the appropriate additions to `Schedules` and `Pool`. Furthermore, note that the way in which the actual CPU time is given to the threads that implement a parallel loop depends on thread scheduler in the implementation of the Java Virtual Machine.

3.4 RandomSync

In the implementation of the pool, we already have exploited the fact that synchronized methods can be used to enforce mutual exclusion.

However, the implicit monitors [13] associated with all objects in Java make it also extremely simple to implement random synchronization for DO-ACROSS-like loops.

The class `RandomSync` defines the implementation of each synchronization variable. An instance variable `postarray` implements the bit-array, while `low` is used to record the lower bound of the execution set of the parallel loop:

```
class RandomSync {
  // Variables
  boolean postarray[];
  int low;
  // Methods
  RandomSync(int low, int high) { ... }
  synchronized void doWait(int j) { ... }
  synchronized void doPost(int i) { ... }
} // End of RandomSync
```

A synchronization variable is constructed with the following constructor, expecting the lower and upper bound of the parallel loop with execution set `[low,high)` as parameters:

```
RandomSync(int low, int high) {
  this.low = low;
  this.postarray = new boolean[high-low];
}
```

In this constructor, a new ‘bit’-array having one bit for each iteration is created (viz. bit `(i-low)` belongs to iteration `i`), and the lower bound is recorded.

Waiting for a post on the synchronization variable in iteration `j` is implemented using the following synchronized method `doWait`:

```
synchronized void doWait(int j) {
  if (low <= j)
    while (!postarray[j-low])
      try wait(); catch (Exception e) ;
}
```

The test `low <= j` makes this method non-blocking for out-of-bounds test (which always go backwards in the iteration space). Because a thread that executes `wait` becomes truly suspended, we have avoided a **busy-waiting** implementation. Because being notified does not necessarily mean that the appropriate bit actually has been set, `wait` is executed in a while-statement (rather than in an if-statement).

The bit that corresponds to iteration `i` is set using the following synchronized method `doPost`:

```
synchronized void doPost(int i) {
  postarray[i-low] = true;
  notifyAll();
}
```

The call `notifyAll()` will eventually resume all threads that are blocked on the same synchronization variable (although, of course, they may only re-enter the monitor one at the time).

3.5 Actual Loop Parallelization

Now, we are ready to discuss the steps that can be taken by a compiler to exploit implicit loop parallelism in a stride-1 loop of the following form, where `low` and `high` denote arbitrary expressions that remain constant during execution of the loop:

```
... class MyClass ... {
  ...
  ... myMethod(...) {
    ...
Li:   for (int i = low; i < high; i++)
        body(i);
    ...
  }
}
```

We assume that the restructuring compiler has ascertained that loop `Li` can be executed in either a DO-ALL-like or DO-ACROSS-like manner. Let `SVARS` denote the number of synchronization variables required (`SVARS == 0` for a DO-ALL-loop).

3.5.1 Construction of `LiRun`

First, depending on whether `myMethod` is a class method (or even a nameless static initializer), or an instance method, the following method `runLi` is added to `myClass` as a class or instance method:¹

```
[static] void runLi(int l, int h, int s, RandomSync sync[])
{ for (int i = l; i < h; i += s)
  body(i);
}
```

In this manner, all references within the loop-body will preserve their meaning, except for references to *local* variables of `myMethod` that are declared outside `Li` (including any arguments of `myMethod`). To enable the parallelization of such loops as well, in these cases the compiler first performs the preparatory rewriting discussed below. After the preparatory rewriting, the method `runLi` is generated, and any reference within the loop-body will preserve its meaning.

¹For readability, we use identifiers `runLi` and `LoopLiWorker`. In a practical implementation, however, the compiler is responsible for avoiding conflicts with existing identifiers.

