Refactoring Sequential Java Code for Concurrency via Concurrent Libraries

Danny Dig, John Marrero, Michael D. Ernst
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
{dannydig,marrero,mernst}@csail.mit.edu

Abstract

Parallelizing existing sequential programs to run efficiently on multicores is hard. The Java 5 package `java.util.concurrent` *(*`j.u.c.`*) supports writing concurrent programs: much of the complexity of writing thread-safe and scalable programs is hidden in the library. To use this package, programmers still need to reengineer existing code. This is* tedious *because it requires changing many lines of code, is* error-prone *because programmers can use the wrong APIs, and is* omission-prone *because programmers can miss opportunities to use the enhanced APIs.*

This paper presents our tool, CONCURRENCER*, that enables programmers to refactor sequential code into parallel code that uses three* `j.u.c.` *concurrent utilities.* CONCURRENCER *does not require any program annotations. Its transformations span multiple, non-adjacent, program statements. A find-and-replace tool can not perform such transformations, which require program analysis. Empirical evaluation shows that* CONCURRENCER *refactors code effectively:* CONCURRENCER *correctly identifies and applies transformations that some open-source developers overlooked, and the converted code exhibits good speedup.*

1 Introduction

Users expect that each new generation of computers runs their programs faster than the previous generation. The computing hardware industry's shift to multicore processors demands that programmers find and exploit parallelism in their programs, if they want to reap the same performance benefits as in the past.

It is arguably easier to design a program with concurrency in mind than to retrofit concurrency later [6, 10]. However, most desktop programs were not designed to be concurrent, so programmers have to refactor existing sequential programs for concurrency. It is easier to retrofit concurrency than to rewrite, and retrofitting is often possible.

The dominant paradigm for concurrency in desktop programs is multithreaded programs where shared-memory accesses are protected with locks. However, programming with locks is error-prone: too many locks can slow down or even deadlock an application, while too few locks result in data races.

Java 5's `java.util.concurrent` (`j.u.c.`) package supports writing concurrent programs. Its `Atomic*` classes offer thread-safe, lock-free programming over single variables. Its thread-safe abstract data types (e.g., `ConcurrentHashMap`) are optimized for scalability.

Java 7 will contain the *ForkJoinTask* framework [8, 11] for fine-grained parallelism. Many computationally-intensive problems take the form of recursive *divide-and-conquer*. Classic examples include sorting (e.g., mergesort, quicksort), searching, and many data structure or image processing algorithms. Divide-and-conquer algorithms are good candidates for parallelization since the subproblems can be solved in parallel.

In order to benefit from Java's concurrent utilities and frameworks, the Java programmer needs to refactor existing code. This is *tedious* because it requires changing many lines of code. For example, the developers of six widely used open-source projects changed 1019 lines when converting to use `AtomicInteger` and `ConcurrentHashMap`. Second, manual refactoring is *error-prone* because the programmer can choose the wrong APIs among slightly similar APIs. In the above-mentioned projects, the programmers four times mistakenly used `getAndIncrement` API methods instead of `incrementAndGet`, which can result in off-by-one values. Third, manual refactoring is *omission-prone* because the programmer can miss opportunities to use the new, more efficient API methods. In the same projects, programmers missed 41 such opportunities.

This paper presents our approach for incrementally retrofitting parallelism through a series of behavior-preserving program transformations, namely refactorings. Our tool, CONCURRENCER, enables Java programmers to refactor their sequential programs to use `j.u.c.` utilities: the programmer selects shared data and a target refactoring, and

Concurrencer analyzes all accesses to the shared data and applies the transformation. Ultimately, it is the programmer's responsibility to identify all shared data and target it with the refactorings.

Currently, Concurrencer supports three refactorings: (i) Convert Int to AtomicInteger, (ii) Convert HashMap to ConcurrentHashMap, and (iii) Convert Recursion to ForkJoinTask. Although these are not all the refactorings that one needs for parallelization, the first two refactorings are among the most commonly used in practice, as evidenced by our study [3] of how open-source developers parallelized five projects. These three refactorings are are a proof-of-concept for the toolset that one needs for parallelization.

The first refactoring, Convert Int to AtomicInteger, enables a programmer to convert an `int` field to an `AtomicInteger`, a utility class that encapsulates an `int` value. The encapsulated field can be safely accessed from multiple threads, without requiring any synchronization code. Our refactoring replaces all field accesses with calls to `AtomicInteger`'s thread-safe APIs. For example, it replaces expression `f = f + 3` with `f.addAndGet(3)` which executes atomically.

The second refactoring, Convert HashMap to ConcurrentHashMap, enables a programmer to convert a `HashMap` field to `ConcurrentHashMap`, a thread-safe, highly scalable implementation for hash maps. Our refactoring replaces map updates with calls to the APIs provided by `ConcurrentHashMap`. For example, a common update operation is (i) check whether a map contains a certain key, (ii) if not present, create the value object, and (iii) place the value in the map. Concurrencer replaces such an updating pattern with a call to `ConcurrentHashMap`'s `putIfAbsent` which *atomically* executes the update, without locking the entire map.

The third refactoring, Convert Recursion to ForkJoinTask, enables a programmer to convert a sequential divide-and-conquer algorithm to a parallel algorithm. The parallel algorithm solves the subproblems in parallel using the ForkJoinTask framework. Using the skeleton of the sequential algorithm, Concurrencer extracts the sequential computation into tasks that run in parallel and dispatches these tasks to the ForkJoinTask framework.

Typically a user would first make a program thread-safe, i.e., the program has the same semantics as the sequential program even when executed under multiple threads, and then make the program run concurrently under multiple threads. Concurrencer supports both kinds of refactorings. The first two refactorings are "enabling transformations" that make a program thread-safe. The third refactoring makes a sequential program run concurrently.

