



IBM Research

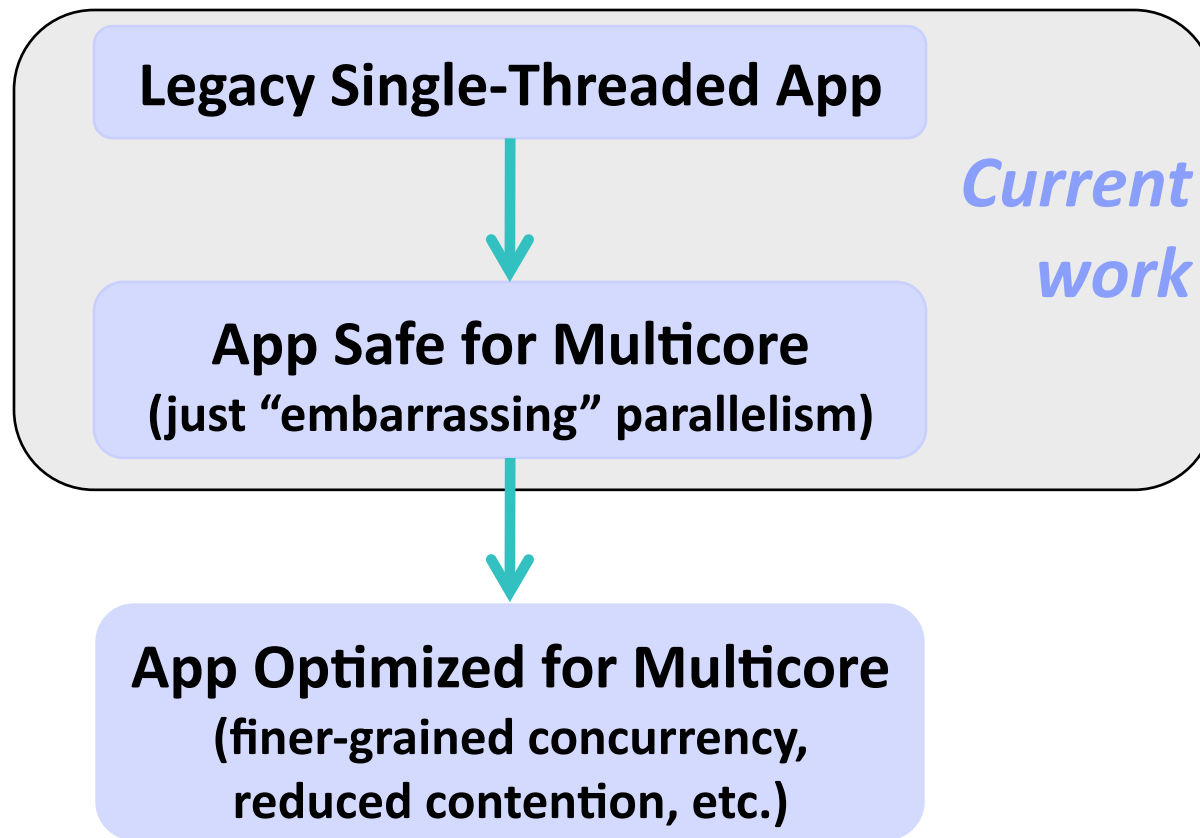
## Refactoring for Reentrancy

Jan Wloka  
IBM Rational

Manu Sridharan, Frank Tip  
IBM Research

ESEC / FSE 2009

## Vision: A Migration Path to Multicores



***Correctness before performance***

## Challenge: Lack of Reentrancy

### Definition of reentrant program

*distinct program executions do not affect each other, sequentially or concurrently*

**Mutable global state** the key barrier to reentrancy

## Running Example of Non-Reentrancy

```
class Config {
    static final Map conInfo = new HashMap();
    static void setOption(String name, String value) {
        conInfo.put(name, value);
    }
    static String getOption(String name) {
        return conInfo.get(name);
    }
}

test1() { Config.setOption("foo", "bar"); ... }
test2() { assert Config.getOption("foo") == null; ... }
```

