Concurrencer: a Tool for Retrofitting Concurrency into Sequential Java Applications 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. To use this package, programmers still need to refactor existing code. This is* tedious*,* error-prone*, and* omission-prone*.*

This demo 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 span several, non-adjacent, 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

The computing hardware industry has shifted to multicore processors. This demands that programmers find and exploit parallelism in their programs, if they want to reap the same performance benefits as in the past.

The dominant paradigm for concurrency in desktop programs is shared-memory, thread-based. However, this paradigm increases the risk for deadlocks and data-races, commonly known as *thread-safety* concerns. In addition, the programmer needs to consider *scalability* concerns as well: will the parallelized program continue to run faster when adding more parallel resources?

To meet the needs of programmers with respect to thread-safety and scalability, the Java standard library has been extended with a package, `java.util.concurrent` (from here on referred as `j.u.c.`), containing several utility classes for dealing with concurrency. Among others, `j.u.c.` contains a set of `Atomic` classes which offer thread-safe, lock-free programming over single variables, and several thread-safe abstract data types (e.g., `ConcurrentHashMap`) optimized for scalability. Java 7 will contain a framework `Fork/Join Task`[1] for fine-grained parallelism of intensive computations.

However, manually refactoring a program to use `j.u.c.` utilities 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 demo presents Concurrencer, our extension to Eclipse's refactoring engine. Concurrencer enables Java programmers to quickly and safely refactor their sequential programs to use `j.u.c.` utilities. In this demo we present three refactorings: (i) Convert Int to AtomicInteger, (ii) Convert HashMap to ConcurrentHashMap, and (iii) Convert Recursion to FJTask.

The first two refactorings are "enabling transformations", i.e., they make a program thread-safe, but do not introduce multi-threading into a single-threaded program. Our previous study [1] of five open-source projects that were manually parallelized by their developers shows that these two refactorings were among some of the most commonly used in practice. The third refactoring introduces multi-threading: it converts a sequential recursive divide-and-conquer algorithm into one which solves the subproblems in parallel using `ForkJoinTask`s.

For evaluation, we compared the manually refactored code in 6 open-source projects with code refactored automatically. The results show that Concurrencer is effective and the parallel code exhibits good speedup.

A more detailed description of Concurrencer can be found in the ICSE'09 research track [2]. Concurrencer can be downloaded from: `http://refactoring.info/tools/Concurrencer`

[1] http://gee.oswego.edu/dl/concurrency-interest/

978-1-4244-3494-7/09/$25.00

2 Concurrencer

Supported Refactorings. The first refactoring, CONVERT INT TO ATOMICINTEGER, enables a programmer to convert an `int` field to an `AtomicInteger`. `AtomicInteger` is a lock-free utility class which encapsulates an `int` value and provides update operations that execute *atomically*. Our refactoring replaces field updates with calls to `AtomicInteger`'s APIs.

For example, a common update pattern on an `int` field is (i) read the current value, (ii) add delta, and (iii) update the field value. To make this update thread-safe, the three operations need to execute *atomically*. Traditionally, programmers use locks to ensure atomicity. Due to the program having to frequently acquire and release the lock, the program does not scale well under heavy lock-contention. CONCURRENCER finds such read/add/update code patterns and replaces them with a call to `AtomicInteger`'s `getAndAdd()` which *atomically* executes the update without locks (instead it uses efficient compare-and-swap).

The second refactoring, CONVERT HASHMAP TO CONCURRENTHASHMAP, enables a programmer to convert an `HashMap` field to `ConcurrentHashMap`, a thread-safe, highly scalable implementation for hash maps. Our refactoring replaces map update patterns with calls to `ConcurrentHashMap`'s atomic APIs.

For example, a common update pattern is (i) check if a map contains a $\langle key, value \rangle$ pair, and if it is not present, (ii) place the pair in the map. For thread-safety, the two operations need to execute atomically. Traditionally, a programmer would use a map-common lock. Since all accesses to the map have to acquire the map's lock, this can severely degrade the map's performance. 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 FJTASK, converts a sequential divide-and-conquer algorithm into an algorithm which solves the recursive subproblems in parallel using the `Fork/Join Task` framework. Our refactoring encapsulates the subproblems as `ForkJoinTask`s and passes them to the framework for effective scheduling.

For example, a sequential `mergeSort(array)` first checks whether the input array is of trivial size (and sorts it directly), otherwise splits the array into two halves, sorts each half, and then merges the sorted halves. CONCURRENCER parallelizes this algorithm using the skeleton of the sequential algorithm. For the base-case it (i) checks whether the array is smaller than a user-defined threshold and (ii) invokes the original sequential sort. For the recursive case, it creates `ForkJoinTask`s for each of the two halves, schedules the two tasks in parallel, waits for the computations to finish, and then merges the two sorted halves.

Implementation. CONCURRENCER is implemented as an extension to Eclipse's refactoring engine, thus it conveniently provides previewing the code changes, undo, etc. The programmer only needs select the method or field to be refactored, and the concurrency refactoring (and the sequential threshold in case of CONVERT RECURSION TO FJTASK).

CONCURRENCER's program analysis determines (i) whether it is safe to remove synchronization locks that might protect field accesses, (ii) its data-flow analysis determines what variables are written in the update patterns and read afterward, and assigns them appropriately when the update pattern is replaced with a single API call. The analysis and the code edits are implemented on top of Eclipse's `JDT`.

Evaluation. We used CONCURRENCER to refactor the same fields that the open-source developers of Tomcat, MINA, JaxLib, Zimbra, GlassFish, and Struts refactored to `AtomicInteger` or `ConcurrentHashMap`. There were a total of 141 such refactorings. Using CONCURRENCER, the developers could have saved changing 1019 LOC manually.

We then compared the manually vs. automatically refactored programs. In terms of errors in using the j.u.c. APIs, the open source developers 4 times erroneously replaced infix expressions like `++f` with `f.getAndIncrement()` (which corresponds to the postfix expression `f++`). CONCURRENCER used the correct API calls. In terms of missing opportunities to convert from old update patterns to the new atomic APIs, the programmers missed 43 out of 83 such opportunities. CONCURRENCER only missed 10 opportunities.

We also used CONCURRENCER to parallelize 6 divide-and-conquer algorithms using CONVERT RECURSION TO FJTASK. CONCURRENCER changed 302 LOC and the parallelized code exhibits on average 1.84x speedup on a 2-core machine and 3.28x speedup on a 4-core machine.

3 Conclusions

Refactoring sequential code to concurrency is not trivial. Even seemingly simple refactorings like replacing data types with thread-safe, scalable implementations provided in `java.util.concurrent`, is prone to human errors and omissions. This demo presents CONCURRENCER which automates three refactorings. Our preliminary experience with CONCURRENCER shows that it is more effective than a human developer in identifying and applying such transformations.

References

[1] D. Dig, J. Marrero, and M. D. Ernst. How Do Programs Become More Concurrent? A Story of Program Transformations. Tech Report MIT-CSAIL-TR-2008-053.

[2] D. Dig, J. Marrero, and M. D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. To Appear in Proceedings of ICSE'09.