ReLooper: Refactoring for Loop Parallelism

Danny Dig
University of Illinois
Urbana-Champaign, USA
dig@illinois.edu

Cosmin Radoi
Mihai Tarce
Marius Minea
Politehnica University
Timisoara, Romania
radoi,tarce,marius@cs.upt.ro

Ralph Johnson
University of Illinois
Urbana-Champaign, USA
rjohnson@illinois.edu

ABSTRACT

In the multicore era, sequential programs need to be refactored for parallelism. The next version of Java provides `ParallelArray`, an array data structure that supports parallel operations over the array elements. For example, one can `apply` a procedure to each element, or `reduce` all elements to a new element in parallel. Refactoring an array to a `ParallelArray` requires (i) analyzing whether the loop iterations are safe for parallel execution, and (ii) replacing loops with the equivalent parallel operations. When done manually, these tasks are non-trivial and time-consuming. We present ReLooper, an Eclipse-based refactoring tool, that performs these tasks automatically. Experience with refactoring real programs shows that ReLooper is useful: it reduces the burden of analyzing and rewriting parallel loops, and it is fast enough to be used interactively.

1. INTRODUCTION

In the multicore era, programmers often turn to parallelism when they need to optimize their programs for performance. Sometimes, this requires rearchitecting the whole program. However, the most common way is to parallelize a program incrementally, by changing one piece at a time. Each step can be seen as a behavior-preserving transformation, i.e., a refactoring. The latter approach is safer, and programmers prefer to maintain a working, deployable version of the program.

To parallelize code, programmers often use parallel libraries that support different kinds of parallelism. For example, TBB [12], TPL [13], and ForkJoinTask [6] support *task parallelism* for C++, C#, and Java, respectively. TBB and TPL also support *data parallelism*, through *Parallel.For* utilities.

Java will include the `ParallelArray` [6] framework, a special kind of array that provides parallel operations. For example, one can `apply` a procedure to the elements of an array, `map` elements to new elements, or `reduce` all elements into a single value like a sum. The framework efficiently executes these parallel operations by splitting the computations on array elements among a pool of worker threads, and relying on a runtime library to balance the work among the processors in the system.

The parallel constructs provided by libraries are in general more verbose than parallel constructs provided by programming languages, thus they require many code changes. In addition, these libraries assume that all parallel computations do not interfere with each other, so they run without any synchronization. It is the programmer's responsibility to verify non-interference. This is non-trivial and time-consuming, and the code changes are tedious.

To refactor an existing array into a `ParallelArray`, the programmer constructs it by using factory methods (e.g., by copying elements from another array). Then the programmer identifies the loops that iterate over all the array elements and she analyzes each loop to infer its intent (e.g., the loop reduces all elements to a value). Next, she replaces the loop body with a call to the equivalent *parallel operation* (e.g., `reduce`). The parallel operation takes an *element operator* as an argument and executes it on each element. Since Java does not support anonymous functions (i.e., lambda expressions), the programmer needs to encapsulate the operator inside an anonymous class, by subclassing one of the 132 provided operator classes, and override the `op` method.

There are several preconditions that the programmer needs to check before applying the refactoring: (i) a loop iterates over *all* elements of the array, (ii) a loop does not contain blocking IO operations, (iii) the loop iterations do not have conflicting memory accesses.

Although parallelizing loops has received significant interest for scientific computation, much of this work is done in the context of scalar arrays (i.e., arrays of primitive types). Since such primitive types cannot be mutated, it suffices that the analysis determines that expressions `a[i]` and `a[j]` in nested loops do not refer to the same array element (i.e., $i \neq j$). However, in object-oriented programs that contain shared (heap-allocated) and mutable objects, the above analysis is not sufficient. The analysis needs to determine that different loop iterations (that are intended to be parallelized) do not write to the same object, which results in a race. For example, it needs to determine that two iterations of the loop do not write to a global, static field, or that no two array cells refer to the same object. In addition, even if the array contained distinct objects, the analysis needs to determine that starting from two distinct objects (reachable in different iterations) and following their field references we do not reach and write to the same object.

Converting an array into ParallelArray is non-trivial and time-consuming: the programmer needs to understand the alias relations and the state updates for all the statements (including arbitrarily long chains of method calls) in the loops over the array. Also the code rewriting is tedious: for example, in the real-world programs that we looked at, each parallelized loop required an average of 10 changes.

We have built an interactive refactoring tool, ReLooper, that automates the safety analysis and the rewriting of code. ReLooper is integrated with Eclipse's refactoring engine, so it offers all the convenient features of a refactoring engine: previewing the changes, preserving the formatting, undoing changes, etc. To use ReLooper, the programmer selects a target array or Vector and chooses ConvertToParallelArray from the refactoring menu. ReLooper performs the safety analysis and warns the programmer if some preconditions are not met, then ReLooper rewrites the code.

At the heart of ReLooper lies a data-flow analysis that determines objects that are shared among loop iterations, and detects writes to the shared objects. The analysis builds on the SSA (single static assignment) intermediate representation of a program, and it analyzes both programs in source code and in byte code. Our analysis is flow-sensitive (but path-insensitive), context-sensitive, field-insensitive.

Since the static analysis performed by ReLooper is conservative, it could give false warnings. Thus it is crucial that ReLooper be interactive: the programmer can decide to ignore the warnings and proceed anyway (after all, she understands the problem domain better than any tool), or she can cancel the current refactoring, fix the problems, and use ReLooper once more. In contrast to the previous work on automatic (i.e., non-interactive) loop parallelization [1, 8, 11, 16], our approach is agile enough for interactive development use, yet only requires minimal confirmation from the user.

Previous analyses for loop parallelization [8,16] model the whole heap and try to mark all aliases between all objects. Our analysis focuses only on the aliases that can potentially lead to data races in the loops intended for parallelization. Therefore, it is fast enough to be used in an interactive mode, while it still catches races in real-world programs.

This paper makes the following contributions:

- **Analysis.** We present several analyses that determine whether loops over arrays containing mutable, nested objects reachable through the heap, can be safely parallelized.

- **Tool.** We have implemented these analyses, the inference of parallel operations, and the code rewriting in an automated refactoring tool, ReLooper, integrated with the Eclipse IDE. ReLooper can be downloaded at: **http://refactoring.info/tools/ReLooper**

- **Evaluation.** We used ReLooper to parallelize several loops in real programs. ReLooper correctly inferred the parallel operations and it efficiently and effectively analyzed whether the loops can be parallelized. These experiments show that ReLooper is useful.