**Problem:** `test2` fails if run after `test1`, but passes if run alone

**Race conditions** on `conInfo` if tests run in parallel

## Existing Approaches to Reentrancy

### Approach # 1: **get rid of mutable global state**

- A possible solution for new projects
- But, for existing code, too intrusive and hard to automate

### Approach # 2: **separate OS processes / classloaders**

- Strong isolation between executions
- But, further optimization difficult (e.g., adding shared state)

## Our Approach: Thread-Local State

**Idea**: Replace all mutable globals with thread-local variables

- Implicit per-thread copy of each variable
- Available on most threading platforms (Java, pthreads, etc.)

**Result**: code is reentrant if executions run in fresh threads

**Key advantage**: *enables local optimizations*

- Selectively replace thread locals with shared state + locking
- *Correctness before performance*

**Challenge**: *handling real-world language / programs*

- In particular, initialization and library usage

## Contributions

### Reentrancer Design

- Goal: *practical tool for real-world programs*
- Mostly automated
- Targets Java, but applicable to other languages

### Reentrancer Implementation and Evaluation

- Runs on *realistic benchmarks* (up to 83 KLOC)
- Fixed observed reentrancy issues
- Enabled parallel speedup for 3 of 5 benchmarks

## Reentrancer Phases

### 1. Check preconditions

- a) Detect potential violations (**automatic**)
- b) Inspect / fix warnings (**manual**)

### 2. Make code reentrant (**automatic**)

- a) Encapsulate mutable global state
- b) Introduce lazy initialization
- c) Introduce thread locals

### 3. Create fresh thread per execution (**manual**)

## Preconditions 1: Static Initialization

**Issue**: thread locals must be initialized as globals were

**Challenge**: initializers can have complex and fragile behavior



```
class A { static final int x = B.y + 1; }  
class B { static final int y = A.x + 2; }
```

**Solution**: change to **explicit lazy initialization**

- Makes thread-local introduction a separate, simple phase
- *Not* behavior preserving for “fragile” dependences like above
- Precondition check warns if behavior may change
  - Indicates badness in general

## Preconditions 2: Library Usage

**Issue:** Library usage can break reentrancy (e.g., file I/O)

### **Challenges**

1. Cannot refactor library code
2. Hard to detect all non-reentrant library routines

**Solution:** **warn about calls to blacklisted methods**

- Doesn't require expensive whole-program analysis
- Catches most common cases
- Always double-check with test suite!

## Running Example

```
class Config {  
    private static final Map conInfo = new HashMap();  
    private static final String versionStr = "2.0";  
    static {  
        conInfo.put("version", versionStr);  
    }  
    static void setOption(String name, String value) {  
        System.out.println(name + ":" + value);  
        conInfo.put(name, value);  
    }  
    static String getOption(String name) {  
        return conInfo.get(name);  
    }  
}
```

## Step 1(a): Detect Violations (automatic)

```
class Config {
    private static final Map conInfo = new HashMap();
    private static final String versionStr = "2.0";
    static {
        conInfo.put("version", versionStr);
    }
    static void setOption(String name, String value) {
        System.out.println(name + ":" + value);
        conInfo.put(name, value);
    }
    static String getOption(String name) {
        return conInfo.get(name);
    }
}
```

## Step 1(b): Fix Warnings (manual)

```
class Config {
    private static final Map conInfo = new HashMap();
    private static final String versionStr = "2.0";
    static {
        conInfo.put("version", versionStr);
    }
    static void setOption(String name, String value) {
        // System.out.println(name + ":" + value);
        conInfo.put(name, value);
    }
    static String getOption(String name) {
        return conInfo.get(name);
    }
}
```

## Step 2(a): Encapsulate Mutable Globals (automatic)

```
class Config {
  private static final Map conInfo = new HashMap();
  private static final String versionStr = "2.0";
  static {
    conInfo.put("version", versionStr);
  }
  static void setOption(String name, String value) {
    conInfo.put(name, value);
  }
  static String getOption(String name) {
    return conInfo.get(name);
  }
}
```

