Transformations for Concurrency

Danny Dig
University of Illinois

John Marrero, Michael Ernst

IWMSE – May 21st
How Do Programs Become More Concurrent?

Use parallelism/concurrency for performance or to enable new applications and services (better user experience)

Must concurrency be designed into a program?

Can it be retrofitted later?

What are common transformations for retrofitting? Do they belong to certain categories?

Can we automate these transformations?
Empirical Study Setup

Study concurrency-related transformations in 5 open-source projects: Eclipse Search, Java DOM, JUnit, Tomcat, MINA

Read version release notes to identify a version with major concurrency ($V_{conc}$)

For each project, manually analyze 3 major version releases ($V_{prev}$, $V_{conc}$, $V_{next}$)

Searched code for “concurrency fingerprints” (e.g., synchronized, Thread, concurrent packages)

Compared the source code
Q: Random Transformations?

A: Objectives are to improve:
- responsiveness
- throughput
- scalability
- thread-safety
Examples of Transformations

Improving Responsiveness
Extract Lengthy Computation into Async
- use separate Thread
- delegate to Event Dispatching Thread
- encapsulate into a runnable method

Improving Throughput
- loop parallelism
Examples of Transformations

Improving Scalability
- reducing duration of the held lock
- copy (synchronized) then iterate (unsynchronized) over snapshot
- use Atomic classes (e.g., AtomicInteger)
- use concurrent collections (e.g., ConcurrentHashMap)

Improving Thread Safety
- add synchronized block
- coarsen synchronized block
- thread-safe lazy initialization
- change lock object from this of the parent class to intrinsic lock of field
Q: Concurrency Retrofitted or Designed?

<table>
<thead>
<tr>
<th>Code</th>
<th>existing</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency changes</td>
<td>modified</td>
<td>N/A</td>
</tr>
<tr>
<td>new</td>
<td>retrofitted</td>
<td>designed</td>
</tr>
</tbody>
</table>

**Retrofitted:** Tomcat `accessCount` exists in both $V_{\text{prev}}$, $V_{\text{conc}}$

- in latter version `int` $\rightarrow$ `AtomicInteger`

**Designed:** JUnit `ActiveTestSuite` exists only in $V_{\text{conc}}$

- uses threads to launch new tests

**Modified:** Eclipse `addQuery` exists in both versions

- synchronized on different locks
Results: Concurrency is Mostly Retrofitted

- Retrofitted: 74%
- Designed: 21%
- Modified: 5%
Q: Concurrency Process?

First incentive to improve responsiveness, then throughput, then scalability

Programmers repeated the same transformation, as if they were in the “mood”
- 32 transformations of the same kind in MINA

Transformations are worth to automate
Retrofitting Parallelism is Tedious
Q: Can We Automate Some Transformations?

Most frequent scenario: retrofit parallelism incrementally
- sequence of small steps (refactoring)
- always maintain a working, deployable version

Still tedious, because programmer:
- changes many LOC
- ensures non-interference of parallel operations
Fully Automatic vs Interactive Parallelization

Benefits:

- programmer has domain knowledge
- combine strengths of programmer and tool (search, remember, compute)

Challenges:

- efficient (keep programmer engaged)
- handle complex general-purpose programs

Limited applicability
(mostly dense-matrix)
- hard for coarse-grained parallelism

Programmer + Tool >> Tool
Our Refactoring Toolset: refactoring.info/tools

Refactorings for **thread-safety**
- make class immutable [attend ICSE'11 talk this Wed]
- convert to Atomic* classes [ICSE'09]
- use concurrent collections (e.g., ConcurrentHashMap) [ICSE'09]
- inferring region annotations [ASE'09]

Refactorings for **throughput**
- loop parallelism via ParallelArray [Under Submission]
- parallel recursive divide-and-conquer [ICSE'09]

Refactorings for **scalability**
- Atomic*, concurrent collections [ICSE'09]
Other Refactoring Tools

Reentrancer [Wloka et al.]: converts global into thread-local

Relocker [Schaefer et al.]: converts from intrinsic locks to more flexible locks (e.g., reentrant locks, read-write locks)

Concurrency refactoring in X10 (Fuhrer et al.)

Current tools focus on improving throughput, scalability, thread-safety
- blind spot on improving responsiveness
Threats to Validity

Internal Validity
- reflect the intent of the developers? Confirmed with the developers
- only looked at major versions? A fine-grained origin analysis could refine the results

External Validity
- do findings in 5 projects (all in Java) generalize?
  - But, open-source, globally developed
  - similar concurrency features in C# and C++

Reliability
http://refactoring.info/studies/ConcurrencyTransformations
Related Work

Patterns for parallel programming [Matson et al.], [Lea], [Goetz et al.]

Empirical studies on bugs in concurrent programs [Chandra], [Farchi et al.], [Lu et al.]

Experiences with parallelizing applications [Everaars et al.], [Pankratius et al.]
Conclusions

Successful programs can be retrofitted for parallelism
- more than 70% of concurrency-related transformations

Transformations are not random, but they fall into distinct objectives: improving responsiveness, throughput, scalability, correctness

Introducing concurrency is a process: transformations are repetitive, worth to automate

More studies like this one are needed