2. MOTIVATING EXAMPLE

As our running example, we use a small program that works with Particle bodies (this is a simplified version of an N-body particle simulation). Figure 1 shows the code for Particle. A particle has a position in space, given by its 2-D coordinates (x and y), and mass (stored in its m field). Class Particle provides an constructor, a factory method that creates random particles, and a moveBy method that updates the position of a particle.

```
class Particle {
	double x, y, m;

	public Particle(double x, double y, double m) {
		this.x = x;
		this.y = y;
		this.m = m;
	}

	static Particle createRandom(){
		return new Particle(Math.random(), Math.random(),
				Math.random() * 100);
	}

	void moveBy(double dx, double dy){
		this.x = x + dx;
		this.y = y + dy;
	}
}
```

Figure 1: Mutable Particle

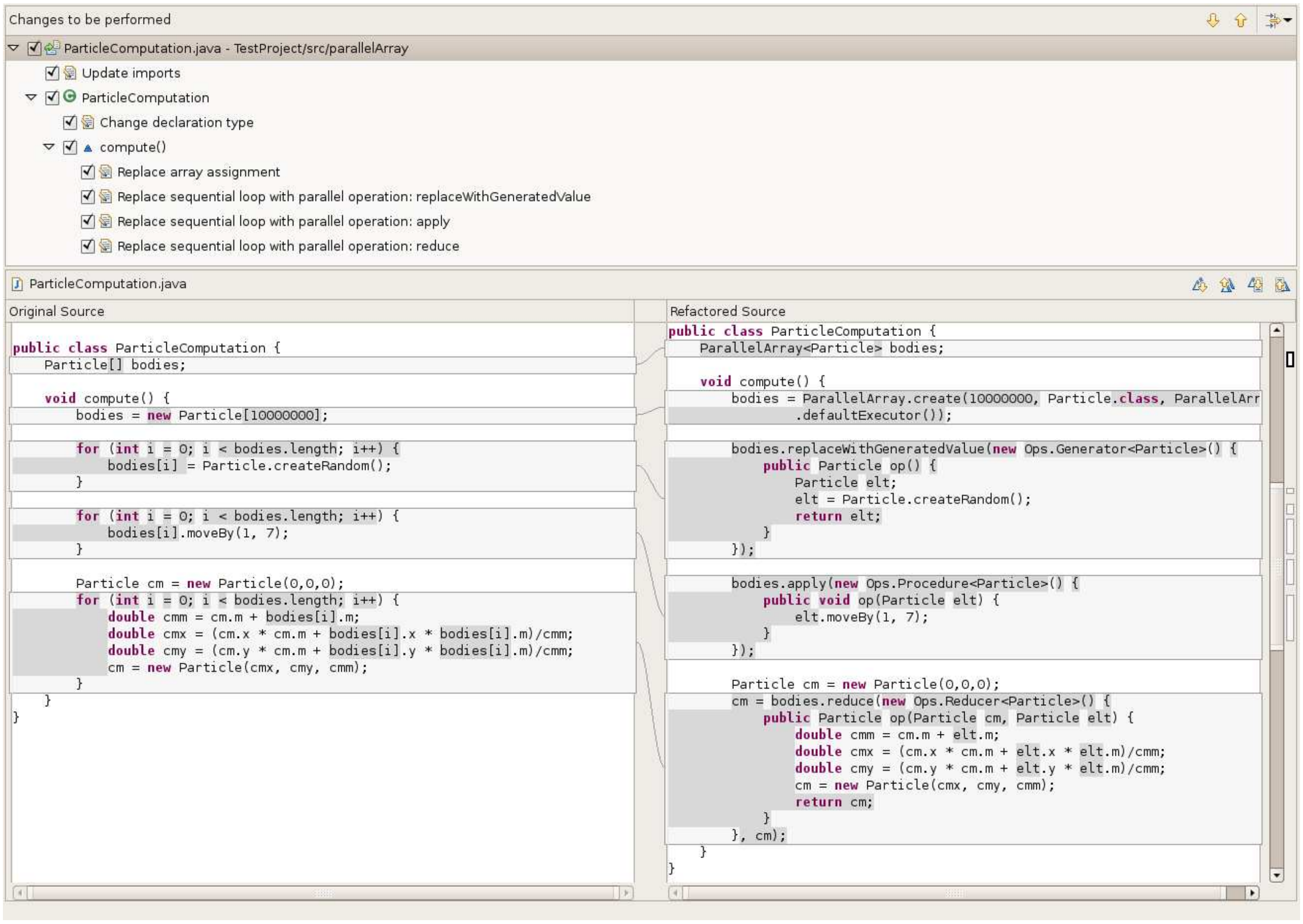
There is one client, ParticleComputation (shown on the left-hand side of Fig. 2) that performs different computations over an array of Particle objects. The first loop in method compute() initializes the array elements using the factory method createRandom(). The second loop iterates over all array elements and moves each particle by a delta. The third loop computes the center of mass for all the particles and stores the result in the cm variable.

The right-hand side of Fig. 2 shows the refactored program, using ParallelArray. For each loop that iterates over the array elements, ReLooper infers the intent of the loop and replaces it with the equivalent parallel operation from ParallelArray. In our example, the first loop initializes the array elements, so ReLooper replaces it by invoking the operation replaceWithGeneratedValue and passes an operator implemented as an anonymous Generator class. ReLooper overrides the op() method to create objects like in the original code. It correctly replaces the last two loops with the appropriate operations (apply and reduce) and generates the anonymous classes that encapsulate the element operators.

Notice that there is no synchronization construct in any of the parallel operations. Any inserted synchronization would have sequentialized the parallel operation, which defeats the whole purpose of running it in parallel. However, without synchronization, the parallel execution could end up in a race. ReLooper warns the user if any such races might occur.

Consider the operation that moves each particle, by applying moveBy to each element in parallel. The moveBy method mutates the state (as defined by fields x and y) based on the current field values. If the array contained duplicate elements (i.e., the same object reference being present in more than one cell of the array), mutating the state of the duplicate in parallel results in a data race. The final values of the state fields are non-deterministic, based on different thread interleavings. However, if there is no sharing between the array elements, the parallel updates can proceed safely.