For each local variable that is declared *outside* L_i but referenced *within* the loop-body, the compiler adds a new private variable to `MyClass` as a class or instance variable (thereby avoiding any name conflicts), depending on whether `myMethod` is a class method or instance method. Thereafter, each occurrence of the local variable is replaced by this new private variable. The declaration of each such local variable is simply deleted (note that Java requires that local variables are explicitly set before used), and possibly replaced by an assignment statement performing any explicit initialization performed in the former declaration. For a parameter of `myMethod`, the compiler generates a statement that copies the value of the variable used in the header into the private variable before any other statement of `myMethod`. Because private variables remain in existence after invocation of a method, it may be useful to assign `null` to all new private reference variables at the end of the method to make all data allocated during the invocation available to the garbage collector again (which will happen if this data is not accessible by means of other remaining reference variables).

EXAMPLE: Consider the following instance method `myMethod` that is a member of the class `MyClass`:

```
class MyClass {
    int u[];
    ...
    void myMethod(int x, int y) {
        int a = .., b = ..., c[] = new int[10000];
        ...
L1: for (int i = b; i < b+20; i++) {
        u[i] = x - a * u[i] * c[i];
        ...
    }
}
```

Before parallelization of the i -loop, the compiler handles some local variables as explained above (viz. x , a and c):

```
class MyClass {
    int u[];
    private int t_a, t_x, t_c[];
    ...
    void myMethod(int x, int y) {
        t_x = x;
        t_a = ..;
        int b = ...;
        t_c = new int[10000];
        ...
L1: for (int i = b; i < b+20; i++) {
        u[i] = t_x - t_a * u[i] * t_c[i];
        ...
        t_c = null; // Unhook
    }
}
```

After this preparatory rewriting, all the references within the following newly constructed instance method `runLi` will preserve their meaning:

```
void runLi(int l, int h, int s, RandomSync sync[]) {
    for (int i = l; i < h; i += s)
        u[i] = t_x - t_a * u[i] * t_c[i];
}
```

Providing a unique entry-point (viz. `runLi`) for each parallel loop enables the parallelization of an arbitrary number of loops in each class. In all cases where adding a new method and private variables to the class `MyClass` is undesirable, the compiler can resort to simply leaving all loops serial.

3.5.2 Insertion of Random Synchronization

For DO-ACROSS-like execution, the compiler adds the appropriate synchronization primitives to the loop-body according to methods described in the literature [16, 17, 18, 19, 29]). Signaling iteration i on the synchronization variable with number k is implemented as `sync[k].doPost(i)`. Likewise, waiting for iteration j on the synchronization variable with number k is implemented as `sync[k].doWait(j)`.

EXAMPLE: In the following single loop, there is a static flow dependence $S_1 \delta_{<} S_2$ on the array a with distance 5:

```
L1: for (int i = 0; i < N-5; i++) {
S1:     a[i+5] = c[i] * 30.0;
S2:     d[i] = a[i] / 20.0;
}
```

Hence, this loop can be executed in a DO-ACROSS-like manner by adding the appropriate synchronization. In this case, only one synchronization variable is required (viz. `SVARS == 1`):

```
L1: for (int i = 0; i < N-5; i++) {
S1:     a[i+5] = c[i] * 30.0;
        sync[0].doPost(i); // POST(ASYNC, i)
        sync[0].doWait(i-5); // WAIT(ASYNC, i-5)
S2:     d[i] = a[i] / 20.0;
}
```

3.5.3 Construction of LoopLiWorker

If `myMethod` is a *class method*, then the following class `LoopLiWorker` is constructed and added to the program:

```
class LoopLiWorker extends LoopWorker {
    public void run() {
        while ( nextWork() )
            myClass.runLi(l, h, s, sync);
    }
}
```

In this case, the whole loop `Li` is replaced by the following block, where `low` and `high` denote the lower and upper bound expression used in the original loop:

```
{ LoopLiWorker worker[] = new LoopLiWorker[NUM];
  for (int i = 0; i < NUM; i++)
    worker[i] = new LoopLiWorker();
  LoopWorker.parloop(low, high, worker, SVARS, SCHED);
}
```

Here, `NUM`, `SVARS`, and `SCHED` denote literal constants that are selected by the compiler representing, respectively, the number of workers, the number of synchronization variables, and the kind of scheduling policy for this parallel loop.

