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 threads-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*, which enables programmers to refactor sequential code into parallel code that uses* `j.u.c.` *concurrent utilities.* CONCURRENCER *does not require any program annotations, although the transformations are very involved: they span multiple program statements and use custom program analysis. A find-and-replace tool can not perform such transformations. 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.

In the multicore era, a major programming task is to retrofit parallelism into existing sequential programs. It is arguably easier to design a program with concurrency in mind than to retrofit concurrency later [8, 12]. 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 this 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 very error-prone: too many locks can greatly 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 a framework `Fork/Join Task` (`FJTask`) [7, 10] for fine-grained parallelism. Many computationally-intensive problems take the form of recursive *divide-and-conquer*. Classic examples include sorting (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.

However, 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 the `j.u.c` utilities. 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 quickly and safely refactor their sequential programs to use

`j.u.c.` utilities. Currently, CONCURRENCER supports three refactorings: (i) CONVERT INT TO ATOMICINTEGER, (ii) CONVERT HASHMAP TO CONCURRENTHASHMAP, and (iii) CONVERT RECURSION TO FJTASK. We previously cataloged [4] the transformations that open-source developers used to parallelize five projects. We found that the first two refactorings were among the most commonly used in practice.

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 alternative is for programmer to place a lock around the updating code, but this is error-prone and the map's lock degrades the map's performance under heavy lock-contention.

The third refactoring, CONVERT RECURSION TO FJTASK, 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 `FJTask` framework. Using the skeleton of the sequential algorithm, CONCURRENCER extracts the sequential computation into tasks which run in parallel and dispatches these tasks to the `FJTask` 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": they make a program thread-safe. The third refactoring makes a sequential program run concurrently.

The transformations performed by these refactorings are involved: they 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 a find-and-replace tool.

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. 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 several 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-in the `Atomic` classes) and *scalability* (the lock-free updates eliminate lock-contention under heavy accesses [8]).

2.2 Code Transformations

A programmer who wanted to use CONCURRENCER to make all accesses to an `int` thread-safe would start by selecting the field and invoking the CONVERT INT TO ATOMICINTEGER refactoring. CONCURRENCER changes the declaration type of the `int`

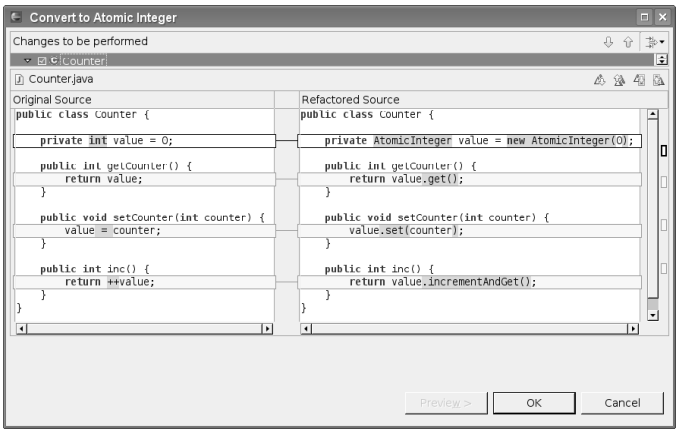

Figure 1. Using CONCURRENCER to convert an `int` to `AtomicInteger` in Apache Tomcat. Left/right shows code before/after refactoring.

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. We use this example to illustrate various code changes.

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'.

Read/Write Accesses. CONCURRENCER replaces read access (e.g., in method `getCounter()`) with a call to `AtomicInteger`'s `get()`. This has the same type `int` as the expression being replaced, so clients are unaffected. Since the mutable `AtomicInteger` does not escape its containing class `Counter`, a client can mutate the field only through the `Counter`'s methods.

CONCURRENCER replaces write accesses (e.g., in method `setCounter`) with a call to `AtomicInteger`'s `set()`.