ReLooper checks that the array cells do not share any objects. ReLooper determines that the loop that initializes the array elements creates new, unique objects in each iteration.



```
Changes to be performed
ParticleComputation.java - TestProject/src/parallelArray
    Update imports
    ParticleComputation
        Change declaration type
        compute()
            Replace array assignment
            Replace sequential loop with parallel operation: replaceWithGeneratedValue
            Replace sequential loop with parallel operation: apply
            Replace sequential loop with parallel operation: reduce
```

ParticleComputation.java

Original Source

```java
public class ParticleComputation {
    Particle[] bodies;

    void compute() {
        bodies = new Particle[10000000];

        for (int i = 0; i < bodies.length; i++) {
            bodies[i] = Particle.createRandom();
        }

        for (int i = 0; i < bodies.length; i++) {
            bodies[i].moveBy(1, 7);
        }

        Particle cm = new Particle(0,0,0);
        for (int i = 0; i < bodies.length; i++) {
            double cmm = cm.m + bodies[i].m;
            double cmx = (cm.x * cm.m + bodies[i].x * bodies[i].m)/cmm;
            double cmy = (cm.y * cm.m + bodies[i].y * bodies[i].m)/cmm;
            cm = new Particle(cmx, cmy, cmm);
        }
    }
}
```

Refactored Source

```java
public class ParticleComputation {
    ParallelArray<Particle> bodies;

    void compute() {
        bodies = ParallelArray.create(10000000, Particle.class, ParallelArr
                .defaultExecutor());

        bodies.replaceWithGeneratedValue(new Ops.Generator<Particle>() {
            public Particle op() {
                Particle elt;
                elt = Particle.createRandom();
                return elt;
            }
        });

        bodies.apply(new Ops.Procedure<Particle>() {
            public void op(Particle elt) {
                elt.moveBy(1, 7);
            }
        });

        Particle cm = new Particle(0,0,0);
        cm = bodies.reduce(new Ops.Reducer<Particle>() {
            public Particle op(Particle cm, Particle elt) {
                double cmm = cm.m + elt.m;
                double cmx = (cm.x * cm.m + elt.x * elt.m)/cmm;
                double cmy = (cm.y * cm.m + elt.y * elt.m)/cmm;
                cm = new Particle(cmx, cmy, cmm);
                return cm;
            }
        }, cm);
    }
}
```

Figure 2: The programmer selects the `bodies` array and ConvertToParallelArray refactoring. ReLooper converts the sequential loops (left-hand side) into parallel loops (right-hand side) using the ParallelArray framework.

In addition, it determines that the objects returned by two calls to `Particle.createRandom` do not share anything.

Moreover, ReLooper checks that the loop iterations do not mutate other shared objects. For example, the loop that computes the center of mass of all particles mutates the shared object `cm`. ReLooper's analysis catches this write to a shared object, but the analysis allows it because this problem is eliminated when `cm` becomes the *accumulator* variable for the `reduce` operation (internally, the reduction creates fresh `cm` variables, and accumulates them in a final step).

3. THE REFACTORING TOOL

The process of using ReLooper has two steps. In the first step, the programmer (who understands the problem domain) expresses the intent to parallelize some loops by selecting an array (from now on simply referred as the target array) and the ConvertToParallelArray operation from the refactoring menu. ReLooper analyses whether the loops intended for parallelism can be safely parallelized and reports to the programmer any potential problems, e.g., data races in the to-be parallelized loops. The programmer can decide to ignore the warnings and to proceed to the next step, or she can cancel the current refactoring, fix the problems, then re-run ReLooper.

In the second step, the programmer confirms the changes that she intends ReLooper to apply. For each loop, there are two choices: (i) replace the loop with the parallel operation, or (ii) leave the loop sequential, but replace accesses to the indexed elements from the target array with indexed elements from `ParallelArray`. By default, ReLooper chooses the first choice for those loops where it did not find any problems, and chooses the second choice for loops where it found problems. The programmer can overwrite ReLooper's default selection, and choose to parallelize some loops, but leave others sequential.

Next, we present the pseudocode of the refactoring algorithm. The algorithm takes as input the target array or Vector object, the source code of the application that uses it, as well as the bytecode of all libraries invoked from the source code. While the source code is needed for the refactoring itself, the safety analysis works on the bytecode and correctly takes all library behavior into account.

First the algorithm searches the source code for all accesses to `array` (e.g., `array.length`) as well as accesses to array elements (e.g., array[i].field). Then the algorithm instruments the write accesses so that they can be analyzed later by the static analysis. This means inserting some "markers" that can be later retrieved from the control-flow graph.

Next, for each loop that iterates over the `array`, the algorithm checks whether it is safe to run all the loop iterations in parallel. The analysis checks three preconditions: (i) the loop traverses all elements of the `array`, (ii) the loop does not have blocking IO operations, and (iii) the loop iterations do not have conflicting memory accesses.

ReLooper presents to the programmer the loops that it thinks are safely parallelizable. The programmer can change the set of parallelizable loops.

Lastly, during the transformation step, for each loop that is parallelizable, the algorithm infers the equivalent parallel operation from `ParallelArray` and replaces the loop with the inferred parallel operation. All other accesses, i.e., outside of loops or accesses in loops that are not safe, are replaced by ReLooper with the equivalent accessors from `ParallelArray`.

4. TRANSFORMATIONS.

ReLooper applies several transformations.

Type Declaration. ReLooper changes the type declaration of the target array into a `ParallelArray`. For the example in Fig. 2, it changes `bodies` with the parametric type `ParallelArray<Particle>`. If the original array contained elements of primitive types (e.g., int, double), ReLooper would have replaced the type with one of the more specific types for scalars (e.g., `ParallelDoubleArray`).

Initializer. ReLooper replaces the array initializer (i.e., code that allocates storage for the array) with a call to the `create` factory method that creates a `ParallelArray`. ReLooper passes to `create` the same array capacity, and specifies the base element type and the pool of worker threads that will be used at runtime. For example, in Fig. 2, ReLooper invokes the `defaultExecutor()` pool which arranges to use most of the processors available at runtime. If the array was initialized from another array, ReLooper would have invoked another factory method, `createUsingHandoff` and passed the other array as argument.

Parallel Operations. ReLooper replaces a loop with the equivalent parallel operation from `ParallelArray`. Table 1 describes how ReLooper infers the parallel operations based on the kind of array accesses in the loop. We denote an array indexed element with `a[i]`, a field of an array element with `a[i].x`, and a reference to the loop index variable (other than in indexed elements) with `i`. Each row describes the predicate of read/write accesses that is evaluated to decide when to use one particular operation.

If a loop contains accesses triggering more than one operation, ReLooper chooses the most specific operation. This is both simpler to understand, and is faster at runtime.

Notice that `replaceWithMappedIndex` is the only API method from `ParallelArray` that provides access to the loop index variable, and it also enables writing the array elements. Therefore, anything that can be expressed with other operations can be expressed with `replaceWithMappedIndex`. For example, an `apply` could be expressed as replacing an array element with itself, and ignoring the loop index variable. However, this is both harder to understand, and less efficient, thus ReLooper infers the most specific operation.