If, on the other hand, `myMethod` is an *instance method*, then the following slightly more elaborate class `LoopLiWorker` is constructed and added to the program:

```
class LoopLiWorker extends LoopWorker {
    MyClass target;

    LoopLiWorker(MyClass target) {
        this.target = target;
    }

    public void run() {
        while (nextWork())
            target.runLi(l, h, s, sync);
    }
} // End of LoopLiWorker
```

In this case, we replace the whole loop `Li` in `myMethod` by the following block, slightly differing from the block shown below because now variable `this` is passed to the constructor:

```
{ LoopLiWorker worker[] = new LoopLiWorker[NUM];
  for (int i = 0; i < NUM; i++)
    worker[i] = new LoopLiWorker(this);
  LoopWorker.parloop(low, high, worker, SVARS, SCHED);
}
```

By generating the code within a block, the compiler limits the life-time of the workers, making these earlier available to garbage collection. In the former case, different worker will simultaneously operate on class data, whereas in the latter case the worker may even operate in parallel on an object of class `MyClass`. However, because the loop has been marked as a parallel loop and, if required, synchronization primitives have been added by the compiler, this should not give rise to any race hazards.

4 Experiments

In this section, we present some experiments that have been conducted on an IBM RISC System/6000 G30 with four 601 processors using the IBM V1.0.2.B Java programming environment. In all experiments, the byte-code has been obtained by compiling the Java programs with the flag '-O'. Moreover, in all experiments (except the first one), we have run the interpreter with both just-in-time compilation as well as the actual concurrent execution of threads enabled.

4.1 Initialization Code

Consider, as first example, the following class having an array `a` as class variable and an array `b` as instance variable:

```
class Init {
    // Class Variables
    final static int N = 620;
    static double a[][] = new double[N][N];
    // Instance Variables
    double b[][];
    ...
} // End of Init
```

The class defines the following class method:

```
// Class Methods
public static void main(String args[]) {
L1: for (int i = 0; i < N; i++) {
    double z = Math.sqrt((double) i);
    for (int j = 0; j < N; j++)
        a[i][j] = z;
    }
    new Init().doit();
}
```

In this class method `main`, first the elements of `a` are initialized. Thereafter, a new object of the class `Init` is created, and the following instance method, in which an instance array `b` is initialized in a similar manner, is called *on* this object:

```
// Instance Methods
void doit() {
L2: for (int i = 0; i < N; i++) {
    double z = Math.sqrt((double) i);
    for (int j = 0; j < N; j++)
        this.b[i][j] = z;
    }
}
```

The constructor of the class has the following form:

```
// Constructors
Init() {
    b = new double[N][N];
}
```

Data dependence analysis reveals that both loop L1 as well as loop L2 can be converted into a parallel loop. Because no references to local variables declared outside the loop are made within the loop-body, no preparatory rewriting is required.

Because L1 appears in a class method, the whole loop is converted into the following call, where NUM and SCHED denote the number of workers that must be allocated and the scheduling policy for this loop, respectively:

```
public static void main(String args[]) {
    ...
L1: { LoopL1Worker worker[] = new LoopL1Worker[NUM];
      for (int i = 0; i < NUM; i++)
          worker[i] = new LoopL1Worker();
      LoopWorker.parloop(0, N, worker, 0, 0);
    }
    ...
}
```

Furthermore, the compiler adds the following class method to the class `Init`, in which the of L1 is executed for iterations defined by the parameters `l`, `h`, and `s`:

```
static void runL1(int l, int h, int s, RandomSync sync[]) {
    for (int i = l; i < h; i += s) {
        double z = Math.sqrt((double) i);
        for (int j = 0; j < N; j++)
            a2[i][j] = z;
    }
}
```