Update Expressions. There are three kinds of update expressions: infix (e.g., `f = f + 2`), prefix (e.g., `++f`), and postfix (e.g., `f++`). CONCURRENCER rewrites an infix update expression using a call to the atomic `addAndGet(int delta)`.

CONCURRENCER rewrites a prefix update expression with a call to the atomic `incrementAndGet()` (e.g., method `inc()` in Fig. 1). It rewrites a postfix expression with a call to the atomic `getAndIncrement()`. CONCURRENCER similarly rewrites the decrement expressions.

`AtomicInteger` only provides APIs for replacing infix expressions involving the `+` operator. CONCURRENCER converts an infix expressions that use the `-` operator to an addition expression (e.g., it converts `f = f - 5` to `f.addAndGet(-5)`). If `AtomicInteger` had methods `divideAndGet` or `multiplyAndGet`, then CONCURRENCER could use them. In such cases, CONCURRENCER warns the user that the update expression cannot be made thread-safe, other than by using locks.

Synchronization. CONCURRENCER converts both a sequential program into one which is thread-safe, and also an already-concurrent program into one which is *more* concurrent. 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-in the `AtomicInteger`). CONCURRENCER only removes the synchronization if the code after refactoring contains exactly one call to the atomic APIs. Otherwise, a context switch can still occur between two consecutive calls to atomic APIs, 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 ++;
}
```

Similarly, CONCURRENCER only removes the synchronization if the synchronized block contains updates to one single field. Since the `AtomicInteger` ensures thread-safety for only one single field, it is of no help in cases when the program needs to maintain an invariant over multiple fields. For example a `ValueRange` object needs to ensure that its `max` int field is always greater than its `min` int field. To handle this, CONCURRENCER would need multivariable thread-safe container classes, currently not provided by `j.u.c`.

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 concurrency when multiple threads 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 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` implements the `Map` interface, therefore it 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 `Map` operations, and
- executes atomically.

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

Replacing a synchronized `HashMap` with `ConcurrentHashMap` offers dramatic scalability improvements [8].

3.2 Code Transformations

A programmer who wanted to use CONCURRENCER to make all accesses to an `HashMap` thread-safe would start by selecting the field and invoking 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.

The patterns have a similar structure: (i) check whether the map contains a certain key, and (ii) depending on the result, invoke one of `put(key, value)` or `remove(key)`. This structure can have small variations. For instance, the check can invoke `containsKey`, `get`, or an equality check using `get`. A temporary variable might hold the result of the check (like in Fig. 2). CONCURRENCER handles all combinations among these map update variations. Although these are the most common variations we have seen in real code, there might be other variations. Failing to convert those updates does not break user code; it only misses the opportunity to eliminate synchronization.

Fig. 2 illustrates one of the simplest transformations for using `putifAbsent`. In order to identify the potential usage of `putIfAbsent`, CONCURRENCER searches for conditional code which checks whether a certain key is not present in the `cache Map` field. If a `put` method call in the same conditional body uses the same key, CONCURRENCER has identified a potential usage of `putIfAbsent`. Next, CONCURRENCER replaces the two calls to the older APIs (`get` and `put`) with one call to `putIfAbsent` which executes atomically, without locking the entire map. The alternative is to protect the pair `get`/`put` with one global lock, but this alternative greatly reduces the application's scalability since the lock would prevent any other concurrent access to the map.

In the example in Fig. 2, the $value$ to be placed in the $\langle key, value \rangle$ map entry is simply created by invoking a constructor. However, in many cases the creational code for the newly inserted value is much more involved. Since the value to be inserted must be available before invoking `putIfAbsent`, in Fig. 3, CONCURRENCER extracts the creational code into a creational method (`createTimeZoneList`) and calls it to compute the value.

CONCURRENCER performs a data-flow analysis to find out if the created value is read after the call to `put`. If so, the created value is stored so that it can be accessed later. The example in Fig. 3 shows another trait: the `timeZoneList` value is also written in the conditional code. The original `getTimeZoneList` method returns either the null value, or the new value created in the conditional code. To preserve these semantics, CONCURRENCER conditionally assigns the newly created value to the `timeZoneLists` variable. The new conditional expression checks whether the call to `putIfAbsent` was successful: if the call succeeded, `putIfAbsent` returns null, otherwise it returns the previous value associated with the key. The refactored code uses the return status to decide whether to store the newly created value into the `timeZoneList` variable.

Before calling the `putIfAbsent` method, the value to be inserted must be available. Therefore, in the refactored code, the creational code is executed regardless of whether the new value is placed into the map. The refactored method has different semantics if the creational code has side effects (e.g., logging). CONCURRENCER checks whether the creational method has side effects, and if so, it warns the user.

We implemented a conservative analysis for determining side-effects. CONCURRENCER warns the user when the creational method assigns to fields or method arguments and when a method is called on a field or local variable. A constructor call is a special case of method call because fields nay be assigned inside the constructor.

An alternative to calling the creational method before `putIfAbsent` is to extract the creational code into a `Future` object, a `j.u.c.` utility class which represents the result of a future computation. In this case, the result of creational code will be retrieved only the first time when the programmer calls `get` on the `Future` object. CONCURRENCER could change all `map.get(value)` accesses to