The transformations performed by these refactorings require matching certain code patterns which can span several non-adjacent program statements, and they require program analysis which uses data-flow information. Such transformations can not be safely executed by find-and-replace.

This paper makes the following contributions:

- **Approach.** We present an approach for retrofitting parallelism into sequential applications through automated, but human-initiated, program transformations. Since the programmer is expert in the problem domain, she is the one most qualified to choose the code and the program transformation for parallelizing the code.
- **Tool.** We implemented three transformations for using thread-safe, highly scalable concurrent utilities and frameworks. Our tool, Concurrencer, is conveniently integrated within Eclipse's refactoring engine. Since Concurrencer is neither complete, nor sound, it can not guarantee absolute thread-safety. Nevertheless, it is safer and faster than making all the changes by hand. Concurrencer can be downloaded from: `http://refactoring.info/tools/Concurrencer`
- **Empirical Results.** We used Concurrencer to refactor the same code that the open-source developers of 6 popular projects converted to `AtomicInteger` and `ConcurrentHashMap`. By comparing the manually vs. automatically refactored output, we found that Concurrencer applied all the transformations that the developers applied. Even more, Concurrencer avoided the errors which the open-source developer *committed*, and Concurrencer identified and applied some transformations that the open-source developers *omitted*. We also used Concurrencer to parallelize 6 divide-and-conquer algorithms. The parallelized algorithms perform well and exhibit good speedup. These experiences show that Concurrencer is useful.

2 Convert Int to AtomicInteger

2.1 AtomicInteger in Java

The Java 5 class library offers a package `j.u.c.atomic` that supports *lock-free* programming on *single* variables.

The package contains wrapper classes over primitive variables, for example, an `AtomicInteger` wraps an `int` value. The main advantage is that update operations execute atomically, without blocking. Internally, `AtomicInteger` employs efficient machine-level atomic instructions like *Compare-and-Swap* that are available on contemporary processors. Using `AtomicInteger`, the programmer gets both *thread-safety* (built into the `Atomic` classes) and *scalability* (the lock-free updates eliminate lock-contention under heavy accesses [6]).

2.2 Code Transformations

A programmer who wanted to use Concurrencer to make all accesses to an `int` thread-safe would start by selecting

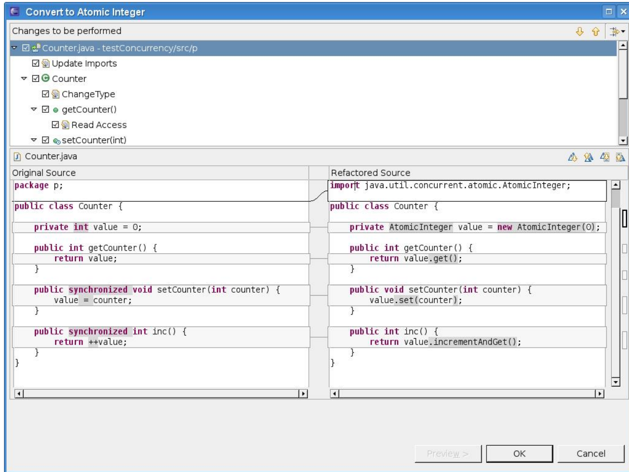

Figure 1: Using CONCURRENCER to convert an `int` to `AtomicInteger` in Apache Tomcat. The screenshot shows a preview of the changes.

the field and invoking the CONVERT INT TO ATOMICINTEGER refactoring. CONCURRENCER changes the declaration type of the `int` field to `AtomicInteger` and replaces all field updates with their equivalent atomic API methods in `AtomicInteger`.

Figure 1 shows how CONCURRENCER refactors some code from Apache Tomcat.

Initialization. Because the refactored `value` field is an `AtomicInteger` object, CONCURRENCER initializes it in the field initializer (otherwise a `NullPointerException` is thrown the first time that a method is invoked on `value`). CONCURRENCER uses the field initializer expression or the implicit expression '0'.

Field Accesses. Table 1 shows how CONCURRENCER replaces field accesses with `AtomicInteger`'s *atomic* APIs. `AtomicInteger` only provides APIs for replacing infix expressions involving the `+` operator. The last rows show that CONCURRENCER converts a subtract expression into an addition expression. If the program contains updates involving other operators (e.g., multiplication, division), then CONCURRENCER warns the user that these update expressions cannot be made thread-safe using `AtomicInteger`. The reason is that `AtomicInteger` has no atomic APIs for these updates.

Synchronization. CONCURRENCER converts both a sequential program into one which is thread-safe, and also an already thread-safe program into one which is *more* scalable. If the original code contains `synchronized` accesses to the `int` field, CONCURRENCER tries to remove the synchronization since this becomes superfluous after the conversion to `AtomicInteger` (thread-safety is built into the `AtomicInteger`).

Conservatively, CONCURRENCER only removes the lock if (i) the refactored code corresponding to the original synchronized block contains only one call to `AtomicInteger`'s

Access	int	AtomicInteger
Read	f	f.get()
Write	f = e	f.set(e)
Cond. Write	if (f==e) f=e_1	f.compareAndSet(e,e_1)
Prefix Inc.	++f	f.incrementAndGet()
Postfix Inc.	f++	f.getAndIncrement()
Infix Add	f = f + e	f.addAndGet(e)
Add	f += e	f.addAndGet(e)
Prefix Dec.	--f	f.decrementAndGet()
Postfix Dec.	f--	f.getAndDecrement()
Infix Sub.	f = f - e	f.addAndGet(-e)
Subtract	f -= e	f.addAndGet(-e)

Table 1: CONCURRENCER replaces accesses to field `f` with calls to `AtomicInteger` APIs (`e` denotes an expression).

APIs, and (ii) the original synchronization block accesses one single field. The first condition ensures that a thread interleaving does not occur between two consecutive calls to atomic APIs. The latter ensures the multivariable invariants are still preserved, since `AtomicInteger` ensures thread-safety for only one single field.

