Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2008-053 September 5, 2008

How do programs become more concurrent? A story of program transformations.

Danny Dig, John Marrero, and Michael D. Ernst

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

How do programs become more concurrent? A story of program transformations.

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

Abstract

For several decades, programmers have relied on Moore's Law to improve the performance of their software applications. From now on, programmers need to program the multi-cores if they want to deliver efficient code. In the multi-core era, a major maintenance task will be to make sequential programs more concurrent. What are the most common transformations to retrofit concurrency into sequential programs?

We studied the source code of 5 open-source Java projects. We analyzed qualitatively and quantitatively the change patterns that developers have used in order to retrofit concurrency. We found that these transformations belong to four categories: transformations that improve the latency, the throughput, the scalability, or correctness of the applications. In addition, we report on our experience of parallelizing one of our own programs. Our findings can educate software developers on how to parallelize sequential programs, and can provide hints for tool vendors about what transformations are worth automating.

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. Other than for a few niche domains (e.g., embedded computing), in general programmers have relied on Moore's Law to improve the performance of software applications. However, because uniprocessors don't scale, the industry shifted to multi-core computers. This demands that programmers find and exploit parallelism in their applications, if they want to reap the same performance benefits as in the past.

Parallel programming and concurrency have been used for decades, but they were the mark of a small elite of programmers. From now on, parallel programming will be a skill that professional programmers will have to acquire. The dominant paradigm for parallel programming in desktop computing is shared-memory, thread-based parallelism. This paradigm adds extra complexity due to nondeterminism and increased 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 [10, 13]. However, most of the current desktop applications were not designed with concurrency in mind. In the multi-core era, a major maintenance task will be to retrofit concurrency into existing applications so that applications can utilize the additional cores. Although the topic of concurrency has been widely documented, little is known about the most common program transformations for retrofitting concurrency.

In this study, we create a taxonomy of the most common program transformations used to retrofit concurrency. We analyze qualitatively and quantitatively the concurrency-related program transformations in 5 open-source projects. Additionally, we report on our own experience with transforming one of our projects to make use of parallelism. Our goals are: (i) to inform software developers about the trend of program transformation they are going to perform during the transition to multi-core era, and (ii) to provide tool developers with empirical data about what program transformations are worth to (semi)-automate in the future.

To build the taxonomy, we manually analyzed several versions of 5 open-source Java projects: two core Eclipse [5] plugins, JUnit [12], Apache Tomcat server [19] and Apache MINA library [16]. 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 out that these parallelizing transformations fall into four categories: transformations that improve the *latency* (i.e., an application feels more responsive), transfor-

mations 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).

In summary, this paper makes the following contributions:

- presents an important problem: what are the program transformations that software maintainers will perform when transitioning to the multi-core era?
- presents the results of an empirical study about the common program transformations for retrofitting concurrency into real programs, and one experiment with parallelizing our own code.
- presents some practical applications for the reported findings: it educates software developers on how to port existing applications to use the multi-cores, and it provides hints for tool builders on which transformations are worth automating.

2 Experimental Setup

2.1 Case Studies

This section describes briefly each case study project, the versions that we analyzed, and the main concurrency-related themes in those versions. We track two major versions for which we know that there are concurrency-related changes. After the version with the major concurrency-related changes, we track changes in one more version to find whether the previously introduced concurrency constructs changed due to bug fixes.

We selected case studies that cover the whole spectrum 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.

2.1.1 Eclipse Search plugin

`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 (the current stable release). 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.

2.1.2 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 plug-ins 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).

2.1.3 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.

2.1.4 Apache MINA library

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.

2.1.5 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 from Sun Microsystems. We studied versions 4.1.1, 5.5, and 6.0. These versions fixed several concurrency-related bugs and improved the scalability.

Table 1 presents some general statistics about the case-study programs.