```
// before refactoring
HashMap<String,File> cache = new HashMap<String,File>();

File rootFolderF;

public void service(Request req, final Response res){
  ...
  String uri = req.requestURI().toString();
  ...
  File resource = cache.get(uri);
  if (resource == null){
    resource = new File(rootFolderF, uri);
    cache.put(uri,resource);
  }
  ...
}
```

```
// after refactoring
ConcurrentHashMap<String,File> cache = new ConcurrentHashMap<String,Fil

File rootFolderF;

public void service(Request req, final Response res){
  ...
  String uri = req.requestURI().toString();
  ...
  cache.putIfAbsent(uri, new File(rootFolderF, uri));
  ...
}
```

Figure 2. Example of ConvertToConcurrentHashMap refactoring from GlassFish using the `putIfAbsent` pattern. Changes are underlined.

```
// before refactoring
private Map<Locale, String[]> timeZoneLists;
private String[] timeZoneIds;

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

  String[] timeZoneList = timeZoneLists.get(jiveLocale);
  if (timeZoneList == null) {
    timeZoneList = new String[timeZoneIds.length];
    for (int i = 0; i < timeZoneList.length; i++) {
      . . . // populate timeZoneList
    }

    // Add the new list to the map of locales to lists
    timeZoneLists.put(jiveLocale, timeZoneList);
  }

  return timeZoneList;
}
```

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

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

  String[] timeZoneList = timeZoneLists.get(jiveLocale);
  String[] createdTimeZoneList = createTimeZoneList(jiveLocale);
  if (timeZoneLists.putIfAbsent(jiveLocale, createdTimeZoneList) == nul
    timeZoneList = createdTimeZoneList;
  }
  return timeZoneList;
}

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

Figure 3. The user selects the `HashMap` field to be made thread-safe, and Concurrencer performs all the transformations. The figure shows an example from Zimbra using the `putIfAbsent` pattern with creational method (changes are underlined).

`map.get(value).get()`, which would force the execution of the creational code. However, this alternative does not solve the problem of side-effects either: the user code might rely on the side effect to happen when placing the value in the map, and now the side-effect only happens the first time the created value is retrieved from the map.

Synchronization. If the original method contained synchronization locks around map updates, CONCURRENCER removes them when they are superfluous (`ConcurrentHashMap` has thread-safety built in). Conservatively, CONCURRENCER only removes the locks if the refactored code corresponding to the original synchronized block contains only one call to `ConcurrentHashMap`'s APIs, and the original synchronization block only contains accesses to one single field. This latter check ensures the invariants over multiple variables are still preserved.

4 Convert Recursion to FJTask

4.1 FJTask Framework in Java 7

Java 7 will contain a framework, `FJTask`, 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., [15]) 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. 4 shows the sequential and parallel version of a gen-

```
// 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_TTHRESHOLD )
    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 4. Pseudocode for divide-and-conquer algorithm. Left hand side shows the sequential version, right hand side shows the parallel version.

eral 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.