Element Operator. After inferring the kind of parallel operation, ReLooper creates the element operator, i.e., the functor that `ParallelArray` invokes on each element. Since Java does not currently support lambda expressions, the operator is encapsulated as an `op` method in one of the `Ops` operator subclasses. ReLooper chooses the right operator subclass

Operation	Element Access
`replaceWithMappedIndex`	`a[i]` = expression with `a[i]` $\wedge$ read `i`
`replaceWithMapping`	`a[i]` = expression with `a[i]`
`replaceWithGeneratedValue`	`a[i]` = expression not containing `a[i]`
`replaceWithValue`	`a[i]` = constant/literal
`apply`	read `a[i]` $\vee$ read `a[i].x` $\vee$ write `a[i].x`
`reduce`	accumulating all elements

Table 1: Decision table used by ReLooper to infer the parallel operations based on accesses in each loop.

among the 132 possible choices offered by `ParallelArray`.

Each operation takes specific operators, for example, `apply` takes an `Ops.Procedure`, and `reduce` takes an `Ops.Reducer`. Since the `Ops` hierarchy provides several choices for operators over primitive types (e.g., `Ops.DoubleReducer`), object types (e.g., `Ops.ObjectToInt`), or parametric types (`Ops.Op<A,R>`), ReLooper chooses the correct operator based on the types of the array elements and the return type of the operation.

ReLooper extracts the body of the loop into the `op` method and replaces the accesses to array elements, with accesses to the arguments of the operator. For example, in Fig. 2, ReLooper replaces accesses to the element `bodies[i]` with accesses to the argument `elt`. In addition, for operators that need to return results (e.g., a `Generator`), ReLooper adds the return statement (see Fig. 2, line 24).

Loop styles. Besides classical `for` loops in Java ReLooper handles the newer-style `forEach` loops too. Since a `forEach` loop in Java can only be used to read array elements, there are only two possible equivalent operations: `apply` or `reduce`.

Other Accesses. If the analysis determines that a particular loop is not safely parallelizable, ReLooper can still parallelize other loops over the array. As for the unsafe loop, or array accesses outside of loops, ReLooper replaces the array accesses with calls to the equivalent `ParallelArray` APIs (and leaves the loop sequential). For example, it replaces a read access `a[i]` with `a.get(i)`, and a write access `a[i] = expression` with `a.set(i, expression)`.

Loops over Vector. ReLooper allows a programmer to parallelize loops over `Vector` types, besides arrays. `Vector` is a collection class that implements a growable array of objects.

The `Vector` class provides an internal `Iterator` that allows one to express loops using the `hasNext` and `next` methods. ReLooper converts such loops too. Most of the `Vector`'s API method are syntactic sugar for accesses over arrays (e.g., `get(i)` is equivalent to read `a[i]`, and `set(i, Obj)` is equivalent to `a[i] = Obj`), so ReLooper handles them similarly. The methods with no obvious array equivalent are the ones that enable growing, e.g., `add` and `addElement`. ReLooper replaces loops that add elements with the `replaceWithMappedIndex()` operation. This operation allows to insert elements at a specific position, and ReLooper refers to this position by an `AtomicInteger` index variable that is incremented atomically.

5. PROGRAM ANALYSIS

This section describes the preconditions that must hold to ensure the safety of loop parallelization, i.e., the safety of replacing a loop with a call to parallel operation. The preconditions in Section 5.1 ensure that a loop traverses all elements of an array. Section 5.2.1 ensures that there are no loop-carried dependences, while Section 5.2 ensures that the memory updates in loop iterations are not conflicting.

5.1 Linear Traversal

Since a parallel operation applies the given element operator to *all* elements of a `ParallelArray`, the refactored code can have different semantics if the original loop does not iterate over all array elements. ReLooper checks whether a loop iterates over all elements *linearly*, i.e., one element at a time, without skipping.

ReLooper treats the common case when the loop index variable is also the index used in the array accesses in the loop body (otherwise, ReLooper does not parallelize the loop). To check that all elements of the array are iterated, the analysis checks whether the loop index variable starts from `0` and goes to the array's `length`, and is incremented by one in each iteration. We also allow traversal in the reversed order, from the last element to the first.

In addition, the analysis ensures that the array traversal does not intentionally abort before reaching the last element. Since a parallel operation is not user-cancellable, the refactored code would inevitably traverse all elements, whereas the original code only traversed some. The analysis checks that the loop body does not contain statements that abort the loop traversal: `return`, `break`, or `throw`. We allow `continue`, a statement that stops the processing of the current iteration and jumps to the next iteration. ReLooper replaces `continue` statements with `return` statements inside the element operator; this ensures that the processing of the current element terminates, while allowing processing of other elements. However, the analysis can not ensure that a loop does not abort through unchecked exceptions.

Note that if the given loop is a `forEach`, ReLooper does not need to check that the loop iterates all elements, since this is guaranteed by the semantics of `forEach`.

Additionally, the analysis checks that the loop body contains only references to the currently indexed element. If the loop body refers to other array elements (potentially written in other iterations), this would introduce a dependency between those iterations.

5.2 Conflicting Memory Accesses

Next, the analysis determines whether the loop iterations have conflicting memory accesses. We have developed a data-flow analysis that detects updates to shared objects in the parallel sections of the program. Before giving more details and examples, we define the key concepts of *parallel section*, *shared object*, and *update*.

Definition 1. **Parallel Section.** A parallel section contains program instructions that are executed in parallel by a particular loop that is a candidate for parallelization.

Definition 2. An **object** is a memory location (on the heap) and is named by a `new` statement.

A **reference** is a strongly typed pointer to an object.

A **shared object** is an object that is pointed to by references from different iterations of a parallel loop. All objects that are referenced by the fields of a shared object are shared themselves.

Definition 3. An **update of shared object** is a write to a field of a shared object. We also refer to this update as a **race**.

We illustrate these definitions with a simple example presented in Fig. 3.

```
1  class Test {
2     Particles[] bodies;
3     int iters;
4
5    void introduceSharing(){
6      Particle cropper = new Particle(2,3,4);
7      for (int i = 0; i < bodies.length; i ++){
8         this.iters = i;
9         Particle d = cropper;
10        d.x = i;
11        d = new Particle(0,0,1);
12        d.y = i;
13     }
14   }
15 }
```

Figure 3: Shared Objects

During a parallel section (lines 8–12), a reference could point all the time to a shared object or it could point to a shared object only at a specific program point. For example, the object pointed by the reference `this` is a shared object. This is the object whose `introduceSharing` method runs the loop. The reference `this` points to a shared object throughout the parallel section. The object `Particle(2,3)` allocated on line 6 is a shared object too. However, the object `Particle(0,0,1)` allocated on line 11 is not a shared object. The reference `d` points to a shared object at line 9, but points to a non-shared object after line 11. Therefore, line 10 introduces a race (an update to a field of a shared object), but the update at line 12 does not. Also the update at line 8 introduces a race, since it is a write to a field of a shared object (i.e., a field of `this`).