2.2 Set up of the experiment

The projects that we analyzed ranged from a few thousand lines of code to several million lines of code (e.g., Eclipse is roughly 2-million LOC). A thorough manual analysis of *all* source code 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

	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
Lines of Code [KLOC]	11	21	24	27	52	62	3	8	10	402	489	338	40	40
# Synchronized Blocks	4	34	42	12	110	121	15	20	18	921	1152	1108	422	211
# Statement/Whole 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"/custom 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: LOC, total number of synchronized blocks, how many synchronized block at the statement level vs. whole method level, how many statement synchronized blocks use "this" lock vs. custom lock.

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 loaded different versions of the project under analysis in the Eclipse IDE. We used Eclipse's search engine and looked for the *fingerprints* of concurrent code. In Java, these include `synchronized` blocks, references to `java.lang.Thread`, references to classes in `java.util.concurrent`, and references to utility classes that run tasks in the UI event thread (e.g., `javax.swing.SwingUtilities` in the Swing toolkit or `org.eclipse.swt.widgets.Display` in the SWT toolkit).

- For those methods and classes that we found (either through release-notes or code search) containing concurrency artifacts, we manually analyzed how they changed between different versions. We recorded the kinds of changes (qualitative) and the number of distinct such changes (quantitative).

3 Concurrency Program Transformations

Subsection 3.1 presents the four objectives for making concurrency-related program transformations: improving latency, throughput, scalability, and correctness. Our hypothesis is that any particular concurrency-related transformation strives 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. We illustrate each transformation with a real example. 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.

Subsection 3.4 concludes by addressing the threats to validity.

3.1 Objectives for Concurrency Transformations

We hypothesize that there are four objectives for making concurrency-related transformations.

Improve Latency. Latency measures how long it takes from the moment of asking for the result of a computation until the result is available. To improve user satisfaction, the UI should feel responsive, even when executing long-running computations. For example, in Eclipse, searching for all references to a program element is a long-running operation, especially if the project contains millions of lines of code. 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 computation finishes.

Improve Throughput. Throughput measures the amount of results that are computed per unit of time. To improve throughput, programmers decompose the computation so that multiple units of execution can process the data in parallel.

Improve Scalability. It is desired that an application's performance scales up when adding more computational units (e.g., more processors). Amdahl's Law quantifies the maximum speed up of a parallelized program. The upper-bound for the speed up is inverse proportional with the percentage of computation that needs to execute sequentially. In concurrent programs, synchronization constructs

increase the sequential part of the computation. Programmers fine-tune the fraction of sequential 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*. Second, changes to an object's state need to be *visible* to other threads. In shared-memory, thread-based systems, synchronization blocks are used to achieve both atomicity and visibility.

3.2 Examples of Concurrency-related Transformations

3.2.1 Improving the Responsiveness

Create 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. To prevent blocking the UI while tests run, `TestRunner` creates and spawns a new thread in which all tests run.

Method Object with Runnable The computation in a method is encapsulated in an object whose API offers a `run` method. Depending on the value of a parameter passed to `run`, the method 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);
     }
    });
```

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, internalNameProperty());
      }
    }
  }
  return this.typeName;
}
```

Delegating thread safety to the Event Dispatching Thread. As 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. Graphical toolkits like Swing or SWT use a dedicated *event dispatching thread* for handling GUI events (e.g., mouse click, pressed button). Applications can delegate thread-safety by enforcing that all updates happen in the event 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 the methods `asyncExec(Runnable)` and `syncExec(Runnable)` for scheduling the runnable in the SWT event thread. Eclipse's Search plugin uses both methods.