## Step 2(a): Encapsulate Mutable Globals (automatic)

```
class Config {
    private static final Map conInfo = new HashMap();
    private static final String versionStr = "2.0";
    static Map getConInfo() {
        return conInfo;
    }
    static {
        getConInfo().put("version", versionStr);
    }
    static void setOption(String name, String value) {
        getConInfo().put(name, value);
    }
    static String getOption(String name) {
        return getConInfo().get(name);
    }
}
```

## On Mutability

If state is immutable, no need to transform

Immutable means deeply immutable [Tschantz&Ernst, OOPSLA05]

- all abstract state of object
- **final** is not enough

Simple type-based analysis to detect immutability

## Step 2(b): Introduce Lazy Initialization (automatic)

```
class Config {  
    private static Map conInfo = new HashMap();  
    private static final String versionStr = "2.0";  
    static Map getConInfo() {  
        return conInfo;  
    }  
    static {  
        getConInfo().put("version", versionStr);  
    }  
    ...  
}
```

## Step 2(b): Introduce Lazy Initialization (automatic)

```
class Config {
    private static Map conInfo;
    private static final String versionStr = "2.0";
    static Map getConInfo() {
        lazyInit(); return conInfo;
    }
    static boolean initRun = false;
    static void lazyInit() {
        if (!initRun) {
            initRun = true;
            conInfo = new HashMap();
            getConInfo().put("version", versionStr);
        }
    }
    ...
}
```

## Step 2(c): Introduce Thread Locals (automatic)

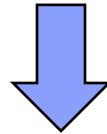
```
class Config {
  private static Map conInfo;
  private static final String versionStr = "2.0";
  static Map getConInfo() {
    lazyInit(); return conInfo;
  }
  static boolean initRun = false;
  static void lazyInit() {
    if (!initRun) {
      initRun = true;
      conInfo = new HashMap();
      getConInfo().put("version", versionStr);
    }
  }
  ...
}
```

## Step 2(c): Introduce Thread Locals (automatic)

```
class Config {
    static ThreadLocal<Map> conInfo = new ThreadLocal<Map>();
    private static final String versionStr = "2.0";
    static Map getConInfo() {
        lazyInit(); return conInfo.get();
    }
    static ThreadLocal<Boolean> initRun =
        new ThreadLocal<Boolean>();
    static void lazyInit() {
        if (!initRun.get()) {
            initRun.set(true);
            conInfo.set(new HashMap());
            getConInfo().put("version", versionStr);
        }
    }
    ...
}
```

## Step 3: Create Fresh Thread per Execution (manual)

```
test1() { Config.setOption("foo", "bar"); ... }  
test2() { assert Config.getOption("foo") == null; ... }
```



```
test1() {  
    runInFreshThread(new Runnable() { public void run() {  
        Config.setOption("foo", "bar"); ...  
    } });  
}  
test2() {  
    runInFreshThread(new Runnable() { public void run() {  
        assert Config.getOption("foo") == null; ...  
    } });  
}
```

## Experiments: Implementation

### Precondition Checker

- Built on WALA (<http://wala.sf.net>)
- Emits warnings for violations

### Code Refactoring

- Extension to Eclipse JDT
- Handles many Java corner cases (interface fields, boxing/unboxing)