ReLooper detects the races at lines 8 and 10, and presents them to the user. Next, we will show how the analysis works.

In order to detect races, the analysis (i) follows the shared objects through the control flow graph (CFG) and (ii) detects updates to shared objects.

The analysis visits program instructions following the CFG, and marks the shared objects. This marking happens through the transfer functions associated with each instruction. It also checks whether the current instruction writes a field of a shared object. The join operator for several edges is set union, and the algorithm uses a worklist approach to iterate up to a fixpoint.

Our sharing analysis is an interprocedural data-flow analysis. The analysis is flow-sensitive, context-sensitive, path-insensitive, and field-insensitive.

The analysis is built upon the SSA (static single assignment) representation of the program. We use the WALA [14] library to get the SSA form for every method of a program. In SSA, every variable is assigned exactly once. SSA splits variables in the original code, so that each definition of a variable generates its own version.

Consider the example in Fig. 4. Each new definition of `c` generates a new variable version. A variable definition is an assignment of a newly allocated object. Therefore, each variable version can be seen as a *label* of a heap-allocated object.

In our example, lines 1 and 4 are definitions of `c`, therefore they generate new versions, $\mathtt{c}_1$, respectively $\mathtt{c}_2$. Note that line 2 which introduces an alias to `c`, is not a definition (i.e., it does not label a new heap location), thus it does not appear in the SSA form. Instead, future uses of `d` (up to a new definition of `d`) are seen as using $\mathtt{c}_1$. This tremendously

```
Particle c = new Particle(2,3,4);     1  Particle c_1 = new Particle(2,3,4);
Particle d = c;                       2  if (condition) {
if (condition) {                      3    c_2 = new Particle(0,0,1);
  c = new Particle(0,0,1);            4  }
}                                     5  c_3 = phi(c_1, c_2)
read c                                6  read c_3
```

Figure 4: Coverting a program into a conceptual SSA form

simplifies the reasoning about aliases.

Since each variable version is defined exactly once (i.e., it is immutable), by tracking these versions and not the original variables, we know at each point in the program which are the objects that can be reached through references. This is exactly the reason why we are using the SSA form: to track the objects, not their references.

Note that a variable version can point to more than one allocated object. This can happen due to the path-insensitivity of SSA. For example, the read at line 5 can see either version $\texttt{c}_1$ or $\texttt{c}_2$, so the SSA form uses a so-called ϕ-function, that merges the two versions in a new version, $\texttt{c}_3$, and uses $\texttt{c}_3$ from now on when reading `c`. For conciseness and intuitiveness, from here on, we will be referring to an SSA variable version as an *object*.

The SSA form also preserves the program flow from the original CFG.

Next, we present our data-flow analysis by first explaining the intraprocedural part, and later we will expand it with method calls (i.e., interprocedural).

5.2.1 Intraprocedural analysis

We denote by S the set of all shared objects. The analysis starts from the entry methods. We define an *entry method* to be a method that contains references to the target array.

In each entry method, S initially contains only its arguments and the object pointed by `this`. For each instruction in the entry method's SSA, we update S based on a transfer function.

For the same instruction we associate different transfer functions, based on whether the instruction is in a parallel section. Figure 5 presents the transfer functions for instructions in a parallel section. We will use the following generic notations: `C` is a class, o_1, o_2, o_3 are objects, `f` is a field access and `C.f` is an access to a static field, `[e]` is an access to an indexed element of an array.

We present the transfer functions as rules that update the set of shared objects, S. The upper part contains the rule premises, and the lower part shows the additional members introduced in the shared set S, for different types of instructions. In all other cases, S does not change (the transfer function is the identity). Note that since our analysis is field-insensitive, assigning a shared object $\texttt{o}_2$ to a field of an object $\texttt{o}_1$ marks the whole object $\texttt{o}_1$ as shared.

For the sequential section, all the rules in the parallel section apply too. In addition, we have new rules. Figure 6 presents only the added rules. In essence, any assignment to an object $\texttt{o}_1$ marks that object as shared, whereas in the parallel section, $\texttt{o}_1$ is marked as shared only if the right-hand side of the assignment denotes a shared object.

We illustrate this analysis using a small variation of our motivating example. The program in Fig. 7 is similar with the one in Fig. 2, but the code in `compute` is different. We have extracted the loop that moves particles into the method

$$\frac{o_2 \in S}{o_1 \in S} \quad \text{for } o_1 = o_2[e], o_1[e] = o_2, o_1 = o_2.f, o_1.f = o_2$$

$$\frac{}{o_1 \in S} \quad \text{for } o_1 = C.f$$

$$\frac{o_2 \in S \vee o_3 \in S}{o_1 \in S} \quad \text{for } o_1 = \phi(o_2, o_3)$$

Figure 5: Transfer functions for instructions in parallel sections

$$\frac{o_1 \neq \mathit{targetArray}}{o_1 \in S} \quad \text{for } o_1 = o_2[e], o_1 = o_2.f, o_1 = C.f, o_1 = \mathit{new}\ C()$$

Figure 6: Extra transfer functions for instructions in the sequential sections, in addition to the functions from Fig. 5.

`moveAll`, we removed the loop that computes the center of mass, and instead we added another loop (lines 32–36) that introduces sharing. The left-hand side of the figure shows the source code where we marked the parallel sections with [[ ]] symbols. The right-hand side shows the set of shared objects, S, as it is updated by the transfer functions. On the same line with the instruction, we show the output of its associated transfer function.

For now, we will only focus on the lines 32–36 (that highlight the intraprocedural analysis), and we will revisit the whole example once we introduced the interprocedural analysis. The assignment in line 32 (in a sequential section) uses the transfer function defined in Fig. 6 and marks the `cropper` object as shared, by adding it to the S set. Before this line, $S = \mathit{this}$. After this line, $S = \{\mathit{this}, \mathit{cropper}_1\}$

In line 35, there is an assignment of a shared object, $\texttt{cropper}_1$ to an indexed element of `bodies[]`, therefore, according to the transfer function in Fig. 5, `bodies` is marked as shared (i.e., $S = \{\mathit{this}, \mathit{cropper}_1, \mathit{bodies}_1\}$).