For example, CONCURRENCER removes the synchronization in the code fragment below:

```
synchronized(lock){
  value = value + 3;
}
```

but does not remove synchronization for the code fragment below:

```
synchronized(lock){
  value = value + 3;
  ..............
  value ++;
}
```

neither for the code fragment below:

```
synchronized(lock){
  value = value + 3;
  anotherField ++;
}
```

3 Convert HashMap to ConcurrentHashMap

3.1 ConcurrentHashMap in Java

The `j.u.c.` package contains several concurrent collection classes. `ConcurrentHashMap` is a thread-safe implementation of `HashMap`.

Before the introduction of `j.u.c.`, a programmer could create a thread-safe `HashMap` using a synchronized wrapper over a `HashMap` (e.g., `Collections.synchronizedMap(aMap)`). The synchronized `HashMap` achieves its thread-safety by protecting *all* accesses to the `map` with a *common* lock. This results in poor scalability

when multiple threads try to access different parts of the map simultaneously, since they contend for the lock.

`ConcurrentHashMap` uses a more scalable locking strategy. *All* readers run concurrently, and *lock-striping* allows a *limited* number of writers to update the map concurrently. The `j.u.c.` implementation uses N locks (the default value is 16), each of them guarding a part of the hash buckets. Assuming that the hash function spreads the values well, and that keys are accessed randomly, this reduces the contention for any given lock by a factor of N.

`ConcurrentHashMap` includes the API methods offered by `HashMap`. In addition, it contains three new APIs `putIfAbsent(key, value)`, `replace(key, oldValue, newValue)`, and a conditional `remove(key, value)`. Each of these new APIs:

- supersedes several calls to `HashMap` operations, and
- executes atomically.

For example, `putIfAbsent` (1) checks whether the map contains a given key, and (2) if absent, inserts the $\langle key, value\rangle$ entry.

Replacing a synchronized `HashMap` with `ConcurrentHashMap` offers dramatic scalability improvements [6].

3.2 Code Transformations

To make all accesses to an `HashMap` field thread-safe, a pogrammer would select the field and invoke the CONVERT HASHMAP TO CONCURRENTHASHMAP refactoring.

Initialization and Accesses. CONCURRENCER changes the declaration and the initialization of the field. Because `HashMap` and `ConcurrentHashMap` implement the same interface (`Map`), initialization and map accesses remain largely the same.

Map Updates. CONCURRENCER detects update code patterns and replaces them with the appropriate `ConcurrentHashMap` API method.

Figure 2 shows the *basic* update patterns that CONCURRENCER replaces with `map.putIfAbsent(key, value)`. The patterns have a similar structure: (1) check whether the map contains a certain key, and (2) depending on the result, invoke `put(key, value)`. This structure has small variations. For example, the check can invoke `containsKey` (like in (i) and (ii)), or `get` (like in (iii) and (iv)). A temporary variable might hold the result of the check (like in (ii) and (iv)).

Before invoking `putIfAbsent`, the $value$ to be inserted must be **available**. If the $value$ to be placed in the $\langle key, value\rangle$ map entry is simply created by invoking a constructor (e.g., `map.put(key, new ClassX())`), then, in the refactored code, CONCURRENCER constructs the $value$ similarly (e.g., `map.putIfAbsent(key, new ClassX())`). However, the creational code for $value$ may span multiple statements, and the pattern elements may not be on adjacent statements.

(i)
```
if (!map.containsKey(key))
   map.put(key, value);
```

(ii)
```
boolean keyExists = map.containsKey(key);
if (!keyExists)
   map.put(key, value);
```

(iii)
```
if (map.get(key) == null)
   map.put(key, value);
```

(iv)
```
Object testValue = map.get(key);
if (testValue == null)
   map.put(key, value);
```

Figure 2: Basic code patterns that are replaced with `map.putIfAbsent(key, value)`.

parameters:
  Statements: $BEFORE_PUT$, $AFTER_PUT$
  variables: $testValue$, $newValue$

1 **if** $!isReadIn(AFTER_PUT, testValue)$ **then**
2   $deleteVariable(testValue)$;
3 **else**
4   //$testValue$ is read later, do not delete it
5   **if** $isWrittenIn(BEFORE_PUT, testValue)$
    $\wedge\ return(putIfAbsent()) == success$ **then**
6     $testValue \leftarrow newValue$

Figure 3: The algorithm for deciding whether to delete from the refactored code the $testValue$ variable (i.e., the variable that holds the presumed $value$ associated with a key). $BEFORE_PUT$ denotes statements inside the if statement that preced the call to `put`. $AFTER_PUT$ denotes statements that succeed the call to `put` (both inside and outside the if statement). If $testValue$ can not be deleted, CONCURRENCER generates the code in lines 6–7 that conditionally reassign $testValue$.

Consider the example in Figure 4, whose left side highlights an example of pattern (iv) in Figure 2. On the right side of Figure 4, CONCURRENCER has extracted the creational code (lines 9–12 on the left side) into a creational method (`createTimeZoneList`), calls it and stores the result in the variable `newTZList` (from hereon referred as the $newValue$ variable), and then passes the $newValue$ as the argument to `putIfAbsent`. Since the variable `timeZoneList` (from hereon referred as $testValue$) is no longer passed to `putIfAbsent`, can it be deleted?