Given the nature of divide-and-conquer algorithms, tasks that run in parallel should have the following characteristics:

- they are CPU-bound not I/O-bound, thus they do not block on I/O
- depending on the sequential threshold, many tasks (e.g. tens of thousands) can be spawned by the recursion branches
- they only need to synchronize when waiting for subtasks to complete

Given these properties, threads are not a good vehicle for running such tasks. Threads have high overhead (creating, scheduling, destroying) which might outperform the useful computation. Therefore 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 task scheduling is based on *work-stealing* [6, 13]: idle worker threads "steal" work from busy threads. The framework avoids contention for the data structures that hold the scheduling and ensures that each theft acquires a large chunk of work, thus making stealing infrequent. It is this effective scheduling that keeps all the cores busy with useful computation.

The most important API methods in `ForkJoinTask` are: `fork(Task)` which spawns the execution of a new task in parallel, `join(Task)` which blocks the current computation until the task passed as an argument finished, `forkJoin(Tasks)` which is syntactic sugar for calling `fork` and then `join`, and `compute` which is the hook-up method invoked by the framework when executing each task. `compute` implements 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 which executes `ForkJoinTask` computations using a pool of worker threads.

4.2 Code Transformations

CONCURRENCER converts a recursive divide-and-conquer algorithm to one which runs in parallel using the `FJTask` framework. The programmer need only select the divide-and-conquer method and supply the `SEQUENTIAL 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 `FJTask` 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 Fig. 5 shows the original, sequential version of the merge sort algorithm. The `sort` method takes as input the array to be sorted and returns the sorted array. The algorithm starts with the base case. In the recursion case, 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, 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` hook-up method neither takes any arguments, nor returns a value, the `SortImpl` has fields for the input arguments and the result of the computation. For each formal parameter of the original recursive method,

```
// 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
  }
}
```

```
1  // Parallel version
2
3  import jsr166y.forkjoin.ForkJoinExecutor;
4  import jsr166y.forkjoin.ForkJoinPool;
5  import jsr166y.forkjoin.RecursiveAction;
6
7  public class MergeSort {
8
9    public int[] sort(int[] whole) {
10     int processorCount = Runtime.getRuntime().availableProcessors();
11     ForkJoinExecutor pool = new ForkJoinPool(processorCount);
12     SortImpl aSortImpl = new SortImpl(whole);
13     pool.invoke(aSortImpl);
14     return aSortImpl.result;
15   }
16
17   private class SortImpl extends RecursiveAction {
18     private int[] whole;
19     private int[] result;
20
21     private SortImpl(int[] whole) {
22       this.whole = whole;
23     }
24
25     protected void compute() {
26       if ((whole.length < 10)) {
27         result = sort(whole);
28         return;
29       } else {
30         int[] left = new int[whole.length / 2];
31         System.arraycopy(whole, 0, left, 0, left.length);
32         int[] right = new int[whole.length - left.length];
33         System.arraycopy(whole, left.length,
34                                  right, 0, right.length);
35         SortImpl task1 = new SortImpl(left);
36         SortImpl task2 = new SortImpl(right);
37         forkJoin(task1, task2);
38         left = task1.result;
39         right = task2.result;
40         merge(left, right, whole);
41         result = whole;
42       }
43     }
44
45     private int[] sort(int[] whole) {
46       . . . copy the original, sequential implementation
47     }
48   }
49
50    private void merge(int[] left, int[] right,
51                                          int[] whole) {
52      . . . . merge left and right array into whole array
53    }
54 }
```

Figure 5. The programmer selects the divide-and-conquer method and provides the sequential threshold (`whole.length < 10`). Concurrencer converts the sequential divide-and-conquer into a parallel one using the FJTask framework. The left-hand side shows the sequential version, the right-hand side shows the parallel version (changes are underlined).