Since line 35 is executed on a branch, on line 36, the analysis uses the merge function, defined as set union, to merge the results of the two branches. Therefore, $S = \{\mathit{this}, \mathit{cropper}_1, \mathit{bodies}_1\}$.

Updates to Local Variable References

Consider the example of the center of mass loop in Fig. 2. The local variable `cm` is declared outside of the parallel section, but its reference is written inside of the parallel section. This is a race.

The SSA representation of the program only keeps track of the definitions and uses of variables. It is unaware of the local variable declaration point in the CFG. This limitation makes it dificult to use the SSA representation for detecting races as the one described above.

In order to detect this class of race conditions, we augmented the data-flow analysis with an AST-based analysis that handles the scenario presented above.

Our analysis allows one exception from this rule. Namely, if it determines that the written variable is used for accumulating the result of a `reduce` operation. Internally, the implementation of the `reduce` creates fresh variables, and accumulates them in a final step, which effectively eliminates

```
// Sequential version                      // Set of shared objects                        // Set of shared objects

class Particle {
   ... same as before

  static Particle createRandom(){             Particle.createRandom() {}
    return new Particle(Math.random(),...       return ...not shared...
                          Math.random());
  }                                           Particle.createRandom() return ...not shared...

  void moveBy(double dx, double dy){          this.moveBy(dx, dy) {}                           this.moveBy() {this}
                                                {}                                               {this}
    this.x = x + dx;                            {}                                               this.x = ... WRITE_TO_SHARED {this}
    this.y = y + dy;                            {}                                               this.y = ... WRITE_TO_SHARED {this}
  }                                           this.moveBy(dx, dy)                              this.moveBy()
}

class ParticleComputation {

  void compute() {                            this.compute() {this}
    Particle[] bodies;
    bodies = new Particle[10000000];            {this}

    [[
      bodies[i] = Particle.createRandom();      Particle.createRandom()   {this}
    ]]

    this.moveAll(bodies);                       this.moveAll(bodies)  {this}

    Particle cropper = new Particle(10,2,1);    {this,cropper}
    [[
      if (i % 13)                               {this,cropper}
        bodies[i] = cropper;                      {this, cropper, bodies}
    ]]                                          {this, cropper, bodies}

    this.moveAll(bodies);                       this.moveAll(bodies)   {this, cropper,bodies}
 }                                            this.compute()

 void moveAll(Particle[] arr) {               this.moveAll(arr) {this}                         this.moveAll(arr) {this, arr}
    [[
      arr[i].moveBy(0,7);                       arr[i].moveBy(0,7)                               arr[i].moveBy(0,7)
    ]]
 }                                            this.moveAll(arr)                                this.moveAll(arr)
}
```

Figure 7: Extended example shows the propagation of the sharing information, and the detection of updates to shared objects. Left column shows the original code, where we marked the parallel sections. The middle column shows the set of shared objects after the analysis of each line of code. The right column shows the shared objects for a different analysis of the same method. For each method call, we underline the receiver and arguments that are shared. For each analysis of a method, we show, on the same line as the method declaration, which formal arguments are shared.

the loop-carried dependency.

5.2.2 *Interprocedural analysis*

Next, we present how the analysis handles method calls.

Every method declaration has its own SSA form. This is computed modularly by WALA. A method's signature includes the receiver (its own `this`), and a list of formal method arguments.

When our analysis visits an `InvokeMethod` SSA instruction, the analysis performs the following:

1. It constructs the set of shared objects, S_m, for the invoked method. For this, it binds the formal method arguments to the actual parameter. For each actual parameter that is marked shared, it adds its corresponding formal parameter in S_m.

2. It computes the data-flow analysis for the invoked method using the intraprocedural analysis presented above. If the method is invoked from a parallel section, the analysis uses the transfer functions for parallel sections, otherwise it uses the functions for the sequential sections.

3. It propagates the changes from the set S_m back to the calling context. For each formal parameter that was marked as shared in this method (and thus added to S_m), the analysis adds the corresponding actual parameter to the shared set S of the invoking method.

Once a method invocation is visited by our algorithm, the analysis memoizes the results of the data-flow analysis for that particular sharing pattern of the input parameters. The analysis also uses this mechanism to handle recursive methods, by not starting a new analysis for an already memoized starting point.

5.2.3 *Detecting updates to shared objects*

The analysis associates a detection rule for every SSA instruction in the parallel sections. These rules have as premises conditions on the sharing set S and detects whether that instruction writes to a shared object. Figure 8 presents the detection rules.

5.3 Detecting I/O operations

The `ParallelArray` framework is implemented using lightweight tasks, a thread-like entity that has a light overhead. The documentation specifies that tasks (and therefore parallel operations) should not perform blocking I/O operations,

$$\frac{o_1 \in S}{WritesToShared} \quad \text{for } o_1[e] = o_2, o_1.f = o_2$$

$$\frac{}{WritesToShared} \quad \text{for } C.f = o_1$$

Figure 8: Detection rules for instructions in the parallel sections

since this can result in poor performance. Essentially, a task blocked on I/O leads to blocking its corresponding worker thread, which blocks its corresponding core. Since the framework only spawns a number of worker threads equal with the number of cores (assuming that tasks do not block), a blocked task leads to losing a core momentarily.

Although the framework documentation mentions this restriction on using I/O operations, it does not enforce it. Our analysis warns the user when the loops intended for parallelization invoke I/O operations. As it visits the CFG for the parallel sections, the analysis catches any invokations of methods from a black-list containing the standard Java I/O classes.

5.4 Putting it all together

Next we put all the pieces together and show how the analysis detects conflicting updates for the program in Fig. 7.

The analysis starts from the entry method, `compute`. In the beginning, the set of shared objects contains only `this`. The instruction at line 22 instantiates our target array. Although this is a definition in a sequential section (which normally marks a shared object), since this instruction writes to our target array, the analysis does not mark it as shared.

Line 25 lies within a parallel section (marked from now on with [[ ]]). Since this line invokes method `createRandom`, the analysis visits this method using the steps described in the interprocedural analysis. Since this method is static (i.e., does not have an object receiver), and it does not have any parameters, there is no binding of arguments. This method creates a new object, i.e., not shared, (line 7), that is bound to `bodies[i]` upon return.

Next, let's examine line 28, which invokes method `moveAll` (referred to as the target method). Notice that this instruction is not in a parallel section. We will now follow the algorithm described for the interprocedural analysis. Since the receiver of the target method is shared (the `this` object from S), the object `this` of the target method is added to S_m. We mark this transfer of sharing through the underlining of `this` at line 28, and respectively at line 39 (middle column). Since the actual argument, `bodies` is not shared, the formal argument `arr` is not added to S_m.