Figure 3 gives the analysis to determine whether to delete the $testValue$ variable. CONCURRENCER checks whether the $testValue$ variable is not *live* (i.e., not in use) after the call to `put` (from hereon these statements are referred as $AFTER_PUT$, e.g., lines 15–19 in Fig. 4), thus it can be deleted. Otherwise, if the $testValue$ is reassigned in the $BEFORE_PUT$ statements (i.e., inside the pattern's if statement, but before calling `map.put` (e.g., lines 9–12

```java
1 // before refactoring
2 private HashMap<Locale, String[]> timeZoneLists;
3 private String[] timeZoneIds;
4
5 public String[] getTimeZoneList() {
6   Locale locale = JiveGlobals.getLocale();

7   String[] timeZoneList = timeZoneLists.get(locale);
8   if (timeZoneList == null) {

9     timeZoneList = new String[timeZoneIds.length];
10    for (int i = 0; i < timeZoneList.length; i++) {
11      . . . // populate timeZoneList
12    }
13

14    timeZoneLists.put(locale, timeZoneList);

15    . . . // Code AFTER_PUT
16  }
17
18  return timeZoneList;
19 }
```

```java
// after refactoring
private ConcurrentHashMap<Locale, String[]> timeZoneLists;
private String[] timeZoneIds;

public String[] getTimeZoneList() {
  Locale locale = JiveGlobals.getLocale();

  String[] timeZoneList = timeZoneLists.get(locale);
  String[] newTZList = createTimeZoneList(locale);
  if (timeZoneLists.putIfAbsent(locale,newTZList)==null){
    timeZoneList = newTZList;
    . . . // Code AFTER_PUT
  }
  return timeZoneList;
}

private String[] createTimeZoneList(Locale locale) {
  String[] timeZoneList;
  timeZoneList = new String[timeZoneIds.length];
  for (int i = 0; i < timeZoneList.length; i++) {
    . . . // populate timeZoneList
  }
  return timeZoneList;
}
```

Figure 4: The user selects the `timeZoneLists` `HashMap` field (line 2) to be made thread-safe, and Concurrencer performs all the transformations. The figure shows an example from Zimbra where the `putIfAbsent` pattern requires invoking a creational method to hold the value to be placed in the map. The pattern elements in the original code that correspond to Figure 2.(iv) have a gray background. The changes in the refactored code are underlined.

in Fig. 4)), in the refactored code CONCURRENCER assigns $newValue$ to $testValue$. To preserve the original semantics (i.e., $testValue$ is reassigned only if the $\langle key, value\rangle$ is placed in the map), the generated code uses the return status of `putIfAbsent` (null denotes that the call succeeded) to determine whether to reassign the $testValue$.

Functions $isReadIn$ and $isWrittenIn$ lexically analyze the statements for read or write accesses to the $testValue$ variable. CONCURRENCER's implementation of this analysis is intraprocedural and does not handle aliasing.

In the refactored code, the creational code is executed regardless of whether the $newValue$ is placed into the map. This can result in (i) a semantical change if the creational code has side effects and (ii) creating unnecessary objects.

CONCURRENCER warns the user if the creational method has side effects. Our analysis is an intraprocedural MOD Analysis [15] that ignores aliasing. The analysis checks whether the creational method *locally modifies* any fields.

To minimize the cost of creating unnecessary objects, CONCURRENCER could use the "double-checked lazy initialization" pattern [4]: first test whether the map does not contain the key (in an unsafe manner), and only if the test succeeds invoke the creational code followed by a thread-safe `putIfAbsent`. Although this strategy does not completely eliminate the chance of creating unnecessary objects, it minimizes the window of opportunity for creating unnecessary objects, while still ensuring thread-safety (in the end, checking and possibly placing the new value in the map is thread-safe, guaranteed by `putIfAbsent`).

Synchronization. If the original method contained synchronization locks around map updates, CONCURRENCER removes them when they are superfluous (`ConcurrentHashMap` has thread-safety built in). CONCURRENCER uses similar checks with the ones used when removing locks from `AtomicInteger`.

4 Convert Recursion to ForkJoinTask

4.1 ForkJoinTask Framework in Java 7

Java 7 will contain a framework, ForkJoinTask, for fine-grained parallelism in computationally-intensive problems. Divide-and-conquer algorithms are natural candidates for such parallelization when the recursion tasks are completely independent, i.e., they operate on different parts of the data or they solve different subproblems. Many recursive divide-and-conquer algorithms display such properties, even though they were never designed with parallelism in mind. Furthermore, static analyses (e.g., [13]) can determine whether there is any data dependency between the recursive tasks, e.g., the recursive tasks write within the same ranges of an array.

Fig. 5 shows the sequential and parallel versions of a divide-and-conquer algorithm. In the parallel version, if the problem size is smaller than a threshold, the problem is solved using the sequential algorithm. Otherwise, the problem is split into independent parts, these are solved in parallel, then the algorithm waits for all computations to finish and composes the result from the subresults.

```
// Sequential version

solve (Problem problem) {
  if (problem.size <= BASE_CASE )
    solve problem directly
  else {
    split problem into independent tasks

    solve each task


    compose result from subresults
  }
}
```

```
// Parallel version

solve (Problem problem) {
  if (problem.size <= SEQ_THRESHOLD )
    solve problem SEQUENTIALLY
  else {
    split problem into independent tasks
    IN_PARALLEL{ //fork
      solve each task
    }
    wait for all tasks to complete //join
    compose result from subresults
  }
}
```

Figure 5: Sequential and parallel pseudocode for a divide-and-conquer algorithm.

Since threads have high overhead (creating, scheduling, destroying) which might overwhelm the useful computation, Java 7 introduces `ForkJoinTask`, a lighter-weight thread-like entity. A large number of such tasks may be hosted by a pool containing a small number of actual threads. The framework schedules the tasks effectively and keeps all cores busy with useful computation.

The most important API methods in `ForkJoinTask` are: `fork()` which spawns the execution of a task in parallel, `join()` which waits for the current task to finish, `invokeAll(Tasks)` which is syntactic sugar for forking the given tasks and then waiting for them to finish, and `compute()`, which encapsulates the main computation performed by the task.