Finally, parallelization of L1 is completed by adding the following class to the program:

```
class LoopL1Worker extends LoopWorker {
    public void run() {
        while ( nextWork() )
            Init.runL1(l, h, s, sync);
    }
}
```

Likewise, loop L2 is replaced by the following construct that will initiate the execution of NUM workers for this loop:

```
void doit() {
L2: { LoopL2Worker worker[] = new LoopL2Worker[NUM];
      for (int i = 0; i < NUM; i++)
          worker[i] = new LoopL2Worker(this);
      LoopWorker.parloop(0, N, worker, 0, 0);
    }
}
```

Method `runL2` has the following form:

```
void runL2(int l, int h, int s, RandomSync sync[]) {
    for (int i = l; i < h; i += s) {
        double z = Math.sqrt((double) i);
        for (int j = 0; j < N; j++)
            b2[i][j] = z;
    }
}
```

Finally, the following class `LoopL2Worker` is constructed, saving the appropriate object in an instance variable `target`. The method `runL2` will be called repetitively on this object until all iterations have been executed:

```
class LoopL2Worker extends LoopWorker {
    Init target;
    LoopL2Worker(Init target) {
        this.target = target;
    }
    public void run() {
        while ( nextWork() )
            target.runL2(l, h, s, sync);
    }
}
```

In figure 4, we show the execution times of both the original loops and the parallel loops (using block scheduling) for a varying number of threads in case just-in-time compilation is disabled. In figure 5, the results of conducting the same experiments are shown using just-in-time compilation. In both cases we see that threads only induce a slight overhead in case one worker is allocated. Moreover, using 4 workers, the speedup becomes close to the best possible speedup of 4.

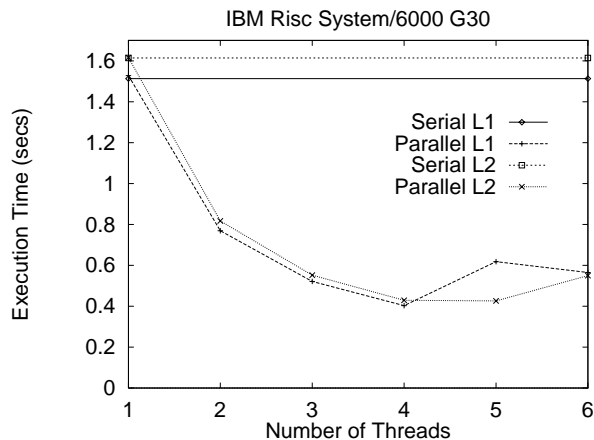


Figure 4: Initialization Code (no JIT)

4.2 Matrix Multiplication

To demonstrate the usefulness of the scheduling policies presented in this paper, we have conducted some experiments using the following class `Matmat`, consisting of only class variables and class methods:

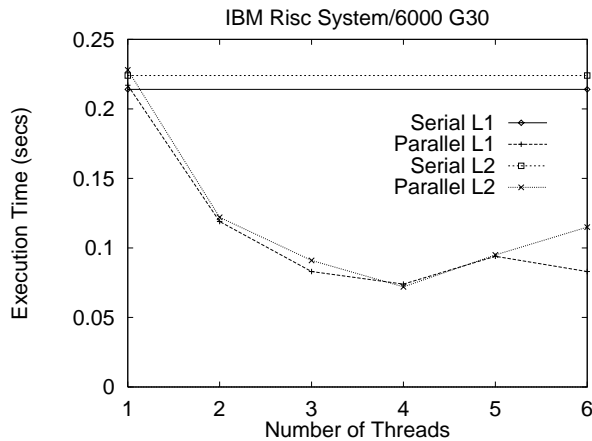


Figure 5: Initialization Code (JIT)

```

class Matmat {
    // Class Variables
    final static int M = 120, N = 120, K = 60;
    static double a[][] = new double[M][N];
    static double b[][] = new double[N][K];
    static double c[][] = new double[M][K];

    // Class Methods
    public static void main(String args[]) {
        ...
L1:   for (int i = 0; i < M; i++)
        for (int j = 0; j < K; j++)
            for (int k = 0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
        ...
    }
} // End of Matmat

