How Do Programs Become More Concurrent? A Story of Program Transformations

Danny Dig
University of Illinois
dig@illinois.edu

John Marrero
Massachusetts Institute of Technology
aeon@csail.mit.edu

Michael Ernst
University of Washington
mernst@cs.washington.edu

ABSTRACT

In the multi-core era, programmers need to resort to parallelism if they want to improve program performance. Thus, a major maintenance task will be to make sequential programs more concurrent. Must concurrency be designed into a program, or can it be retrofitted later? What are the most common transformations to retrofit concurrency into sequential programs? Are these transformations random, or do they belong to certain categories? How can we automate these transformations?

To answer these questions we analyzed the source code of five open-source Java projects and looked at a total of 14 versions. We analyzed qualitatively and quantitatively the concurrency-related transformations. We found that these transformations belong to four categories: transformations that improve the responsiveness, the throughput, the scalability, or correctness of the applications. In 73.9% of these transformations, concurrency was retrofitted on existing program elements. In 20.5% of the transformations, concurrency was designed into new program elements. Our findings educate software developers on how to parallelize sequential programs, and provide hints for tool vendors about what transformations are worth automating.

Categories and Subject Descriptors: D.1.3. [Programming Techniques]: Concurrent Programming; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms: Design, Management, Measurement

Keywords: Program transformation, concurrency, parallelism

1. INTRODUCTION

For several decades, the computing hardware industry has kept up with Moore's Law, doubling the speed of desktop computers every 18 months. In addition to algorithmic improvements, application programmers have relied on Moore's Law to improve the performance of software applications. However, because uni-processors' clock rates can no longer be improved, the industry shifted to multi-core computers. This demands that programmers find and exploit parallelism in their applications, if they want to reap the performance improvements from the multi-core hardware.

IWMSE '11, May 21, 2011, Waikiki, Honolulu, HI, USA

Parallel programming and concurrency have been used for decades, but they were the skill set of elite programmers. From now on, parallel programming will be a skill that any professional programmer will have to acquire. The dominant paradigm for parallel programming in desktop computing is shared-memory, thread-based parallelism. Due to non-determinism, this paradigm adds extra complexity and increases the potential for deadlocks and data races.

Dealing with concurrency is easier if concurrency is designed into the system from the beginning, rather than being retrofitted later on [12, 15]. However, most programs were not designed with concurrency in mind. In the multi-core era, a major maintenance task will be to retrofit concurrency into existing programs so that they can take advantage of the hardware improvements. Must concurrency be designed into a program, or can it be retrofitted later? What are the most common transformations to retrofit concurrency into sequential programs? Are these transformations random, or do they belong to certain categories? How can we automate these transformations?

To answer these questions, we create a taxonomy of the most common program transformations related to concurrency in five open-source widely successful projects. We analyze qualitatively and quantitatively these transformations along two or three versions of each project. Our goals are: (i) to inform software developers about the trend of program transformations they are going to perform during the multi-core era, (ii) to shed light into the process, and (iii) to provide recommendations to tool builders about what transformations need to be (semi)-automated in the future.

To build the taxonomy, we manually analyzed two or three versions of five open-source Java projects: two core Eclipse [6] plugins, JUnit [13], Apache Tomcat server [25] and Apache MINA library [18]. Some of these projects are large, so we guided our analysis by reading the release notes, searching in the source code for the concurrency fingerprints (e.g., references to `synchronized` or `Thread`), and comparing the source code of different versions of program elements that contain the concurrency fingerprints.

We found that these parallelizing transformations are not random, but they fall into four categories: transformations that improve the *latency* (i.e., an application feels more responsive), transformations that improve the *throughput* (i.e., more computational tasks executed per unit of time), transformations that improve the *scalability* (i.e., the performance scales up when adding more cores), and transformations that improve correctness (i.e., fix concurrency-related bugs so that application behaves according to specification).

Also, we found that programmers make consistent changes in each version, as if they were in the "mood": they focus on one objective at a time, and repeat the same kind of transformations. This suggests that it is worth automating these transformations. We survey the recent tools [4, 5, 11, 14, 23, 27] that started automating

transformations and found that while they support transformations for throughput, correctness, and scalability, they do not cover transformations for improving responsiveness.

In summary, this paper makes the following contributions:

- presents the results of a qualitative and quantitative empirical study about the common concurrency-related program transformations into five real programs (see Section 3). To the best of our knowledge, this is the first such study.
- presents some practical applications for the reported findings. First, it educates software developers on how others ported existing applications to use the multi-cores (Section 3.2). Second, it brings evidence that concurrency can be retrofitted later on (Section 4.1), and shows that it can be done in orderly fashion (Section 4.2). Third, it surveys the state-of-the-art tools (Section 4.3) for making concurrency-related transformations and it provides hints for tool builders on which transformations are worth automating.