`ForkJoinTask` has several subclasses for different patterns of computation. `RecursiveAction` is the proper choice for the recursive tasks used in divide-and-conquer computations. The framework also defines `ForkJoinExecutor`, an object that executes `ForkJoinTask` computations using a pool of worker threads.

4.2 Code Transformations

CONCURRENCER converts a recursive divide-and-conquer algorithm to one that runs in parallel using the ForkJoinTask framework. The programmer needs only select the divide-and-conquer method and supply the $SEQ_THRESHOLD$ parameter that determines when to run the sequential version of the algorithm. Using this user-supplied information, CONCURRENCER automatically performs all transformations.

We made a design choice to keep the original interface of the recursive method unchanged, so that an outside client would still invoke the method as before. The fact that the refactored method uses the ForkJoinTask framework is an implementation detail, hidden from the outside client.

We illustrate the transformations that CONCURRENCER performs on a classic merge sort algorithm. The left-hand side of Figure 6 shows the original, sequential version of the merge sort algorithm. The `sort` method takes as input the array to be sorted, and it returns the sorted array. The algorithm starts with the base case (line 27). In the recursive case (lines 29–40), it copies the first half of the array and the second half of the array, sorts both halves, and merges them (code for `merge` not shown).

Creating the ForkJoinTask. CONCURRENCER creates a `RecursiveAction` class (line 16), which is a subclass of `ForkJoinTask`. This class encapsulates the parallel computation of the original recursive method, thus CONCURRENCER names this class by adding the `Impl` suffix to the name of the original recursive method.

Since the `compute` method neither takes any arguments, nor returns a value, `SortImpl` has fields for the input arguments and the result of the computation. The constructor initializes the input fields (line 21).

Implementing the `compute` method. The `compute` method is called by the framework when it executes a `ForkJoinTask`. CONCURRENCER implements this method using the original recursive method as the model for computation. CONCURRENCER performs three main transformations on the original recursive method: (i) it changes the base case of the recursion, (ii) it replaces recursive calls with `RecursiveAction` instantiations, and (iii) it executes the parallel tasks and then gathers the results of the subtasks.

First, CONCURRENCER infers the *base-case* used in the recursion: the base case is a conditional statement which does not contain any recursive calls and which ends up with a return statement. Then CONCURRENCER replaces the base-case conditional expression with the $SEQ_THRESHOLD$ expression provided by the user (line 25). Next, CONCURRENCER replaces the return statement in the base case of the original recursive method with a call to the sequential method (line 26). If the original method returned a value, CONCURRENCER saves this value in the `result` field.

Second, CONCURRENCER replaces the recursive calls with creation of new `RecursiveAction` objects (lines 34, 35). The arguments to the recursive call are passed as arguments to the constructor of the `RecursiveAction`. CONCURRENCER stores the created tasks into local variables `task1` and `task2`.

Third, CONCURRENCER executes the parallel tasks and then assembles the result from the subresults of the tasks. CONCURRENCER calls the `invokeAll` method while passing

```java
// Sequential version

public class MergeSort {

  public int[] sort(int[] whole) {
    if (whole.length == 1) {

      return whole;
    } else {
      int[] left = new int[whole.length / 2];
      System.arraycopy(whole, 0, left, 0, left.length);
      int[] right = new int[whole.length - left.length];
      System.arraycopy(whole, left.length,
                       right, 0, right.length);
      left = sort(left);
      right = sort(right);

      merge(left, right, whole);
      return whole;
    }
  }

  private void merge(int[] left, int[] right,
                     int[] whole) {
    . . . . merge left and right array into whole array
  }
}
```

```java
1  // Parallel version
2
3  import jsr166y.forkjoin.ForkJoinPool;
4  import jsr166y.forkjoin.RecursiveAction;
5
6  public class MergeSort {
7
8    public int[] sort(int[] whole) {
9      int processorCount = Runtime.getRuntime().availableProcessors();
10     ForkJoinPool pool = new ForkJoinPool(processorCount);
11     SortImpl aSortImpl = new SortImpl(whole);
12     pool.invoke(aSortImpl);
13     return aSortImpl.result;
14   }
15
16   private class SortImpl extends RecursiveAction {
17     private int[] whole;
18     private int[] result;
19
20     private SortImpl(int[] whole) {
21       this.whole = whole;
22     }
23
24     protected void compute() {
25       if ((whole.length < 100)) {
26         result = sort(whole);
27         return;
28       } else {
29         int[] left = new int[whole.length / 2];
30         System.arraycopy(whole, 0, left, 0, left.length);
31         int[] right = new int[whole.length - left.length];
32         System.arraycopy(whole, left.length,
33                          right, 0, right.length);
34         SortImpl task1 = new SortImpl(left);
35         SortImpl task2 = new SortImpl(right);
36         invokeAll(task1, task2);
37         left = task1.result;
38         right = task2.result;
39         merge(left, right, whole);
40         result = whole;
41       }
42     }
43
44     private int[] sort(int[] whole) {
45       . . . copy the original, sequential implementation
46     }
47   }
48
49   private void merge(int[] left, int[] right,
50                      int[] whole) {
51     . . . . merge left and right array into whole array
52   }
53 }
```

Figure 6: The programmer selects the divide-and-conquer method and provides the sequential threshold (`whole.length < 100`). Concurrencer converts the sequential divide-and-conquer into a parallel one using the ForkJoinTask framework. The left-hand side shows the sequential version, the right-hand side shows the parallel version (changes are underlined).

the previously created tasks as arguments. Concurrencer places the `invokeAll` method after the last creation of `RecursiveAction` (line 36). Then Concurrencer saves the subresults of the parallel tasks into local variables. If the original recursive method used local variables to store the results of the recursive calls, Concurrencer reuses the same variables (lines 37, 38). Subsequent code can thus use the subresults to assemble the final result (line 39). Lastly, Concurrencer assigns to the `result` field the combined subresults (line 40).