```

The most straightforward way to parallelize this implementation on a shared-address-space architecture is to convert the outermost i -loop into a parallel loop. Using the framework of this paper, this transformation is performed by replacing the whole loop by a single call:

```

public static void main(String args[]) {
    ...
L1:   { LoopL1Worker worker[] = new LoopL1Worker[NUM];
        for (int i = 0; i < NUM; i++)
            worker[i] = new LoopL1Worker();
        LoopWorker.parloop(0, M, worker, 0, sched);
    }
    ...
}

```

Furthermore, the following class method is added to the class `Matmat`:

```

static void runL1(int l, int h, int s, RandomSync sync[]) {
    for (int i = l; i < h; i += s)
        for (int j = 0; j < K; j++)
            for (int k = 0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
}

```

Parallelization of the loop L1 is completed by adding the following class to the program:

```

class LoopL1Worker extends LoopWorker {
    public void run() {
        while (nextWork()) {
            Matmat.runL1(l, h, s, sync);
        }
    }
}

```

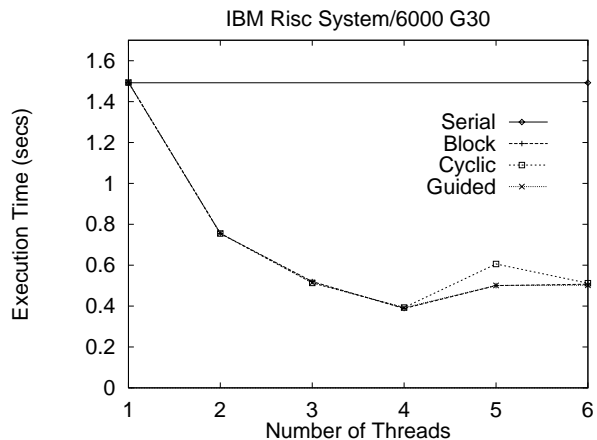


Figure 6: Matrix Multiplication

In figure 6, we show the execution time of the original serial loop and the parallel loop implemented using threads for a varying number of threads and the three different scheduling policies presented in this paper. Because work is spread evenly over the iterations, the scheduling policies have similar performance.

Now, suppose that the array `a` is used to store a lower triangular matrix, so that the innermost loop in both the original method as well as in the `runL1` method can be expressed as follows:

```

for (int k = 0; k < i; k++)

```

Obviously, this implies that the amount of work is not spread evenly over the iterations. In figure 7, we see that in this case block scheduling suffers from some load imbalance.

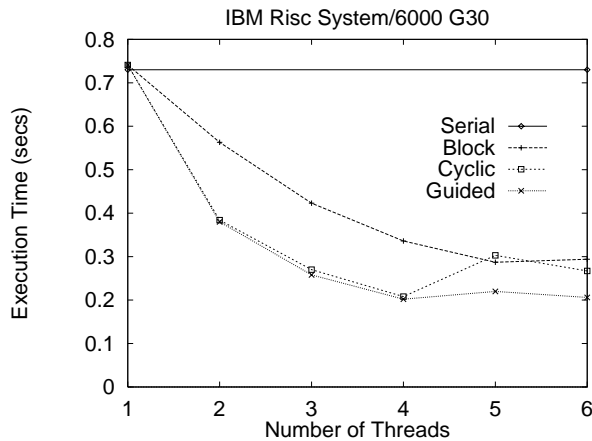


Figure 7: Triangular Matrix Multiplication

In this figure, we also see that this load imbalance problem is alleviated if more workers than actual processors are allocated.