2. EXPERIMENTAL SETUP

2.1 Concurrency Fingerprints

Before we present the design of the experiment, we provide a gentle introduction to concurrency in Java. Java uses lock-based synchronization to achieve atomic execution of statements. In Java, every object has a built-in, *intrinsic* lock associated with it. Java provides a concise syntax to denote that a whole method body is protected by the intrinsic lock: the programmer simply adds the `synchronized` keyword to the method signature declaration.

When the programmer needs more flexibility in expressing atomicity, Java provides synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the lock; such locks are called *extrinsic* locks[1]. Synchronized statements are useful for improving concurrency by providing fine-grained synchronization: (i) they allow synchronization at finer level than whole method body, and (ii) they allow a more flexible locking scheme by allowing more than one lock to protect accesses to members of a class. In addition Java provides even more flexible locks[2].

Java programs use `java.lang.Thread` to execute concurrent work. Graphical toolkits also provide utility classes to run tasks in the UI event thread (e.g., `javax.swing.SwingUtilities` in Swing, or `org.eclipse.swt.widgets.Display` in SWT). The Java standard libraries include a package, `java.util.concurrent`, with several utility classes useful in concurrent programming.

2.2 Case Studies

We describe briefly each case study project, the versions that we analyzed, and the main concurrency-related themes in those versions.

We selected case studies that cover the whole lifecycle of concurrency. Two case studies (Search and DOM) are *infants* with respect to concurrency: they were freshly converted from sequential to parallel programs. Two case studies (Tomcat and MINA) are *veterans* with respect to concurrency: they were parallelized a long time ago, or they were designed with concurrency in mind. One case study (JUnit) is somewhere in the middle.

[1]Note that a programmer can also specify `this` as a lock

[2]e.g., `ReentrantLock` enables non-block-structured lock operations and fairness, `ReadWriteLock` enables to distinguish between reader and writer locks allowing multiple readers to execute concurrently

2.2.1 Eclipse Search

`org.eclipse.search` is a core plugin in the Eclipse IDE. It handles Java-specific searches as well as general file search queries. We studied version 2.1.3 (March 2004), 3.0 (June 2004), and 3.3.2 (April 2008). A major theme in Eclipse 3.0 is improving the responsiveness of the IDE so that the UI feels more alive. Eclipse accomplished this goal by allowing long-running operations, such as search, to run in background threads.

2.2.2 Eclipse Java DOM

`org.eclipse.jdt.core.dom` is a subcomponent of the core Java tooling in Eclipse. It contains a parser and the Abstract Syntax Tree (AST) nodes, as well as several utility classes. The AST DOM nodes are used by all Java plugins that display or manipulate Java source code. We studied versions 2.1.3, 3.0, and 3.3.2. According to the responsiveness theme, the AST DOM nodes are concurrently accessed from several tools (e.g., the method override indicator or the semantic coloring in the editor).

2.2.3 JUnit

JUnit is a framework for executing test cases. It is the Java implementation of the xUnit family of testing frameworks. We studied versions 1.0, 3.8.2, and 4.0. JUnit 3.8.2's UI improved its responsiveness; in addition tests can be run in separate threads improving the throughput.

2.2.4 Apache Tomcat Server

Apache Tomcat is a web container, or application server, enabling Java code to run in cooperation with a web server. Tomcat is the official Reference Implementation for the Java Servlet and the JavaServer Pages (JSP) specifications. We studied versions 4.1.1 (Oct 2003), 5.5 (Sept 2005), and 6.0 (Oct 2006). These versions fixed several concurrency-related bugs and improved scalability.

2.2.5 Apache MINA

Apache MINA is a network application framework which helps users develop high performance and high scalability network applications easily. It provides an abstract, event-driven, asynchronous API over various transports such as TCP/IP and UDP/IP via Java NIO. We studied versions 1.0 (October 2006) and version 1.1 (April 2007). Version 1.1 contains scalability improvements.

For each program version that we analyzed, first row presents its size (in LOC) and second row presents the total number of synchronized blocks.

2.3 Design of the Experiment

The projects that we analyzed range from a few KLOC to hundreds of KLOC. A thorough manual analysis of *all* source code changes in such large projects is not feasible. Below we describe the process that we used to guide our analysis for each project.