Now we will follow the data-flow analysis through the method `moveAll`. At line 41, there is another method invocation, `moveBy`, in a parallel section. The receiver, `arr[i]`, of this method is not shared, so `this` in method `moveBy` of class `Particle` is not shared (i.e., not added to `moveBy`'s S_m). The analysis then follows through method `moveBy` (line 11). Since nothing is shared here, method `moveBy` does not propagate any sharing.

Upon the return from `moveAll` in line 44, the analysis propagates any changes into the set of shared objects back to the calling context in method `compute`. In this case, since there were no changes, there is nothing to propagate back to S.

Next, the analysis reaches line 30. We explained lines 30–34 in section 5.2.1. Notice that S expands with the objects `cropper` and `bodies`.

The analysis reaches the instruction on line 36 where method `moveAll` is invoked again. The receiver `this` is shared so the target method's `this` is added to S_m (like we showed at the previous invocation). Additionaly, as this time the actual parameter `bodies` is shared, so the formal parameter `arr` is added to S_m.

Next, the analysis checks whether there is a memoization of the data-flow analysis for method `moveAll` with the input set of shared objects $S_m = \{this, arr\}$. As the method was only previously analyzed for the input set $S_m = \{this\}$ (the invocation at line 28), the analysis cannot use any memoized results so it needs to analyze the method again for the new input set.

The analysis revisits line 41. The receiver for `moveBy` invocation, `arr`, is now shared, so `this` is added to `moveBy`'s S_m. Again, since `moveBy` is only memoized for a empty set of shared objects, the analysis revisits the method for the new input set ({`this`}). The analysis reaches line 13. This line contains a shared object (`this`) as the target of a field update. Using the detection function from Fig. 8, the analysis reports the correct data race.

6. EVALUATION

6.1 Research Questions

To evaluate the effectiveness of ReLooper, we answer the following research questions:

- **Q1:** Does the analysis find safety problems? Is it fast?
- **Q2:** What is the rewriting effort? How many changes does ReLooper automatically perform?
- **Q3:** What is the speedup of the refactored code?

These questions answer a higher-level question, is ReLooper useful, from different points of view. The safety and the rewriting questions measure whether ReLooper reduces the burden of parallelizing sequential loops. The speedup question measures whether it is worth to parallelize those loops.

6.2 Methodology

To answer these questions, we used ReLooper to parallelize computationally intensive loops from real programs. We use both programs that are traditionally used as benchmarks in the parallel programming literature, as well as programs that we or others have previously developed.

Table 2 lists the programs that we used. They range from a few thousand to a hundred thousand non-blank, non-comment LOC. BarnesHut, MonteCarlo, and Em3d are part of the JOlden benchmark. BarnesHut computes the force interactions for `N` bodies. MonteCarlo is a financial simulation of stock market. Em3d computes the electromagnetic field propagation in three dimensions. POSTagger and Coref are two natural language processing applications developed in the cognitive group at the University of Illinois. POSTagger tags each word with its part of speech, and Coref finds the words in a text that refer to the same entity. Lucene is a text search engine library.

To find the computationally intensive loops, we profiled the programs, or read the program documentation. Then we used ReLooper to convert the `array` or `Vector` used in the loops into a `ParallelArray`. ReLooper raised safety warnings in programs. We checked whether the reported problems were genuine, then we fixed those problems by making those parts thread-safe (e.g., by using the `Atmomic*` classes from the Java's concurrency package). Then we ran ReLooper once more to parallelize the safe loops. Finally, we ran the parallel programs and confirmed that they were producing the same results as the sequential programs, and reported the speedup of the parallel parts.

6.3 Results

6.3.1 *Does the analysis find safety problems?*

The analysis columns in Table 2 presents the results of the analysis. The Warnings column shows how many warnings ReLooper raised, and how many of these were genuine warnings. After manually analyzing the code, we found that the warnings in most of the programs were genuine, and ReLooper did not miss any races (no false negatives). The only false positives were given in the Coref case study, and were due to the field-insensitiveness of our analysis.

There were two kinds of warnings that ReLooper raised: (i) warnings about conflicting memory accesses, and (ii) warnings about I/O operations.

We fixed the problems reported in the first category. Sometimes, this required changing code very far away from the code in the loops. For example, POSTagger was using a machine learning classifier that was writing to static fields. The write happened in a method four levels down on the call stack from the method invoked in the loop. Analyzing all this code manually is a considerable effort. We fixed the reported problem by replacing the global state (static fields) with thread-local state (fields encapsulated in `ThreadLocal` utility).

With respect to the I/O warnings, we traced the problem and found that indeed, the parallel loops were performing I/O operations in most case studies. We heeded the advice given by ReLooper and did not parallelize those loops.

Not only is ReLooper effective in finding problems, but it is efficient too. The analysis runs fast enough for an interactive tool. In the worst case, in the POSTagger program, it took 25 seconds, but the amount of code visited starting from the parallel loop was large (it contained many machine learning components: the classifier, feature extraction, etc.).

6.3.2 *What is the rewriting effort?*

The Transformation section in Table 2 shows the amount of code that ReLooper changed. For each case study, the changes are reported for one single refactoring (with the exception of Em3d where we applied two refactorings). The LOC changed represent the total of lines deleted, added, or updated during the refactoring (they do not include the changes required to fix the safety warnings). The next columns show how many of the loops that were iterating over the target `array` or `Vector` were parallelized, and how many loops were left sequential. The latter are loops that have I/O operations, or had race conditions that we could not solve.

6.3.3 *What is the speedup?*

The Speedup section in Table 2 shows the speedup of the refactored loops reported against the original sequential loops. We ran the parallel code on a dual-core laptop. The first column shows the relative performance of the refactored code when running on one single processor. As seen, in most cases, the overhead of the parallel code is small. In some cases, the slow down is due to the synchronization primitives that we added to fix the conflicting memory updates. When running on two cores, we get some speedup, without doing any performance tuning.

7. RELATED WORK

Safety analysis for loop parallelism.

Parallelizing loops has received significant interest for scientific computation, initially for Fortran programs. However, much of this work is done in the context of numerical computation on scalar arrays and does not deal with the problems posed by sharing heap-allocated array elements. More recently, in the context of Java, [1] identifies alias-free array regions for an optimizing compiler, but only for one-dimensional scalar arrays. In [11], aliasing of array objects is treated using SSA form and index partitioning, but in an intraprocedural context only.