Now, suppose that for some reason, only a few rows of the matrix stored in `c` have to be computed. This can be accomplished by using a boolean array `filter`:

```
for (int i = 0; i < M; i++)
    if (filter[i])
        ...
```

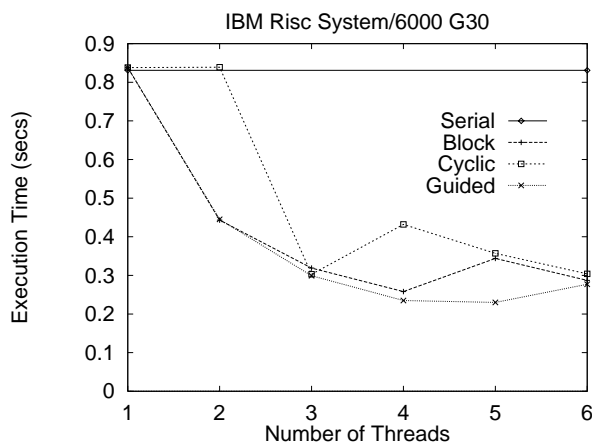


Figure 8: Matrix Multiplication (with filter)

If, for example, every other element of the boolean array `filter` is set, a severe load imbalance may result using cyclic scheduling, as illustrated in figure 8.

If, as another example, only the first 60 elements are set, guided self-scheduling suffers from a similar load imbalance, as can be seen in figure 9. These experiments indicate that, in general, we can make no decisive statement about which scheduling policy is the best.

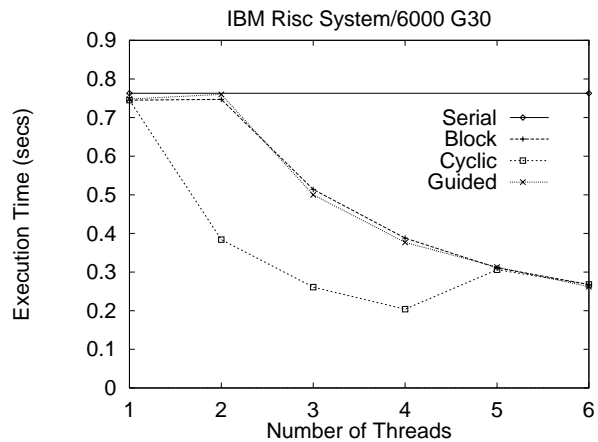


Figure 9: Matrix Multiplication (with filter)

4.3 Random Synchronization

In the following class, the loops L1 and L2, referring to class variables only, are candidates for parallelization:

```
class Dependence {
    // Class Variables
    final static int N = 300;
    final static int K = 2400;
    static double a[] = new double[N];
    static double b[][] = new double[N][K];

    // Class Methods
    public static void main(String args[]) {
        ...
        L1: for (int i = 7; i < N-5; i++) {
        S1:     a[i+5] = 10.0 - b[i-7][0]
        L2:     for (int j = 0; j < N; j++)
        S2:         b[i][j] = a[i] - 20.0;
        }
        ...
    }
} // End of Dependence
```

Although no data dependence is carried by the innermost `j`-loop, we rather convert the outermost into a parallel loop, because in that manner startup overhead of the parallel loop can be amortized over much more iterations.

Unfortunately, the i -carries the *static* data dependences $S_1\delta<S_2$ with distance 5 and $S_3\delta<S_1$ with distance 7. Hence, parallelization of loop L1 is only valid if the appropriate random synchronization is used to enforce the instances of these data dependences.

The whole loop L1 is replaced by the following call:

```
public static void main(String args[]) {
    ...
    L1: { LoopL1Worker worker[] = new LoopL1Worker[NUM];
        for (int i = 0; i < NUM; i++)
            worker[i] = new LoopL1Worker();
        LoopWorker.parloop(7, N-5, worker, 2, sched);
    }
}
```

The following class method `runL1` is added to the class `Dependence`:

```
static void runL1(int l, int h, int s, RandomSync sync[]) {
    for (int i = l; i < h; i += s) {
        sync[1].doWait(i-7); // WAIT(BSYNC, i-7)
        a[i+5] = 10.0 - b[i-7][0];
        sync[0].doPost(i); // POST(ASYNC, i)
        sync[0].doWait(i-5); // WAIT(ASYNC, i-5)
        for (int j = 0; j < Dep.N; j++)
            b[i][j] = a[i] - 20.0;
        sync[1].doPost(i); // POST(BSYNC, i)
    }
}
```