- We read the version-release documents for each project. These release documents are produced by the developers of the projects and usually describe the major architectural or design changes in each version. For Eclipse we used its help system, section `Eclipse 3.0 Plugin Migration Guide`, specifically the documents "Incompatibilities between Eclipse 2.1 and 3.0" and "Adopting 3.0 mechanisms and API". For Tomcat we used `http://tomcat.apache.org/tomcat-5.5-doc/changelog.html` and for MINA we used `http://issues.apache.org/jira/browse/DIRMINA`.
- We selected a version (say V_{conc}) for which the documentation confirms major concurrency-related changes. Then we

	Eclipse Search			Eclipse DOM			JUnit			Tomcat			MINA	
	2.1.3	3.0	3.3.2	2.1.3	3.0	3.3.2	1.0	3.8.2	4.0	4.1	5.5	6.0	1.0	1.1
Size [KLOC]	11	21	24	27	52	62	3	8	10	402	489	338	40	40
# Synch Blocks	4	34	42	12	110	121	15	20	18	921	1152	1108	422	211
# Stmt./Method	2/2	23/11	28/14	3/9	97/13	99/122	0/15	2/18	1/17	415/506	457/695	413/604	172/250	90/121
# `this`/extr. Lock	2/0	3/20	11/17	0/3	82/15	82/17	0/0	2/0	1/0	51/364	73/384	75/338	13/159	6/84

Table 1: Statistics of the case-study programs for each version we analyzed: size in KLOC, total number of synchronized blocks. The third row presents how many of these synchronized blocks protect whole method bodies (and necessarily use the intrinsic locks) and how many are finer-grained at the statement level. For these statement level synchronized blocks, the fourth row presents how many of them use the intrinsic `this` lock vs. custom extrinsic lock.

searched the source code for the *fingerprints* of concurrent code (e.g., `synchronized`, `Thread`, etc.).

- For program elements (e.g., methods and classes) that contained concurrency artifacts, we manually analyzed how these program elements changed between the concurrent version V_{conc} and a previous major release, say V_{prev}. In addition, we analyzed the same program elements in one more version, V_{next}, after V_{conc}. This helped us to find how the previously introduced concurrency constructs evolved. We also searched for additional program elements in V_{next} that contain concurrency fingerprints.
- We recorded the kinds of concurrency-related changes (qualitative), and the number of distinct such changes (quantitative).

Table 1 presents some general statistics about the studied programs.

3. CONCURRENCY-RELATED PROGRAM TRANSFORMATIONS

A concurrency-related program transformation is an *addition*, *removal*, or *edit* of a concurrency artifact (e.g., `synchronized`, `Thread`, concurrent utility) to an existing or a new program element (e.g., class, method, statement).

In the programs that we studied, there are four objectives for making concurrency-related program transformations: improving latency, throughput, scalability, or correctness. Our hypothesis is that any particular concurrency-related transformation tries to achieve at least one of these four objectives. The same transformation can achieve more than one of the four objectives.

Subsection 3.2 lists all types of concurrency-related transformations that we found in the five case-studies. The webpage [2] presents before-and-after code examples for each transformation.

In subsection 3.2 we present each transformation under the objective it achieves in the case study where it comes from. In subsection 3.3 we give a more general categorization, by describing how each transformation can achieve more than one objective.

Section 5 concludes by addressing threats to validity.

3.1 Objectives for Concurrency Transformations

In the five case studies, the concurrency-related transformations were not random, but they fell into four objectives.

Improve Responsiveness.

Responsiveness measures how long it takes from the moment of asking for the result of a computation until a part of the result is available. To improve user satisfaction, an application should feel responsive, even when executing long-running computations. For example, in Eclipse, searching for all references to a program element can be a long-running operation. Because the search runs in a background thread, a user can still browse through the source code, or inspect the partial search results, before all search finishes.

Improve Throughput.

Throughput measures the amount of results that are computed per unit of time. To improve throughput, programmers break down computation so that processors can process data in parallel.

Improve Scalability.

It is desired that the speed up of a parallel application scales up when adding more processors. The upper bound for the speed up is inverse proportional with the percentage of computation that runs sequentially. In parallel programs, synchronization constructs often serialize the computation. Programmers fine-tune the fraction of synchronized code in order to improve scalability.

Correctness.

An application should behave according to its specification even when it is accessed concurrently from multiple threads. In an object-oriented sequential program, the class invariants need to hold only before method-entry and after method-exit. However, in a concurrent program, the same invariants need be preserved at all points where a context switch can occur, even in the middle of a method. To enforce these invariants, operations that manipulate the internal state need to be executed *atomically*. Furthermore, changes to an object's state need to be *visible* to other threads. In shared-memory, thread-based systems, synchronization is the most common means to achieve both atomicity and visibility.

3.2 Examples of Concurrency Transformations

3.2.1 Improving the Responsiveness

Separate UI and Computation Threads.

In JUnit, concurrency is used to improve the responsiveness of the UI. JUnit 3.8.2 has three modes of displaying the results: a textual output on the console, one Swing UI and one AWT UI. If the tests ran in the UI thread, the UI would block until all tests finished executing, thus preventing a user from stoping a test run in the middle. To prevent blocking the UI while tests run, `TestRunner` uses two threads: one for the UI, and another one (spawned from the UI) in which it runs all tests.