## Experiments: Methodology

Five single-threaded benchmarks, up to 83 KLOC

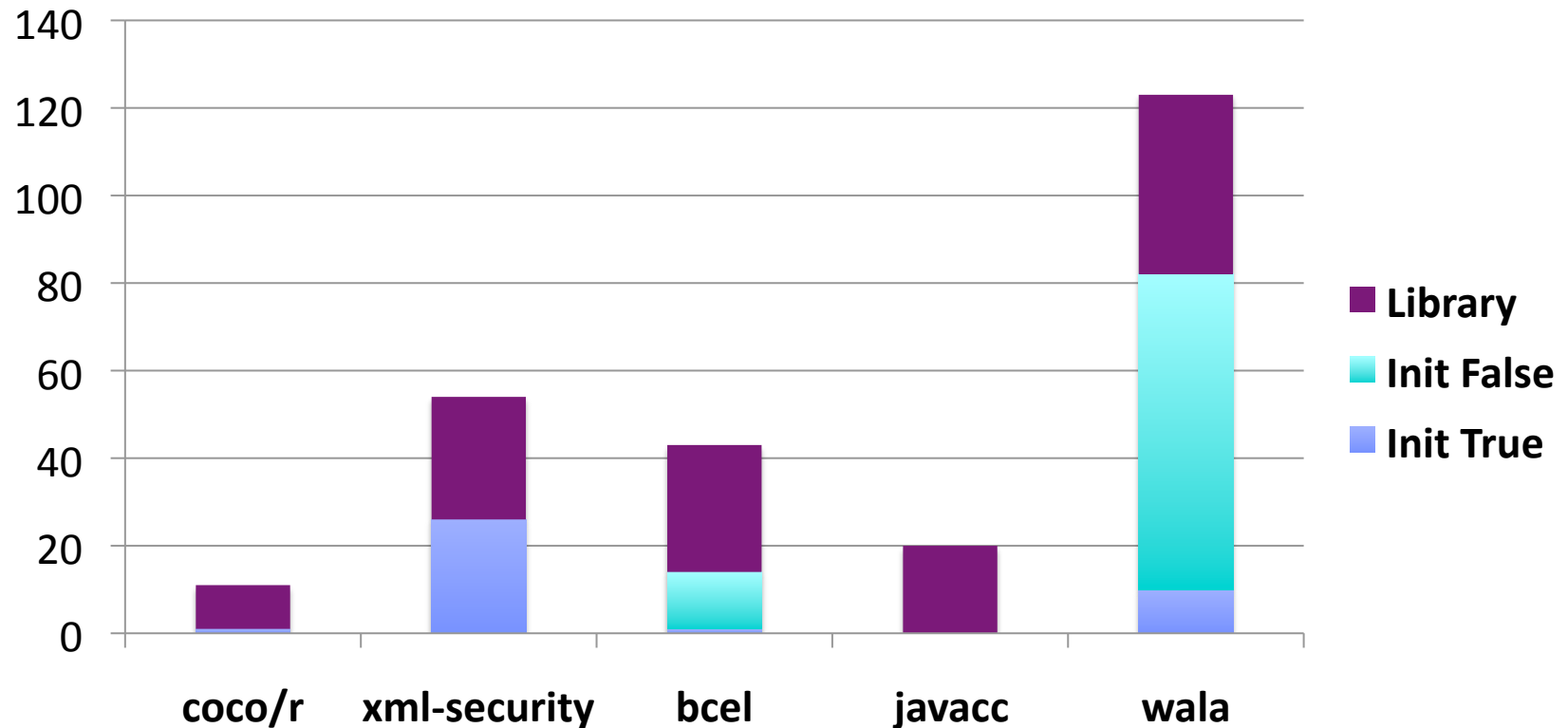
For each benchmark:

1. Establish reentrancy problem(s)
2. Inspect / fix precondition check warnings
3. Run refactoring
4. Ensure test suite passes
5. Ensure reentrancy problem(s) fixed
6. Measure performance

## Experiments: Key Questions

1. Precondition violations
  - a) How many?
  - b) How many false positives?
2. How much code is changed by refactoring?
3. Performance of running test suite
  - a) Before vs. after on one core?
  - b) Before vs. after on two cores?