Finally, the following class is added to the program:

```
class LoopL1Worker extends LoopWorker {
    public void run() {
        while (nextWork())
            Dep.runL1(l, h, s, sync);
    }
}
```

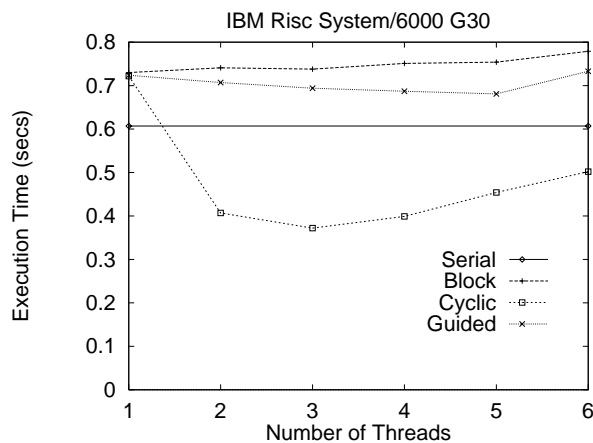


Figure 10: Random Synchronization

In figure 10, we show the execution time of the original fragment and the parallel implementation of the DO-ACROSS-loop for a varying number of threads on the IBM. Clearly, the overhead associated with random synchronization is more substantial than the overhead of a truly parallel loop. Furthermore, this experiment illustrates the using the wrong scheduling policy may effectively serialize the parallel loop.

4.4 Pixel Initialization

The following example is based on an applet example found in [20]:

```
public class MemoryImager extends Applet {
    ...
    final static int d_x = 700;
    final static int d_y = 1000;
    ...
    public void generateImage() {
        int pixels[] = new int[d_x * d_y];
        ...
        L1: for (int y = 0; y < d_y; y++)
            for (int x = 0; x < d_x; x++) {
                int r = (x*y) & 0xff;
                int g = (x*2*y*2) & 0xff;
                int b = (x*4*y*4) & 0xff;
                pixels[(y*d_x)+x] = (255 << 24)|(r << 16)|
                    (g << 8)|b;
            }
        ...
    }
}
```

In this example, an integer array is constructed that will be used to initialize a new image. Data dependence analysis reveals that every iteration refers to a unique element in the array `pixels`, so that both loops can be executed in parallel. Again, we prefer the parallelization of the outermost loop.

Because the loop-body of the y -loop refers to the local array `pixel` that is declared *outside* this loop, the following preparatory rewriting is performed:

```
public class MemoryImager extends Applet {
    ...
    private int t_pixels[];
    ...
    public void generateImage() {
        t_pixels = new int[d_x * d_y];
        ...
        L1: for (int y = 0; y < d_y; y++)
            for (int x = 0; x < d_x; x++) {
                ...
                t_pixels[(y*d_x)+x] = ...
            }
        ...
        t_pixels = null; // Unhook
    }
}
```

Thereafter, the following instance method `runL1` is added to the class `MemoryImager`:

```
void runL1(int l, int h, int s, RandomSync sync[]) {
    for (int y = 1; y < h; y += s)
        for (int x = 0; x < d_x; x++) {
            int my_r = (x^y) & 0xff;
            int my_g = (x*2^y*2) & 0xff;
            int my_b = (x*4^y*4) & 0xff;
            t_pixels[(y*d_x)+x] = (255 << 24)|(my_r << 16)|
                                 (my_g << 8)|my_b;
        }
}
```

As in the previous example, eventually a loop-worker is added to the program and the original loop is replaced by the appropriate construct. In figure 11, we show the execution time of the original serial loop and the parallel loop.