Delegate Computation to Event Dispatching Thread.

As an alternative to spawning a new thread, graphical applications can delegate computation to a special dedicated *event dis-*

patching thread for handling GUI events (e.g., mouse click, pressed button).

In addition, this is also an alternative to using synchronized blocks to ensure thread safety. Graphical applications can delegate thread safety by confining the UI update operations to a single thread.

For example, JUnit updates the progress indicator bar inside a runnable, scheduled for asychronous execution in the event dispatching thread. The class `javax.swing.SwingUtilities` provides a method `invokeLater` for non-blocking, *asynchronous* execution of a runnable – a runnable's `run` method is executed after all pending events have been processed. `SwingUtilities` also provides a method, `invokeAndWait`, for *synchronous* execution of a runnable – the invoking code blocks until all pending events have been processed *and* the runnable's `run` method was executed:

```
public void testEnded(String stringName) {
...
  SwingUtilities.invokeLater(
    new Runnable() {
    public void run() {
      if (fTestResult != null) {
        fCounterPanel.setRunValue(fTestResult.runCount())
            ;
        fProgressIndicator.step(fTestResult.runCount(),
                                fTestResult.wasSuccessful())
                                    ;
      }
    }
    }
  );
}
```

Eclipse's SWT toolkit uses a similar utility class, `Display`, which provides `asyncExec(Runnable)` and `syncExec(Runnable)` for scheduling the runnable in the SWT event thread. Eclipse's Search plugin uses both methods.

Method Object with Runnable.

A long-running computation can be encapsulated in an object whose API offers a `run` method. Depending on the value of a parameter passed to `run`, the computation executes in the main thread, or in a background thread. The following code snapshot illustrates this idiom in Eclipse Search:

```
run(runInSeparateThread, new ReplaceOperation() {
    protected void doReplace(IProgressMonitor pm){
       replace(pm, replaceText);
    }
    });
```

3.2.2 *Improving the Throughput*

Introduce Loop Parallelism.

Loop-parallelism [17] is an idiom used to parallelize iterations of a computationally intensive loop. The loop computation is split among several threads, with each thread executing the same operations on a subset of the whole data. At the end of the computation, the partial results are assembled to form the final result.

In JUnit, `TestSuite` represents a collection of tests. Its `run` method iterates over all the tests in a test suite, and calls `runTest`, which runs a particular test and reports the results in the `TestResult`:

```
class TestSuite
public void run(TestResult result) {
  for (Enumeration e= tests(); e.hasMoreElements(); ) {
    if (result.shouldStop() )
        break;
    Test test= (Test)e.nextElement();
    runTest(test, result);
  }
}
```

Notice that there are no loop-carried dependencies over individual iterations. Therefore, this loop can be split into iterations that execute in parallel. `ActiveTestSuite` refines the behavior of `TestSuite` by overriding the "hook" method `runTest`. The refined implementation of `runTest` spawns a new thread in which it runs a test:

```
class ActiveTestSuite
public void runTest(final Test test, final TestResult
    result) {
  Thread t= new Thread() {
  public void run() {
    try {
      test.run(result);
    } finally {
        ActiveTestSuite.this.runFinished();
    }
  }};
  t.start();
}
```

3.2.3 *Improving the Scalability*

Reducing the Duration of the Held Lock.

Rather than holding a lock for the duration of a long-running operation that does not need be synchronized, a lock can be released and re-acquired later. This enables other waiting threads to grab the lock and continue execution. Below we show an example from Eclipse's Search plugin. Notice that the lock is released during the long-running operation that opens and renders the window dialog:

```
public boolean okToClose() {
  ...
  synchronized (this) {
    fWindowClosingDialog= createClosingDialog();
  }
  fWindowClosingDialog.open(); //long-running
  synchronized (this) {
    fWindowClosingDialog= null;
  }
}
```

Copy-then-Iterate.

Holding a lock while iterating over a collection and executing a long-running operation prevents other threads from executing. Instead of holding a lock during the whole iteration, one could hold a lock just to copy the collection, then release the lock while iterating over the copy. We illustrate this idiom in Eclipse's Search plugin:

```
void fireStarting(ISearchQuery query) {
  Set copiedListeners= new HashSet();
  synchronized (fListeners) {
    copiedListeners.addAll(fListeners);
  }
  Iterator listeners= copiedListeners.iterator();
  while (listeners.hasNext()) {
    IQueryListener l= (IQueryListener) listeners.next();
    l.queryStarting(query);
  }
}
```

Notice that this idiom prevents interference among concurrent threads, since the copied collection is a local, stack-confined object. This "snapshot" style iterator uses a reference to the state of the collection at the point when the iterator was created. However, the copied collection will not reflect additions, removals, or changes to the original collection since the iterator was created.