## Results: Precondition Violation Warnings

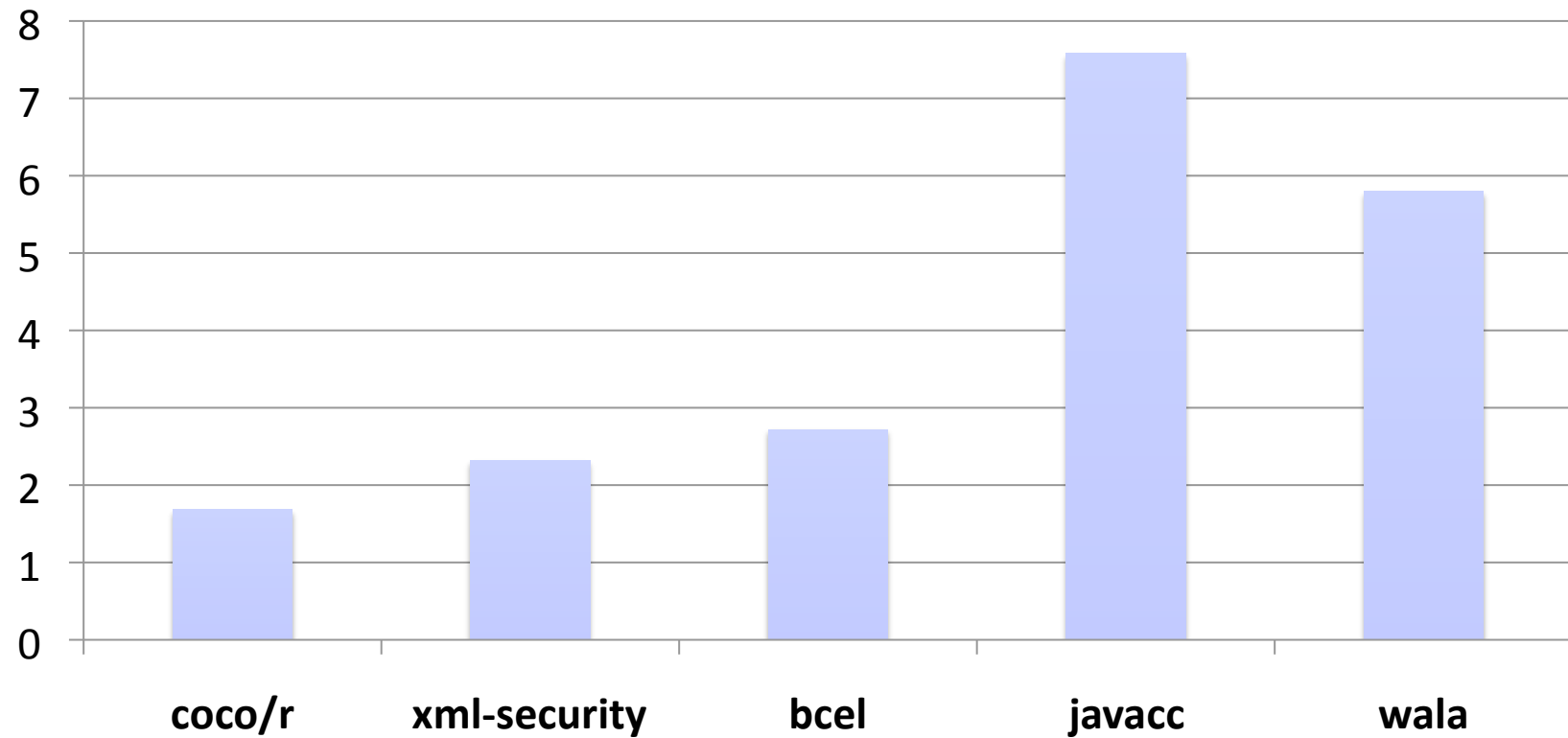


False positives due to weak immutability inference

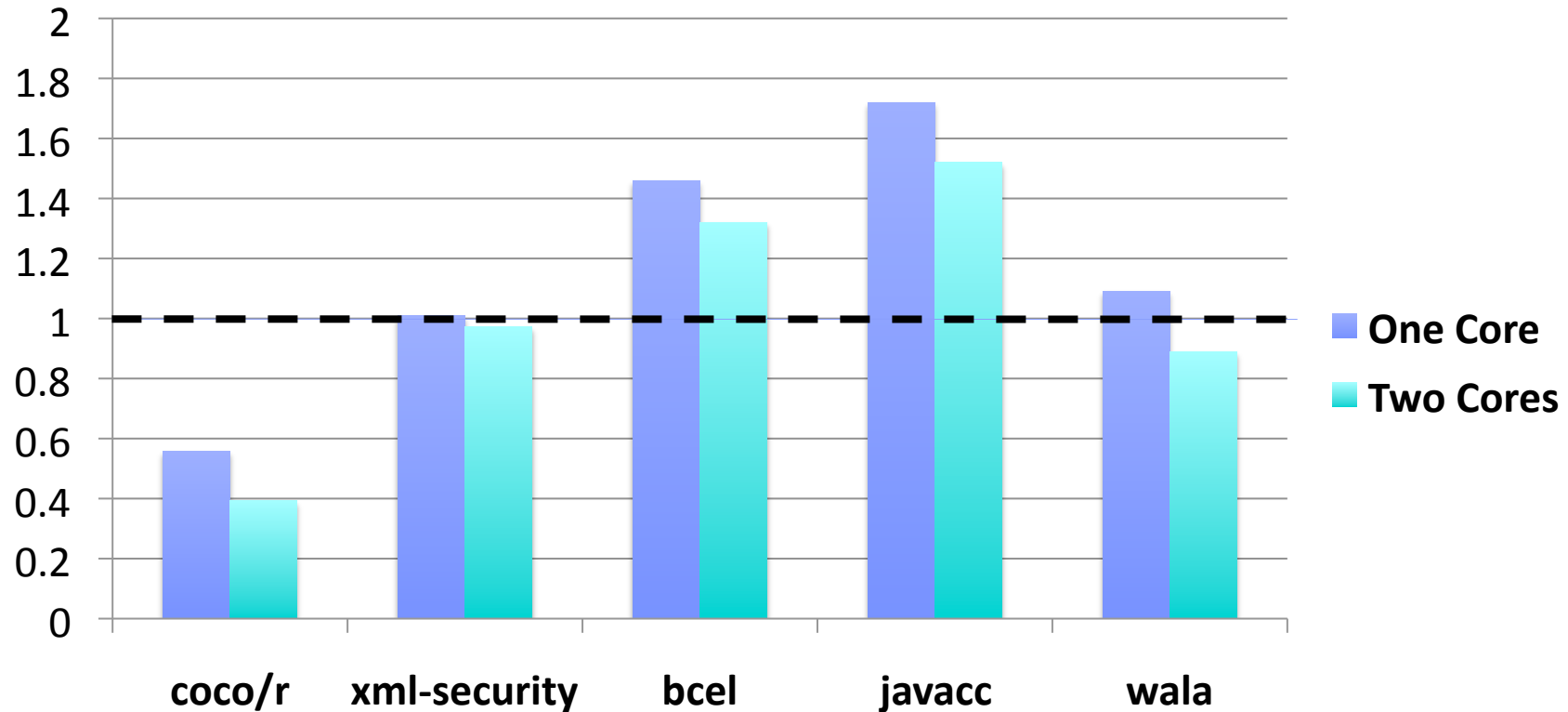
Most warnings benign

## Results: Code Churn

KLOC changed



## Results: Test Suite Runtime After Refactoring



Parallel speedup for 3 of 5 benchmarks

coco/r anomaly: removed fresh class loaders

## Related Work

Refactoring to `java.util.concurrent`  
[DME09]

Cilk++ hyperobjects [FHLL09]

Java immutability [TE05,ZPAAKE07,QTE08]

## Conclusions

Reentrancy via **changing globals to thread-locals**

- **Correctness before performance**

Prototype implementation works and enables parallel speedups

### Future work

- Better immutability inference
- More incremental workflow



**Thanks!**