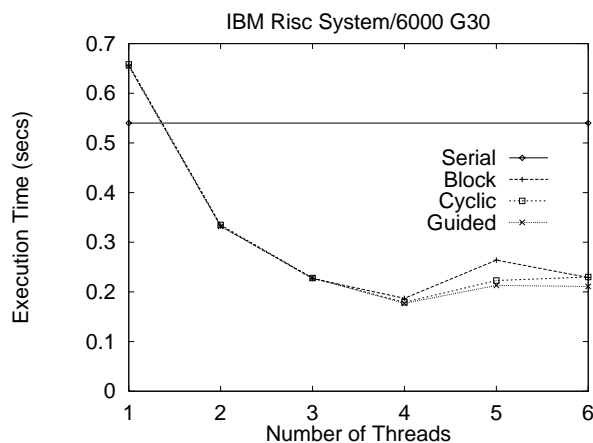


Figure 11: Pixel Initialization

5 Conclusions

In this paper, we have shown how a source to source restructuring compiler can exploit implicit loop parallelism in Java programs using multi-threading. We have presented a class hierarchy for implementing parallel loops in Java. The top layer of this hierarchy is completely independent of the source program, and can be provided in a separate, immutable package. Classes in the second layer of this hierarchy are constructed explicitly by the compiler and are added to the transformed program.

Experiments indicate that speedup can be obtained on a platform on which the Java byte-code interpreter supports actual concurrent execution of threads, whereas threads only induce a slight overhead for serial execution. Different scheduling policies are provided in our framework.

Future work will focus on incorporating the framework presented in this paper in a Java restructuring compiler. Furthermore, because the actual concurrent execution of threads is typically only supported on shared-address space architectures, future research will focus on how the networking capabilities of Java can be automatically exploited in a message-passing like manner to exploit computing power that is available over a network.

References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.
- [3] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, Boston, 1988.
- [4] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer, Boston, 1993.
- [5] Utpal Banerjee. *Loop Parallelization*. Kluwer, Boston, 1994.
- [6] D.W. Barron. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.
- [7] David Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Department of Computer Science, Rice University, 1987.
- [8] Ron G. Cytron. Doacross, beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–844, 1986.

- [9] Ron G. Cytron. Limited processor scheduling of doacross loops. In *Proceedings of the International Conference on Parallel Processing*, pages 226–234, 1987.
- [10] Erik H. D’Hollander. Partitioning and labeling of index sets in DO loops with constant dependence vectors. In *Proceedings of the International Conference on Parallel Processing*, pages 139–144, 1989. Volume 2: Software.
- [11] C.N. Fischer and R.J. LeBlanc. *Crafting a Compiler*. Benjamin-Cummings, Menlo Park, California, 1988.
- [12] David Flanagan. *Java in a Nutshell*. O’Reilly & Associates, Sebastopol, CA, 1996.
- [13] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
- [14] David J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, 1978. Volume 1.
- [15] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Programming*. The Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.
- [16] Zhiyuan Li and Walid Abu-Sufah. On reducing data synchronization in multiprocessed loops. *IEEE Transactions on Computers*, C-36:105–109, 1987.
- [17] Samuel P. Midkiff. *The Dependence Analysis and Synchronization of Parallel Programs*. PhD thesis, C.S.R.D., 1993.
- [18] Samuel P. Midkiff and David A. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the International Conference on Parallel Processing*, pages 544–551, 1986.
- [19] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36:1485–1495, 1987.
- [20] Patrick Naughton. *The Java Handbook*. McGraw-Hill, New York, 1996.
- [21] Patrick Niemeyer and Joshua Peck. *Exploring Java*. O’Reilly & Associates, Sebastopol, CA, 1996.
- [22] 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.
- [23] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, 1986.
- [24] Thomas W. Parsons. *Introduction to Compiler Construction*. Computer Science Press, New York, 1992.
- [25] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer, Boston, 1988.
- [26] 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.
- [27] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London, 1989.
- [28] Michael J. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, California, 1996.
- [29] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.