Using Atomic Classes.

`java.util.concurrent.atomic` APIs support lock-free thread-safe programming on single variables. For example, `AtomicInteger` wraps an integer value and provides APIs like `getAndIncrement`

or `compareAndSet` that execute two operations atomically. The atomic classes are implemented using efficient *compare-and-swap* hardware instructions. Under low to moderate contention, atomics scale better than locks [12].

Using Concurrent Collections.

The new `java.util.concurrent` APIs in the Java standard libraries provide scalable alternatives to previous collection data structures. For example, `ConcurrentHashMap` is an efficient implementation of `Hashtable` that allows several readers to execute concurrently (without blocking). It allows a number of writers to execute concurrently (without blocking) by splitting the range of hash values into different hash buckets.

We found several examples of conversions from old collections to concurrent collections and from primitive types to atomics in both MINA and Tomcat.

3.2.4 Correctness

Concurrency is gradually refined in consecutive versions of real-world programs. For example, in servers like Tomcat which were designed from the beginning to be concurrent, we noticed additions of synchronization blocks. We could correlate some of them with bug reports and patches saying that the change was triggered by either insufficient or inexistent previous synchronization.

Add Synchronized Block.

This idiom add synchronization protection to a previously unprotected field access.

Coarsen synchronized block.

If a previously synchronized block did not cover all the shared fields involved in an class invariant, developers expanded the synchronization block over to cover the shared fields.

Thread-safe Lazy Initialization.

Lazy initialization of fields needs to be thread-safe, i.e., prevent multiple threads from initializing the same field. Below is an example from Eclipse's DOM AST. Notice that the lock is acquired only if the field is not initialized:

```
public SimpleName getName() {
  if (this.typeName == null) {
    // lazy init must be thread-safe
    synchronized (this) {
      if (this.typeName == null) {
        preLazyInit();
        this.typeName = new SimpleName(this.ast);
        postLazyInit(this.typeName, iNameProperty());
      }
    }
  }
  return this.typeName;
}
```

Change Lock Object.

Rather than using `this`, the default lock for synchronized blocks, fields that are aliasing objects passed as method arguments need to be protected by the same lock that protects the method argument. In the example below, the lock object was changed from `this` to the collection that the block protects:

```
public void addListener(ISearchResultListener l) {
  synchronized (fListeners) {
    fListeners.add(l);
  }
}
```

3.3 Summary of Transformations

Table 2 lists the concurrency-related transformations that we found in the five case studies.

We label these transformations depending on (i) whether new concurrency constructs were added or existing concurrency constructs were changed, and (ii) whether the concurrency constructs are applied to existing program elements, or to new program elements. Based on theses combinations, the table below shows three outcomes[3]: concurrency was retrofitted, designed, or modified.

		Code	
		existing	new
Concurrency	changes	modified	N/A
	new	retrofitted	designed

Sometimes, the concurrency fingerprints (Section 2.1) in versions V_1 and V_2 were associated with program elements (e.g., classes, methods, fields, statements) that exist in both V_1 and V_2. For example, field `StandardSession.accessCount` exists in both Tomcat_5.5 and Tomcat_6.0. However, in the latter version its type was converted from `int` to `AtomicInteger`. We notice that programmers add *new concurrency* to an *existing* program element, thus we say that concurrency was *retrofitted*.

Other times, the concurrency fingerprint was associated with a program element that we could not find in the previous version that we took into account. For example, in JUnit_3.8.2, we found that class `ActiveTestSuite` was using threads to launch new tests. However, we could not find this class in the previous version that we took into account, i.e., JUnit_1.0. Thus we could only infer that programmers added a brand new program element that contained a concurrency idiom from the first time when the element was introduced, i.e., *new concurrency* was added to a *new* program element. In this case we say that concurrency was *designed* into the program element from the beginning.

Other times, we found changes in concurrency constructs that related to program elements existing in both versions. For example, EclipseSearch_3.3.2 plugin contains some synchronized statements in method `AbstractTextSearchViewPage.addQuery`. In the previous version that we took into account, i.e., version 3.0, we found the same method `addQuery`. In this version, the statements were synchronized on a different lock. In this case, we say that concurrency was *modified*.

Adding synchronized blocks is a transformation that cross-cuts most other transformations, since synchronization blocks are the primary construct from which the other transformations are made of. Therefore, we distinguish between adding synchronized blocks as a side effect of a transformation vs. adding synchronization blocks because the previously parallelized code was insufficiently synchronized. In Table 2 we report only the latter; these changes are correlated with bugs reports.

The last column of Table 2 links these transformations with objectives for making concurrency-related transformations in the five case studies.