3.2.2 Improving the Throughput

Introduce Loop Parallelism Loop-parallelism [15], is an idiom used to parallelize iterations of a computationaly 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();
}
```

Task parallelism Task parallelism [15] is used when the computation can be decomposed into a collection of tasks that can execute concurrently. The tasks can execute either the same code or different code. Task parallelism is often employed in servers that receive requests and process them in separate threads.

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 copied collection. 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 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 `Hashtable` implementation 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.

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.

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.

Change lock type 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 type 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 findings

we summarize our findings in Tables 2, 3, and 4. These tables are *descriptive*: they describe transformations that happened in real code.

Table 2 summarizes the kinds of transformations and how many times they appear in the case studies. For each case-study we report only the changes (deltas) made in the later version. That is, we do not report the concurrency idioms that were already present in the previous version. The changes that we report in Table 2 include only edits to methods and classes that are present in both versions.

Adding synchronized blocks is a transformation that cross-cuts most other transformations, since syhchronization blocks are the primary construct from which the other transformations are made of. Therefore, we make a distinction between adding synchronized blocks as a side effect of a transformation vs. adding synchronization blocks because the previously parallelized code was unsufficiently synchronized. In Tables 2, 3, and 4 we report only the latter; these changes are corelated with bugs reports.

Table 3 presents concurrency idioms only in the methods and classes that were added in the later version.

	JUnit 1.0 → 3.8.2	Tomcat 4.1 → 5.5
Delegating to Event Dispatch	9	
Use Atomic Classes		27
Creating new Threads	3	
Use ThreadLocal		2
Introduce Loop Parallelism	1	

Table 3. Parallelizing idioms for program elements that are *added* in the later version.

	Latency	Throughput	Scalability	Correctness
Method Object with Runnable	4			
Remove synchronized block	1			
Reducing Lock Duration	2			
Copy-then-iterate	7			
Delegating to Event Dispatch	16			
Thread-safe Lazy Initialization	78			
Creating new Threads	2	1		
Loop Parallelism		1		
UseConcurrent Collections			33	
Use Atomic Classes			42	
Add synchronized block				18
Coarsen synchronized block				6
Use ThreadLocal fields				2
Change Lock Type				3

Table 4. Correlation of transformations with objectives in the 5 case studies. The data shows how many times each transformation was employed for a particular objective.

Table 4 links these transformations with objectives for making concurrency-related transformations. For each transformation we report how many times it was employed to achieve an objective in the 5 case-studies.

We make several observations from this study. First, the same kind of transformation can be used to achieve different performance objectives (latency, throughput, scalability). Second, there is a trade-off between correctness and improving performance. The transformations that achieve correctness involve adding new synchronized blocks or coarsening existing ones, which in turn increases the fraction of code that executes sequentially. Third, retrofitting concurrency is a process. In applications that are just beein parallelized (e.g., Eclipse Search and DOM), the objective is to improve latency. In applications that have been parallelized for several versions (e.g., Tomcat and MINA), the objective is to fix concurrency errors and improve scalability.

3.4 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

	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
Delegating to Event Dispatch	7						
Method Object with Runnable	4						
Introduce Loop Parallelism							
Introduce Task Parallelism							
Change Lock Type		3					
Reducing Lock Duration				2			
Copy-then-iterate	5			2			
Thread-safe Lazy Initialization			78				
UseConcurrent Collections						1	32
Use Atomic Classes						10	5
Creating new Threads							
Remove Synchronized Block		1					
Coarsen Synchronized block					6		
Add Synchronized block		3			15		

Table 2. Parallelizing transformations (*edits*) for program elements that exist in both versions.

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.

External Validity We only looked at 5 projects, and they were all developed in Java. Maybe other projects would not display the same program transformations. 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 (excluding the ones that involve `java.util.concurrent`) are not language-specific. 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. However, these transformations would fall under one of the four major categories: improving responsiveness, throughput, or scalability, or fixing concurrency bugs.

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

4 Experience with converting one application

4.1 Concurrency Transformations

To learn more about the transformations a software developer performs when parallelizing existing code, we parallelized one of our own projects, RefactoringCrawler [4]. RefactoringCrawler is a tool for inferring refactorings that happened between two versions of a library. RefactoringCrawler currently detects seven types of refactorings, including renamings, changes of method signatures, moved methods, etc.

We first ran the Eclipse's profiler to find the parts of the code that are most computationally intensive. The profiler identified the method `findSimilarMethods()` as one of the hot spots. This method iterates over all pairs of source methods from the two versions of the input library and finds which methods have a textual resemblance (see Fig. 1(a)). Depending on the size of the library under analysis, there can be tens of thousands of pairs that need to be analyzed. Since there are no dependencies between the iterations of the loop, we parallelized this loop using the *Loop Parallelism* transformation.

Fig. 1(b) shows how we parallelized the code. Spawning a thread for each loop iteration is not effective: thread creation and destruction dominates the computation for each iteration. Moreover, running thousands of threads on a desktop computer with a small number of processors decreases the performance due to the cost of context-switching between threads. Instead, we allocate a number of threads equal to the number of available processors, so that each processor runs only one single thread.