Reimplementing the recursive method. Concurrencer rewrites the implementation of the original recursive method to invoke the ForkJoinTask framework (lines 9–12). Concurrencer creates a new task and initializes it with the array to be sorted, then it passes the task to the executor. `invoke` blocks until the computation finishes, then the sorted array in the `result` field is returned (line 13).

Discussion. Concurrencer handles several variations on how the subresults are combined to form the end result. For example, the subresults of the recursive calls might not be stored in temporary variables, but they might be combined directly in expressions, like in the `fibonacci` function:

```java
return fibonacci(n-1) + fibonacci(n-2).
```

Concurrencer creates and executes the parallel tasks as before, and during the subresult combination phase it uses the same expression to combine the subresults:

```java
result = task1.result + task2.result
```

With respect to where the recursive method stores the result, there can be two kinds of recursive methods: (i) recursive methods that return a value, the result, and (ii) recursive

methods that do not return any value, but they mutate one of the arguments to hold the result of the computation.

Fig. 6 is an example of the first kind of computation. The transformations for recursive methods that mutate one of their arguments to store the result are similar to the ones presented above, even slightly simpler: CONCURRENCER does not generate the code involving the `result` field.

5 Discussion

Similar with other practical refactoring engines, our approach is not complete because it relies on a set of patterns and transformations. Even though they might not cover all possible scenarios, they do cover the most common ones, as illustrated by the evaluation on real-world codebases (see Section 6). Moreover, it is easy to extend these transformation patterns.

Also since the analysis does not handle aliasing and it is intraprocedural, our approach is not sound. However, for the currently supported refactorings, not handling aliasing is not harmful. An `int` field used in CONVERT INT TO ATOMICINTEGER cannot be aliased, while in general programmers do not alias an `HashMap` field used in CONVERT HASHMAP TO CONCURRENTHASHMAP. Even though our approach is neither sound, nor complete, it is still useful. That is, CONCURRENCER saves programmer's time overall.

CONCURRENCER does not remove the user-defined lock for some field accesses (e.g., when the synchronized block protects accesses to 2 fields), while it removes the lock for other accesses that can be safely protected by the `j.u.c.` library. An interleaving can still occur between lock-protected and library-protected accesses (since they are not protected by the same mechanism). If CONCURRENCER cannot remove the user-defined lock for all field accesses, it warns the user, who can decide to cancel the refactoring. Most of the times a user does not have to analyze the output of the refactoring, but only when CONCURRENCER raises warnings.

6 Evaluation

Research Questions. To evaluate the effectiveness of CONCURRENCER, we answered the following questions:

- **Q1:** Is CONCURRENCER useful? More precisely, does it ease the burden of making sequential code thread-safe and of writing parallel code?
- **Q2:** Are the results thread-safe? How does the manually refactored code compare with code refactored with CONCURRENCER in terms of using the correct APIs and identifying all opportunities to replace field accesses with thread-safe API calls?

Refactoring & project	# of refactorings	LOC changed total	LOC changed by CONCURRENCER
Convert Int To AtomicInteger			
MINA	5	21	21
Tomcat	5	26	26
Struts	0	0	0
GlassFish	15	60	60
JaxLib	29	240	240
Zimbra	10	54	54
Convert HashMap To ConcurrentHashMap			
MINA	6	14	14
Tomcat	0	0	0
Struts	6	68	68
GlassFish	14	86	82
JaxLib	7	62	62
Zimbra	44	388	377
Total	**141**	**1019**	**1004**
Convert Recursion to ForkJoinTask			
mergeSort [13]	1	36	36
fibonacci [11]	1	25	25
maxSumConsecutive [11]	1	68	68
matrixMultiply [5, 11, 13]	1	108	108
quickSort (Zimbra)	1	35	35
maxTreeDepth (Eclipse)	1	30	30
Total	**6**	**302**	**302**

Table 2: Case studies of refactorings. The last two columns show the number of lines of code that were changed to perform the refactoring, and how many of those lines can be changed by CONCURRENCER. The remaining changes must be performed manually.

- **Q3:** With respect to running concurrent tasks in parallel, is the refactored code more efficient than the original sequential code?

We evaluated CONCURRENCER's refactorings in two ways. For code that had already been refactored to use Java 5's `AtomicInteger` and `ConcurrentHashMap`, we compared the manual refactoring with what CONCURRENCER would have done. This answers the first two questions. For CONVERT RECURSION TO FORKJOINTASK, we could not find projects using ForkJoinTask, since it is scheduled for Java 7's release. We used CONCURRENCER to refactor six divide-and-conquer algorithms. This answers the first and the third questions.

6.1 Methodology

Setup for CONVERT INT TO ATOMICINTEGER and CONVERT HASHMAP TO CONCURRENTHASHMAP.

Table 2 lists 6 popular, mature open-source projects that use `AtomicInteger` and `ConcurrentHashMap`. We used the head versions from their version control system as of June 1, 2008.

We used CONCURRENCER to refactor *the same* fields that open-source developers refactored to `AtomicInteger` or

ConcurrentHashMap. We compare the code refactored with CONCURRENCER against code refactored by hand. We look at places where the two refactoring outputs differ, and quantify the number of *errors* (i.e., one of the outputs uses the wrong concurrent APIs) and the number of *omissions* (i.e., the refactored output could have used a concurrent API, but it instead uses the obsolete, lock-protected APIs).

For AtomicInteger the projects' version control repository contains both a version with the int field and a later version with the AtomicInteger field, thus we use the version with int as the input for CONCURRENCER. For CONVERT HASHMAP TO CONCURRENTHASHMAP we were not able to find the versions which contained HashMap. It seems that those projects were using ConcurrentHashMap from the first version of the file. In those cases we manually replaced *only* the type declaration of the ConcurrentHashMap field with HashMap; then we ran CONCURRENCER to replace HashMap updates with the thread-safe APIs (putIfAbsent, replace, and delete) in ConcurrentHashMap. For both kinds of refactorings we carefully examined the refactored source code to check whether the open-source programmers or CONCURRENCER ommitted to make some accesses thread-safe.