4. SUMMARY OF FINDINGS

Several observations arise from this study. For example, the same kind of transformation can be used to achieve different objectives (latency, throughput, scalability, correctness).

Using data from Table 2, we answer three questions.

[3]note that for brand new program elements, concurrency constructs did not exist before, so they can't be changed

	Search 2.1.3 → 3.0	Search 3.0 → 3.3.2	DOM 2.1.3 → 3.0	JUnit 1.0 → 3.8.2	Tomcat 4.1 → 5.5	Tomcat 5.5 → 6.0	MINA 1.0 → 1.1	Objective
Spawn new Threads				3_D				Res/Thrp
Deleg to Evt. Disp. Thread	7_R			9_D				Res/Corr
Method Obj. Runnable	4_R							Res
Loop Parallelism				1_D				Thrp
Reduce Lock Duration				2_M				Res/Sca
Copy-then-iterate	5_R			2_R				Res/Sca/Corr
Use Atomic Classes					27_D	10_R	5_R	Sca/Corr
Use Conc. Collections						1_R	32_R	Sca/Corr
Add Synch. Block		3_R			15_R			Corr
Coarsen Synch. Block					6_M			Corr
Lazy Initialization			78_R					Corr
Change Lock Object		3_M						Corr
Use ThreadLocal					5_D			Sca/Corr
Remove Synch. Block		1_M						Res/Sca

Table 2: Concurrency-related transformations in the five case studies. We mark with $_R$ the transformations that added new concurrency to existing program elements (i.e., concurrency was retrofitted), we mark with $_D$ transformations that added new concurrency to new program elements (i.e., concurrency was designed), and we mark with $_M$ transformations that change concurrency already present in existing code (i.e., concurrency was modified). The last column correlates the transformations with objectives (Res = Responsiveness, Thrp = Throughput, Sca = Scalability, Corr = Correctness)

4.1 Q1: Is Concurrency Designed into a Program, or Is It Retrofitted?

Table 2 shows several cases when concurrency was added to a program element that existed in both versions of the program, thus concurrency was *retrofitted*. It also shows cases when concurrency was added to a program element that exists only in the latter version, thus concurrency was designed into that program element. It also shows cases when programmers modified existing concurrency constructs. The table shows that the same concurrency idioms can be applied when retrofitting or designing concurrency.

Most importantly, by summarizing the data, we find that in 73.9% (162/(162+45+12)) of transformations, concurrency was retrofitted, i.e., bolt-on existing program elements, whereas in 20.5% of the cases concurrency was designed from scratch into program elements. In 5.4% of transformations, previously existing concurrency was modified.

These 5 case studies show that indeed it is possible to retrofit concurrency.

4.2 Q2: Is There a Process for Working with Concurrency?

Looking at the kinds of transformations that were applied between two versions of a program, we see that they are not random, but they are *consistent*. For example, we see that programmers improved the latency of EclipseSearch between versions 2.1.3 and 3.0, while they improved scalability and fixed correctness bugs between 3.0 and 3.3.2. Similarly, we see that Tomcat and MINA programmers focused on improving the scalability and correctness. In Eclipse DOM, the objective is to make the program thread-safe.

Thus, in programs that are being prepared for introducing concurrency later (e.g., DOM), programmers focus on making the code thread-safe. In programs where they are just introducing concurrency (e.g., Search, JUnit), the focus is on improving performance (e.g., latency or throughput). In programs that have been parallelized a long time ago, programmers focus on improving scalability. Thus, we see a recurring pattern: first make a program *right* (i.e., thread-safe), then make it *fast*, then make it *scalable*.

The data in Table 2 shows another trend. Even when programmers have a wide choice of transformations for achieving the same objective, they repeat the same kind of transformation, as if they were in the "mood" for making that transformation. For example, when MINA programmers were in the "mood" of using concurrent collections, they have made 32 changes of the same kind. This is consistent with the findings of Murphy-Hill et al. [19] that show that programmers repeat the same kind of refactoring within a short time-period. Since many of the transformations in these 5 case studies are repetitive, they are worth to automate.

4.3 Q2: How Do Current Tools Support These Transformations?

In the refactoring community there is a recent surge of interest [4, 5, 11, 14, 23, 27] on automating refactorings for concurrency and parallelism.

We survey this recent work in the light of the transformations and the trends that we noticed in the five case study programs. The good news is that the new breed of tools are automating transformations that appear in real world programs. For example, Concurrencer [4] supports refactorings for converting to `Atomic` classes, concurrent collections, and task parallelism for recursive divide-and-conquer algorithms. ReLooper [5] is a refactoring tool that enables loop parallelism using Java's `ParallelArray`. Reentrancer [27] improves thread-safety by converting global data into thread-local data, while Immutator [14] helps convert a mutable class into an immutable class. Relocker [23] helps programmer switch from built-in locks to more flexible locks.