Our approach is different in purpose from the above-cited works, which perform full program analysis to find loops that can be safely and automatically parallelized by a compiler. Instead, we provide a tool that attempts to parallelize loops explicitly chosen by the programmer, employing a demand-driven analysis to signal problems that the human may have missed. Thus, our stated analysis goal is to quickly and correctly identify dependencies for most cases occurring in practice. Although one issue we flag is object sharing, our purpose is not to do a full-fledged alias or shape analysis, for which there is a significant body of research.

Closer to our work is [16], which presents a loop-based dependence analysis using an points-to mapping that is element- and instance-sensitive. Complex data structures track assignments at arbitrary field depth. However, interprocedural analysis and destructive updates are not implemented and sketched as extensions only. Marron et al. [8] addresses the same problem of proving array entries disjoint, using an abstract heap with refined relations of aliasing, connectivity and interference between references. The analysis is presented in an intraprocedural context, and results are reported for small/medium size Java programs.

One of our analyses performs race detection, though we are concerned with *potential* races that would occur with parallelization, and not with misplaced locking. Of the many papers on static race detection, [10] is close in its treatment of complex object relations. It employs a k-object sensitive alias analysis [9] and an escape analysis to increase the precision of its findings. An earlier race detection approach with similar focus is [2].

Loop parallelization has become highly relevant for dynamic compilation; however, most analyses are highly conservative, e.g., cannot handle loops involving method calls or have restrictive conditions on object referenes.

Refactoring for parallelism.

The earliest work on interactive tools for parallelization stemmed from the Fortran community, and it targets loop parallelization. Interactive tools like PFC [4], ParaScope [5], and SUIF Explorer [7] rely on the user to specify what loops to interchange, align, replicate, or expand. ParaScope and

	Size	Analysis				Transformation			SpeedUp	
	SLOC	Warnings		#Analyzed	Time	Changed	#Loops		1-core	2-core
		Mem(Genuine)	IO	Methods	[sec]	LOC	Parallel	Seq		
POSTagger	35810	4(4)	8	354	25	12	1	1	0.97	1.8
Coref	117660	14(0)	6	257	21	16	1	2	0.97	1.32
MonteCarlo	1127	3(3)	1	83	8.6	15	1	1	0.99	1.42
Barnes-Hut	540	2(2)	0	14	1.7	13	1	1	0.98	1.7
Em3d	189	1(1)	0	22	6.1	52	6	0	0.98	1.35
Lucene	51483	14(14)	0	137	10	44	2	6	0.99	1.96

Table 2: Case studies of using ReLooper on different projects. The size of each project is given in non-comment, non-blank LOC. The Analysis columns show the number of memory and IO warnings that ReLooper raises, how many of these were genuine warnings, the number of analyzed methods, and the running time of the analysis. The Transformation columns show the total number of LOC changed, the number of parallelized loops, and the number of loops left sequential. The Speedup column shows the speedup of the refactored code, relative to the original code.

SUIF Explorer visually display the data dependences. The user must either determine that each loop dependence shown is not valid (due to conservative analysis), or transform a loop to eliminate valid dependences.

More recently, in our previous work [3] on retrofitting parallelism via introducing concurrent libraries we present refactorings that convert primitive types to thread-safe data type (e.g., `int` to `AtomicInteger`), or refactorings for task parallelism. The work presented in this paper expands the toolset with data parallelism.

Wloka et al. [15] present an automated refactoring for making code reentrant. This refactoring changes global data (stored in static fields) into thread-local data. We have manually performed this refactoring to eliminate some of the writes to shared objects pointed by ReLooper in the evaluation programs. The refactoring for reentrancy can be seen as an enabling refactoring for many other refactorings, including the one we present here.

8. CONCLUSIONS

There are two ways in which ReLooper helps Java programmers who want to parallelize their programs by using `ParallelArray`: (i) it helps them discover when parallelizing a loop is unsafe and (ii) it performs the messy conversion of the loop, selecting a good operator from the 132 that come with `ParallelArray`. Just as `ParallelArray` is not the only class in the Java concurrent libraries, ReLooper is not the only tool needed to parallelize Java programs. However, it works well for the kinds of problems for which `ParallelArray` is most suitable, and the other kinds of problems can be solved by other refactoring tools.

There are libraries for other languages that are similar to Java's concurrent library, such as Intel's Theaded Building Blocks, a C++ library. A tool like ReLooper would be useful for programmers who want to use those libraries, as well. There are plans to add closures to C++ and Java. This will make loop conversion easier, but it will not make it any easier to detect when parallelizing a loop is unsafe. Even if the actual conversion of a loop is easy enough to do by hand, automating the analysis of whether a refactoring is safe makes it easier for a programmer to refactor code safely. Fast analysis is as important as fast transformations.

9. ACKNOWLEDGMENTS

This work is partially funded by Microsoft and Intel through the UPCRC Center at Illinois, and a DOE grant ER25752. Cosmin and Mihai did a part of this work as undergraduate summer interns at the Information Trust Institute at the University of Illinois.

We thank John Brant, Fredrik Kjoelstad, Yun Young Lee, Brett Daniel, Milos Gligoric, Jeff Overbey, and Stas Negara for feedback on earlier drafts of this manuscript.

10. REFERENCES

[1] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. Moreira. Automatic loop transformations and parallelization for Java. In *International Conference on Supercomputing*, 2000.

[2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[3] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[4] J.R.Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In *Supercomputers: Design and Applications*, pages 186–205, 1984.

[5] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the parascope editor. In *ICS*, pages 433–447, 1991.

[6] D. Lea. ParallelArray package extra166y. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`, 2009.

[7] S.-W. Liao, A. Diwan, J. Robert P. Bosch, A. Ghuloum, and M. S. Lam. Suif explorer: an interactive and interprocedural parallelizer. *SIGPLAN Not.*, 34(8):37–48, 1999.

[8] M. Marron, M. Méndez-Lojo, M. V. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, pages 43–49, 2008.

[9] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOPLAS*, 14(1), Jan. 2005.

[10] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[11] V. Sarkar and S. Fink. Efficient dependence analysis for Java arrays. In *Euro-Par*, 2001.

[12] Threading Building Blocks.
`http://www.threadingbuildingblocks.org/`.

[13] Task Parallel Library. `http://research.microsoft.com/en-us/projects/tpl`.

[14] WALA: T. J. Watson Libraries for Analysis.
`http://wala.sf.net`.

[15] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *ESEC/SIGSOFT FSE*, pages 173–182, 2009.

[16] P. Wu, P. Feautrier, D. A. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. In *ICS*, 2002.