Setup for CONVERT RECURSION TO FORKJOINTASK.

We used CONCURRENCER to parallelize six divide-and-conquer algorithms. We use two sets of inputs: (i) classic divide-and-conquer algorithms used in others' evaluations [5, 11, 13], and (ii) divide-and-conquer algorithms from real projects.

Table 2 shows the input programs. maxSumConsecutive takes an array of positive and negative numbers and computes the subsequence of consecutive numbers whose sum is maximum. matrixMultiply multiplies two matrices. maxTreeDepth computes the depth of a binary tree.

6.2 Q1: Is CONCURRENCER useful?

The top part of Table 2 shows the number of refactorings that open-source developers performed in the selected real world projects. The penultimate column shows how many lines of code were manually changed during refactoring. Using CONCURRENCER, the developers would have saved editing 1004 lines of code; instead they would have had to only change 15 lines not currently handled by CONCURRENCER. We show one such example at the end of Section 6.3.

The bottom part of Table 2 shows the LOC changed when converting the original recursive algorithm to one that uses the ForkJoinTask framework. To do the manual conversion, it took the first author an average of 30 minutes for each conversion. This includes also the debug time to make the parallelized algorithm work correctly. Using CONCURRENCER, the conversion was both correct and took less than 10 seconds. Doing the conversion with CONCURRENCER saves the programmer from changing 302 LOC.

6.3 Q2: How does manually and automatically refactored code compare?

CONCURRENCER applied all the correct transformations that the open-source developers applied. We noticed several cases where CONCURRENCER outperforms the developers: CONCURRENCER produces the correct code, or it identifies more opportunities for using the new, scalable APIs.

For CONVERT INT TO ATOMICINTEGER, we noticed cases where the developers used the wrong APIs when they refactored by hand. We noticed that developers erroneously replaced infix expressions like ++f with f.getAndIncrement(), which is the equivalent API for the postfix expression f++. They should have replaced ++f with f.incrementAndGet(). Table 3 shows that the open-source developers made 4 such errors, where CONCURRENCER made no error. The erroneous usage of the API can cause an "off-by-one" value if the result is read in the same statement which performs the update. In the case studies, the incremented value is not read in the same statement which performs the update. Nevertheless, this shows the manual conversion is error-prone.

For CONVERT HASHMAP TO CONCURRENTHASHMAP we noticed cases when the open-source developers or CONCURRENCER omitted to use the new atomic putIfAbsent and conditional delete operations, and instead use the old patterns involving synchronized, lock-protected access to put and delete, or leave the code totally unprotected. Even when they used the locks, the refactored code is not thread-safe, since they used different locks than the ones that ConcurrentHashMap's implementation uses. Moreover, the refactored code is non-optimal for these lines of code because it locks the whole map for the duration of update operations. In contrast, ConcurrentHashMap's new APIs offers better scalability because they do not lock the whole map.

Table 4 shows the number of such omissions in the case-study projects. We manually determined all the uses of put or delete that could be replaced with the new putIfAbsent, replace, or conditional delete. We found that the open-source developers missed many opportunities to use the new APIs. This intrigued us, since the studied projects are all developed professionally, and are known to be of high-quality (e.g., Zimbra was acquired by Yahoo, Struts is developed by Apache foundation, GlassFish is developed mainly by SUN). Also, we found several instances when the open-source developers correctly used the new APIs, so they certainly were aware of the new APIs.

We can hypothesize that the open-source developers did not convert to the new APIs because the new APIs would have required creational methods which had side effects. Therefore, we conservatively only count those cases when the creational method is guaranteed not to have side-effects (e.g., the value to be inserted in the map is produced by simply instantiating a Java collection class). Even so, Table 4

	incrementAndGet		decrementAndGet	
	correct usages	erroneous usages	correct usages	erroneous usages
Tomcat	0	1	0	1
MINA	0	1	0	1

Table 3: Human errors in using `AtomicInteger` updates in refactorings performed by open-source developers.

	putIfAbsent			remove		
	potential	omissions		potential	omissions	
	uses	human	tool	uses	human	tool
MINA	0	0	0	0	0	0
Tomcat	0	0	0	0	0	0
Struts	6	1	0	0	0	0
GlassFish	7	3	1	6	5	0
JaxLib	11	2	0	0	0	0
Zimbra	49	27	9	4	3	0
Total	73	33	10	10	8	0

Table 4: Human and CONCURRENCER omissions in using `ConcurrentHashMap`'s `putIfAbsent` and conditional `remove`.

shows that the open-source developers missed several opportunities to use the new APIs. CONCURRENCER missed many fewer opportunities. These are all rare, intricate patterns currently not supported by CONCURRENCER, but they could all be supported by putting more engineering effort in the tool. Below is one such example of potential usage of `putIfAbsent` coming from Zimbra code:

```
private ConcurrentHashMap<String, Component> components;

public void addComponent(String subdomain)
                         throws ComponentException {
  Component existingComponent = components.get(subdomain);
  if (existingComponent != null) {
    throw new ComponentException("Domain already taken");
  }
  components.put(subdomain, component);
}
```

The example above is a rare variation of the pattern iv in Figure 2: the condition in the ifStatement is reversed.

6.4 Q3: What is the speedup of the parallelized algorithms?

Table 5 shows the speedup of the parallelized algorithms ($speedup = time_{seq}/time_{par}$). For the sorting algorithms we use random arrays with 10 million elements. For `fibonacci` we compute the fibonacci value for the number 45. For `maxSumConsecutive` we use an array with 100 million random integers. For `matrixMultiply` we use matrices with 1024x1024 doubles. For `maxTreeDepth` we use a dense tree of depth 50.