While these tools automate refactorings for improving correctness [4, 14, 27], throughput [4, 5], and scalability [4, 23], none of them supports refactorings for improving latency. Refactorings for extracting a lengthy computation into an asynchronous computation are desperately needed. It appears that in three of the five case studies programmers initially use multi-threading for improving application responsiveness.

Refactoring tools need to support the whole lifecycle of concurrency. While current tools focus on introducing concurrency into sequential programs, the real-world data shows that developers revisit concurrency decisions they have made in the previous versions. As of now, only one tool [23] lets the programmer change existing built-in Java locks with more flexible locks. As more and more programs become concurrent, it is imperative to develop tools that let the programmer quickly explore several choices. This is extremely important for making existing parallel programs more scalable.

5. THREATS TO VALIDITY

Internal Validity.

One could ask whether our retrospective analysis and classification of the prallelizing transformations reflects the real intent of the developers of these projects. For three projects (Eclipse Search, Eclipse DOM, and JUnit) we confirmed with the developers that indeed our classification reflects their intent. For Tomcat, we were able to trace many of the concurrency changes to bug reports.

Also, with respect to noticing trends of transformations in real programs, we only looked at three versions that are relatively far apart. This can introduce noise. For example, the transition from JUnit 1.0 - 3.8.2 is an outlier since it is the only case where developers made changes with respect to all four concurrency objectives. Had we picked versions that were closer, we could have noticed changes that are more consistent. In addition, studying finer-grained deltas could shed even better light in answering whether concurrency is retrofitted or designed. We hypothesize that in some of the 20% of the program elements where concurrency appeared to be designed from the beginning we could find that concurrency was retrofitted, had we found the origin of that program element. Also, in the 5% of the transformations where concurrency was modified, we cannot tell whether concurrency was retrofitted or designed, unless we trace the origin of that program element. We leave such fine-grained analysis for future work.

External Validity.

We only looked at five projects, they were all developed in Java, and they were all open-source. Maybe other projects would not display the same program transformations. Also it may be the case that since several of these programs have been around for many years, retrofitting parallelism was the only option there. Maybe for newer projects currently being developed, parallelism is part of the design from the start.

We have several reasons to believe that the trends we saw in these projects could generalize to other projects. First, the projects that we looked at were developed by different teams, with contributors from a large open-source community. Therefore, there is a diversity of transformations in each project. Second, although in this study we only looked at Java programs, the transformations themselves are not language-specific. Even the transformations that involve `java.util.concurrent` library have equivalents in other parallel libraries (e.g., TBB [24] for C++ and TPL [26] for C#). We expect a similar range of transformations for other languages that use shared-memory, thread-based parallelism. Third, our classification is not complete: studying more projects would reveal new kinds of transformations. While we might discover new objectives like improving fault tolerance, we expect that the majority of the transformations would fall under one of the four major categories: improving responsiveness, throughput, or scalability, or fixing concurrency bugs. Fourth, our definitions of retrofitting or designing concurrency refer to individual program elements, not to whole programs. Concurrency can be designed even in a project that has been around for many years, if concurrency is introduced along with new program elements.

Reliability.

A detailed analysis of transformations at the class and method level is available online [2], along with the source code for the versions we analyzed. This allows an interested reader to replicate our results.

6. RELATED WORK

Mattson et al. [17] present a comprehensive catalog of patterns for parallel programming. Their catalog accomodates patterns and idioms for a large class of parallel programming architectures, including high-performance computers. Lea [15] and Goets et al. [12] wrote similar catalogs for concurrency patterns in Java. In contrast, our goal is not to document all patterns for writing concurrent programs, but we are interested in finding what are some of the patterns that are most commonly used in practice. Our focus is on the transformation process to convert a sequential Java program to concurrency, whereas patterns are often the end target of such transformations.

Some of the transformations that we identified have been long known to the high-performance computing (HPC) community. For example, *Loop Parallelism* is one of the traditional approaches in HPC, where the majority of algorithms are expressed in terms of iterative constructs. The OpenMP API was created primarily to support parallelization of loop-driven problems. OpenMP supports parallel loop execution and *reduction* operations that combine the partial results (e.g., summing the partial results).

Most empirical studies on concurrency have focused on finding the patterns for concurrency bugs. Chandra and Chen [1] collect 12 concurrency bugs from three applications. Farchi et al. [9] analyzed the concurrency-related bugs in code written by students. Lu et al. [16] conducted an extensive study of 105 real-world concurrency bugs from 4 open-source large projects. However, as far as we know, ours is the first empirical study to characterize concurrency-related transformations in real code.