Next, for each thread, we allocate a quota of the data to be analyzed. This quota is equal to the size of the data to be analyzed divided by the number of available processors.

We encapsulate the computation inside a `FutureTask`, which is one of the utility concurrency APIs provided by the new `java.util.concurrent` Java standard library. `FutureTask` describes a result-bearing computation [13]. The computation is implemented within a `Callable`, the result-bearing equivalent of `Runnnable`. Method `findQuotaMethods()` does the main work: it iterates over a subset of the methods in the two versions of a library and determines whether they are textually similar. This method assembles the partial result inside a temporary

collection, `result`.

Once it launches threads for all `FutureTasks`, the code waits for all partial computations to finish. The method `FutureTask.get()` blocks until the task finishes. From the partial results, the code assembles the final result.

The second transformation we made involved *task-parallelism*. RefactoringCrawler detects different kinds of refactorings in a sequential order, first detecting renamed methods, then methods whose signatures changed, etc. (see Fig. 2(a)). We changed RefactoringCrawler so that different detections run concurrently in different threads (see Fig. 2(b)).

To ensure that all threads finish execution before displaying the results, we use `CountDownLatch`, a synchronizer utility class from the new `java.util.concurrent` package. The latch is initialized with the number of threads that will run the tasks; the termination of each task decrements the counter. Calling `wait` on the latch forces the execution to wait until all tasks are finished.

In addition, these concurrently-running threads read and write some common data-structures (not shown in the figure). Therefore, we made the code that accesses these data structures thread-safe. Namely, we used synchronization blocks to guard the read and write accesses to the shared data-structures. We used fine-level granularity locks and hold the locks only in the critical sections. In total we used 11 synchronization blocks.

4.2 Performance improvements

To measure the performance effects of these transformations, we used RefactoringCrawler to detect refactorings in JHotDraw [11], a 20KLOC open-source framework for building graphical editors. RefactoringCrawler found 24 refactorings in JHotDraw.

As a metric for performance improvement we use the $SpeedUp$ of the parallelized code over the sequential code. We compute $SpeedUp$ as the fraction of the total time taken to detect refactorings with the original, sequential tool, and the total time to detect refactorings with the parallelized tool. We ran RefactoringCrawler on a Linux desktop PC with 4 cores. The parallelized RefactoringCrawler correctly detects the same refactorings as the sequential tool, while achieving of 2.41x speedup when running on 4 cores.

4.3 Lessons learned

- When parallelizing code, *first make it right, then make it fast*. Safety always comes first, before improving the performance. In an eager attempt to improve the performance, we first missed some of the thread-safety properties. This was in part due to some hidden accesses to the shared data. The whole conversion of code took the first author 3 days. However, most of the time was spent tracing and fixing some data races.

- The `java.util.concurrent` APIs in the new java libraries improved our productivity. This package contains convenient abstractions for synchronization among different threads (e.g., barriers, latches, future tasks) and thread-safe collections (e.g., `ConcurrentHashMap`).

- One challenge is to improve scalability without penalizing cases when the code runs on one single processor. Without careful examination, it is easy to degrade the performance of the parallelized code on a single processor. For example, using the thread-safe `Vector` or the thread-safe `Collections.synchronizedXYZ` wrapper around a thread-unsafe collection can degrade the application's performance on a single processor.

5 Related Work

Mattson et al. [15] 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 [13] and Goets et al. [10] 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. [8] analyzed the concurrency-related bugs in code written by students. Lu et al [14] 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.

There is a plethora of tools for automatically detecting concurrency-related bugs. We mention Atomizer [9] for de-

tecting atomicity violations and Eraser [18] for detecting dataraces in lock-based multithreaded programs.

Everaars et al. [6] 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 [17], converting legacy C code into C++ [7, 20], 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.

6 Conclusions

With the advent of the multi-core era, concurrency will have to be retrofitted into existing sequential applications. In this study we look at some of the most common ways to retrofit concurrency into 5 widely used open-source applications. Our findings confirm our hypothesis: programmers transform existing programs in order to improve latency, throughput, or scalability, or to fix concurrency errors. Our own experience with parallelizing one application taught us that safety and correctness comes before improving the performance.

We found out that retrofitting concurrency is not a one-time event, but it is a continuous process. First, the inceptive for retrofitting 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.

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] Concurrency Transformations. http://people.csail.mit.edu/dannydig/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, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *ECOOP'06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer Verlag, 2006.

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

[6] 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.

[7] 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.

[8] 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.

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

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

[11] JHotDraw Framework. http://www.jhotdraw.org.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

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

[20] 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.