7 Related Work

The earliest work on interactive tools for parallelization stemmed from the Fortran community, and it targets

program	**speedup**	
	2 cores	**4 cores**
mergeSort	1.18x	1.6x
fibonacci	1.94x	3.82x
maxSumConsecutive	1.78x	3.16x
matrixMultiply	1.95x	3.77x
quickSort	1.84x	3.12x
maxTreeDepth	1.55x	2.38x
Average	1.7x	2.97x

Table 5: Speedup of the parallelized divide-and-conquer algorithms.

loop parallelization. Interactive tools like PFC [7], ParaScope [9], and SUIF Explorer [12] rely on the user to specify what loops to interchange, align, replicate, or expand. ParaScope and 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.

Freisleben and Kielman [5] present a system that parallelizes divide-and-conquer C programs, similar in spirit to our CONVERT RECURSION TO FORKJOINTASK refactoring. To use their system, a programmer annotates (i) what computations are to be executed in parallel, (ii) the synchronization points after which the results of the subproblems are expected to be available, (iii) the input and output parameters of the recursive function, and (iv) the sequential threshold. The annotated program is preprocessed and transformed into a program which uses message-passing to communicate between the slave processes that execute the subproblems. Unlike their system, CONCURRENCER is not restricted to algorithms that use only two recursive subdivisions of the problem, and CONCURRENCER automatically infers all the parameters of the transformation (except the sequential threshold).

Bik et al. [2] present Javar, a compiler-based, source-to-source restructuring system that uses programmer annotations to indicate parallelization of loops and of recursive algorithms. Javar rewrites the annotated code to run in parallel using multiple threads. Javar's support for parallelizing recursive functions is not optimal: each recursive call forks a new thread, whose overhead can be greater than the useful computation. Unlike Javar, (i) CONCURRENCER does not require any programmer annotations, (ii) the parallel recursion benefits from the efficient scheduling and load-balancing of the ForkJoinTask framework, and (iii) we report on experiences with using CONCURRENCER to parallelize several divide-and-conquer algorithms.

Vaziri et al. [14] present a *data-centric approach* to making a Java class thread-safe. The programmer writes annotations denoting *atomic sets*, i.e., sets of class fields that should be updated atomically, and *units-of-work*, i.e., meth-

ods operating on atomic sets that should execute without interleaving from other threads. Their system automatically generates one lock for each atomic set and uses the lock to protect field accesses in the corresponding units-of-work. Their system eliminates data races involving multiple variables, whereas CONCURRENCER works with `AtomicInteger` and `ConcurrentHashMap` that are designed to protect only single-variables. However, CONCURRENCER does not require any programmer annotations.

Balaban et al. [1] present a tool for converting between obsolete classes and their modern replacements. The programmer specifies a mapping between the old APIs and the new APIs, and the tool uses a type-constraint analysis to determine whether it can replace all usages of the obsolete class. Their tool can replace only a single API call at a time, whereas our tool replaces a set of related but dispersed API calls (like the ones in Fig. 4).

8 Conclusions and Future Work

Refactoring sequential code to introduce concurrency is not trivial. A good way to introduce concurrency into a program is via use of a library such as `j.u.c.`. Reengineering existing programs in this way is still tedious and error-prone.

Even seemingly simple refactorings—like replacing data types with thread-safe, scalable implementations—is prone to human errors. In this paper we present CONCURRENCER, which automates three refactorings for converting integer fields to `AtomicInteger`, for converting hash maps to `ConcurrentHashMap`, and for parallelizing divide-and-conquer algorithms. Our experience with CONCURRENCER shows that it is more effective than a human developer in identifying and applying such transformations, and the parallelized code exhibits good speedup.

We plan to extend CONCURRENCER to support many other features provided by `j.u.c.`. Among others, CONCURRENCER will convert sequential code to use other thread-safe `Atomic*` and scalable `Collection` classes, will extract other kinds of computations to parallel tasks using the `Executors` framework (task parallelism), and will convert `Array`s to `ParallelArray`s, a construct which enables parallel execution of loop operations (data parallelism).

As library developers make better concurrent libraries, the “introduce concurrency” problem will become the “introduce a library” problem. Tool support for introducing such concurrent libraries is crucial for the widespread use of such libraries, resulting in thread-safe and scalable programs.

Acknowledgements. John Brant, Adam Kiezun, Sasa Misailovic, Ralph Johnson, Jeff Overbey, Nick Chen, and the anonymous reviewers gave insightful comments on drafts of this paper. This work was funded in part by DARPA contract HR0011-07-1-0023.

References

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA'05*, pages 265–279.

[2] A. J. C. Bik, J. E. Villacis, and D. Gannon. Javar: A prototype Java restructuring compiler. *Concurrency - Practice and Experience*, 9(11):1181–1191, 1997.

[3] D. Dig, J. Marrero, and M. D. Ernst. How do programs become more concurrent? A story of program transformations. Technical Report MIT-CSAIL-TR-2008-053, MIT, September 2008.

[4] Double-checked lazy initialization pattern. http://artisans-serverintellect-com.si-eioswww6.com/default.asp?W122

[5] B. Freisleben and T. Kielmann. Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *Computers and Artificial Intelligence*, 14:579–596, 1995.

[6] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

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

[8] JSR-166y Specification Request for Java 7. http://gee.oswego.edu/dl/concurrency-interest/.

[9] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the Parascope editor. In *ICS'91*, pages 433–447.

[10] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley, 1999.

[11] D. Lea. A Java fork/join framework. In *JAVA'00*, pages 36–43.

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

[13] R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99*, pages 72–83.

[14] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*, pages 334–345.

[15] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.