The closest work to ours is an empirical study [21] conducted by Pankratius et al. on parallelizing four applications. While the authors describe the transformations they have introduced in the sequential application, the study does not intend to present the evolution of those case studies by tracing several versions, neither provides a taxonomy of transformations. Like us, the authors conclude that refactoring tools for parallelism can have a significant impact.

There is a plethora of tools for automatically detecting concurrency-related bugs. We mention Atomizer [10] for detecting atomicity violations and Eraser [22] for detecting dataraces in lock-based multithreaded programs.

Everaars et al. [7] report on their experience with converting a Fortran 77 sequential application into a concurrent application. They have used *coarse-grain* transformations to plug sequential modules into a new multi-threaded executable. The heart of their approach is finding and expressing the sequential modules, as well as the communication patterns between these modules and the framework. They use a language, MANIFOLD, to express the coordination and communication protocol.

Converting sequential programs to concurrency is much in the spirit of other efforts in the past for retrofitting architectural qualities: retrofitting type-safety in unsafe legacy code [20], converting legacy C code into C++ [8, 28], retrofitting security in unsecure legacy systems [3]. Although one can argue that such architectural qualities should be designed in the system, often they need to be retrofitted later on.

7. CONCLUSIONS

With the advent of the multi-core era, concurrency will have to be retrofitted into existing sequential applications. Our empirical study of concurrency-related transformations in five widely used open-source applications shows that in 73.9% of the cases concurrency was successfully retrofitted in existing program elements, in 5.4% of the cases, concurrency was modified in existing elements,

and in 20.5% of the cases it was designed into new program elements. Our findings suggest that programmers follow an orderly process where they focus on well defined objectives: to improve responsiveness, throughput, or scalability, or to fix concurrency errors.

We found that introducing concurrency is not a one-time event, but it is a continuous process. First, the incentive for using concurrency is to increase the responsiveness, then the throughput of an application. As the application matures and makes more use of concurrency, the predominant changes fall into fixing concurrency errors, fine-tuning, and improving the scalability. Given the importance and the length of such transformations, tool developers should consider (semi)automation for each stage in the concurrency lifecycle in order to improve programmer's productivity.

More studies and more data are needed to completely understand the process of transforming sequential code for parallelism. We hope that this paper encourages others to conduct new empirical studies.

8. ACKNOWLEDGEMENTS

This research was partially funded by Intel and Microsoft through the UPCRC Center at Illinois, and partially funded by DARPA contract HR0011-07-1-0023. The authors thank Adam Kiezun, Stephen McCamant, Angeline Lee, Derek Rayside, and anonymous reviewers for providing helpful suggestions. Danny thanks Monika Dig, his greatest supporter.

9. REFERENCES

[1] S. Chandra and P. M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 97–106. IEEE Computer Society, 2000.

[2] Accompanying online data with concurrency transformations. http://refactoring.info/studies/ConcurrencyTransformations.

[3] D. B. da Cruz, B. Rumpe, and G. Wimmel. Retrofitting security into a web-based information system. In *Software & Systems Engineering*, pages 35–38, 2003.

[4] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *31st International Conference on Software Engineering (ICSE)*, pages 397–407, 2009.

[5] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in java. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 793–794, New York, NY, USA, 2009. ACM.

[6] Eclipse Foundation. http://eclipse.org.

[7] C. Everaars, F. Arbab, and B. Koren. Using coordination to restructure sequential source code into a concurrent program. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 342, Washington, DC, USA, 2001. IEEE Computer Society.

[8] R. Fanta and V. Rajlich. Restructuring legacy c code into c++. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 77. IEEE Computer Society, 1999.

[9] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2. IEEE Computer Society, 2003.

[10] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[11] R. Fuhrer and V. Saraswat. Concurrency refactoring for x10. In *3rd ACM Workshop on Refactoring Tools*, pages 1–4, 2009.

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

[13] JUnit Testing Framework. http://junit.org.

[14] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Refactoring for Immutability. To Appear in *Proceedings of 33rd International Conference on Software Engineering (ICSE'11)*, Hawaii, USA, 2011.

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

[16] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, 2008.

[17] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[18] Apache MINA library. http://mina.apache.org/.

[19] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[20] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[21] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 53–60, 2008.

[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[23] M. Schaefer, M. Sridharan, J. Dolby, and F. Tip. Refactoring java programs for flexible locking. To Appear in *Proceedings of 33rd International Conference on Software Engineering (ICSE'11)*, Hawaii, USA, 2011.

[24] Threading Building Block (TBB) for C++. http://threadingbuildingblocks.org/.

[25] Apache Tomcat servlet container. http://tomcat.apache.org/.

[26] Task Parallel Library (TPL). http://msdn.microsoft.com/en-us/library/dd460717.aspx.

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

[28] Y. Zou and K. Kontogiannis. Migration to object oriented platforms: A state transformation approach. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 530– 539. IEEE Computer Society, 2002.