CONCURRENCER creates corresponding fields. In addition, if the recursive method has a non-void return type, CONCURRENCER creates a `result` field having the same declared type as the return type of the method. This field holds the result of the computation.

CONCURRENCER also generates a constructor having the same formal parameters as the original recursive method. A call to the original method is replaced with a call to this constructor, passing the actual parameters to the constructor. The constructor uses these parameters to initialize the class fields.

Implementing the `compute` method. The `compute` method is a hook-up method 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 `SEQUENTIAL THRESHOLD` expression provided by the user (line 26). Next, CONCURRENCER replaces the return statement in the base case of the original recursive method with a call to the sequential method (line

27). 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 35, 36). The arguments of the recursive call are passed as arguments to the constructor of the `RecursiveAction`. CONCURRENCER stores the newly created tasks into local variables, named `task1`, `task2`, etc.

Third, CONCURRENCER executes the parallel tasks and then assembles the result from the subresults of the tasks. CONCURRENCER invokes the `forkJoin` method while passing the previously created tasks as arguments. CONCURRENCER places the `forkJoin` method after the last creation of `RecursiveAction` (line 37). 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 38, 39). Subsequent code can thus use the subresults to assemble the final result (line 40). Lastly, CONCURRENCER assigns to the `result` field the combined subresults (line 41).

Reimplementing the recursive method. CONCURRENCER changes the implementation of the original recursive method to invoke the `FJTask` framework (lines 10-13). CONCURRENCER creates an executor object and initializes it with the number of threads to be used by the worker thread pool. The number of threads is equal with the number of available processors (found at runtime). The Java runtime system ensures that each processor will be assigned one worker thread. Since the divide-and-conquer algorithm is CPU-bound, creating more threads will not speed up the execution; on the contrary, it might slow the execution due to thread scheduling overhead. 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. Once the computation finished, the sorted array available in the `result` field is returned (line 14).

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. For example, a `fibonacci` function returns:

```
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:

```
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 which return a value, the result, and (ii) recursive methods which do not return any value, but they mutate at least one of the arguments to hold the result of the computation.

Fig. 5 is an example of the first kind of computation. The transformations for recursive methods which mutate one of their arguments to store the result are similar with the ones presented above, even slightly simpler, i.e., CONCURRENCER does not generate the code involving the `result` field.

5 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 running concurrent tasks in parallel?
- **Q2:** With respect to thread-safety, 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?
- **Q3:** With respect to running concurrent tasks in parallel, is the refactored 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 FJTASK, since `FJTask` is scheduled for Java 7's release, we could not find existing codebases using `FJTask`. We used CONCURRENCER to refactor several divide-and-conquer algorithms, and we answer first and third question.

5.1 Methodology

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

Table 1 lists 6 popular, mature open-source projects that use `AtomicInteger` and `ConcurrentHashMap`. We used the head versions available in 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.,

refactoring	in project	# of refactorings	LOC changed	LOC CONCURRENCER can handle
Convert Int	MINA	5	21	21
To AtomicInteger	Tomcat	5	26	26
	Struts	0	0	0
	GlassFish	15	60	60
	JaxLib	29	240	240
	Zimbra	10	54	54
Convert HashMap	MINA	6	14	14
To ConcurrentHashMap	Tomcat	0	0	0
	Struts	6	68	64
	GlassFish	14	86	86
	JaxLib	7	62	62
	Zimbra	44	388	377
Total for AtomicInteger and ConcurrentHashMap		**141**	**1019**	**968**
Convert Recursion	mergeSort([15])	1	36	36
to FJTask	fibonacci([13])	1	25	25
	maxSumConsecutive([13])	1	68	68
	matrixMultiply ([5, 13, 15])	1	108	108
	quickSort(Zimbra)	1	35	35
	maxTreeDepth(Eclipse)	1	30	30
Total for FJTask		**6**	**302**	**302**

Table 1. Programs used as case studies for CONVERT INT TO ATOMICINTEGER, CONVERT HASHMAP TO CONCURRENTHASHMAP, and CONVERT RECURSION TO FJTASK refactorings. Last two columns show LOC changed due to refactoring, and how many LOC can be changed by CONCURRENCER.

the refactored output could have used a concurrent API, but it instead uses the obsolete, lock-protected APIs).