```
List findSimilarMethods() {

List similarMs = new ArrayList();
List methodsInV1 = ...
List methodsInV2 = ...

forEach (m1 : methodsInV1)
  forEach (m2 : methodsInV2)
    if (m1.isSimilar(m2))
      similarMs.add(<m1,m2>);

return similarMs;
}
```

(a)

```
List findSimilarMethods() {

List similarMs = new ArrayList();
List methodsInV1 = ...
List methodsInV2 = ...

numProc = Runtime.getRuntime().availableProcessors();
quota = methodsInV1.size()/numProc;

for (i < numProc){
  FutureTask future = new FutureTask(new Callable(){
     List call() {
       return findQuotaMethods(i, (i+1)*quota);
     }

     List findQuotaMethods(start, end){
        result = new ArrayList();
        quotaList = methodsInV1.sublist(start,end);
        forEach (m1 : quotaList)
          forEach (m2 : methodsInV2)
            if (m1.isSimilar(m2))
               result.add(<m1, m2>);
     }});
  tasks.add(future);
  thread = new Thread(future);
  thread.start();
}

for (j < numProc){
   partialResult = tasks.get(i).get();
   similarMs.addAll(partialResult);
}

return similarMs;
}
```

(b)

Figure 1. Loop Parallelism transformation in RefactoringCrawler: (a) depicts the sequential program, (b) depicts the parallelized program.

```
detectRefactorings(){

detectRenameMethod(utility,
        v1Graph, v2Graph);

detectChangedSignature(utility,
         v1Graph, v2Graph);

detectMovedMethod(utility,
             v1Graph, v2Graph);

detectPullUpMethod(utility,
           v1Graph, v2Graph);

displayAllRefactorings();
}
```

(a)

```
detectRefactorings(){

endGate = new CountDownLatch(4);

renameThread = new Thread(){
   run () {
     try {
     detectRenameMethod(utility,
         v1Graph, v2Graph);
     } finally {
        endGate.countDown();
   }
  };
renameThread.start();

changeSigThread = new Thread(){
    run () { ....}};
changeSigThread.start();

moveMethThread = new Thread(){
    run () { ....}};
moveMethThread.start();

pullUpThread = new Thread(){
    run () { ....}};
pullUpThread.start();

endGate.wait();
displayAllRefactorings();
}
```

(b)

Figure 2. Task parallelism in RefactoringCrawler. Figure depicts code (a) before and (b) after the transformation.