For `AtomicInteger` we were able to find both the version with the `int` field and the version with `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`.

Setup for CONVERT RECURSION TO FJTASK.

We used CONCURRENCER to parallelize several divide-and-conquer algorithms. We use two sets of inputs: (i) classic divide-and-conquer algorithms used in others' evaluations [5, 13, 15], and (ii) divide-and-conquer algorithms from real projects.

Table 1 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.

The interested reader can find the input and the refactored programs on CONCURRENCER's webpage.

5.2 Q1: Is CONCURRENCER useful?

The top part of Table 1 show 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 968 lines of code; instead they would have had to only change 51 lines not currently handled by CONCURRENCER.

The bottom part of Table 1 show the LOC changed when converting the original recursive algorithm to one which uses the `FJTask` 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.

5.3 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 2 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.

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`. Although the refactored code is thread-safe, it 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 3 shows the number of such omissions in the case-study projects. We manually analyzed all the usages of `put` or `delete` and compiled a list of all the places where those usages could have been 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 3 shows that the open-source developers missed several opportunities to use the new APIs. CONCURRENCER missed much 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.

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

Table 4 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.

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

Table 2. Human errors in using `AtomicInteger` updates in refactorings performed by open-source developers.

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 4. Speedup of the parallelized divide-and-conquer algorithms.

6 Related Work

The earliest work on interactive tools for parallelization stemmed from the Fortran community and it targets loop parallelization. Interactive tools like PFC [9], ParaScope [11], and SUIF Explorer [14] rely on the user to specify what loops to interchange, align, replicate, or expand, what scalars to vectorize, etc. 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 in the tool), 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 FJTASK 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

	putIfAbsent			remove		
	potential usages	human omissions	CONCURRENCER omissions	potential usages	human omissions	CONCURRENCER omissions
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 3. Human and CONCURRENCER omissions in using `ConcurrentHashMap`'s `putIfAbsent` and conditional `remove`.

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 `FJTask` framework, and (iii) we report on experiences with using CONCURRENCER to parallelize several divide-and-conquer algorithms.

Vaziri et al. [16] 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., methods 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 is more general than ours, since it can work with any API mapping, for example one between `HashMap` and `ConcurrentHashMap`. CONCURRENCER is less general, since the conversion between `HashMap` and `ConcurrentHashMap` is custom implemented. However, such a conversion requires more powerful AST pattern matching and rewriting than the one used in their tool. 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. 2, 3).

Boshernitsan et al. [3] present iXj, a general framework for code transformations. iXj has an intuitive user interface that enables the user to quickly sketch a pattern for the code transformation. Although useful for a broad range of transformations, iXj is not able to transform code where the pattern matching executes against several dispersed statements (like the ones in Fig. 2, 3, 5). In such scenarios, a user needs to use a custom implemented transformation tool like CONCURRENCER.

7 Conclusions and Future Work

Refactoring sequential code to concurrency is not trivial. A good way to introduce concurrency into a program is via use of a good concurrency 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 `Arrays` to `ParallelArrays`, 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 "in-

troduce a library" problem. Tool support for introducing such concurrent libraries is crucial for the widespread use of such libraries, resulting in more thread-safe, more scalable programs.

References

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of Object-oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.

[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] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 567–576, New York, NY, USA, 2007. ACM.

[4] 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.

[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] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[7] B. Goetz. What's New for Concurrency on the Java Platform. Keynote Talk at JavaOne Conference, 2008. http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-5515&yr=2008&track=javase.

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

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

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

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

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

[13] D. Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[14] 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.

[15] R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPOPP*, pages 72–83, 1999.

[16] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 334–345, New York, NY, USA, 2006. ACM.