THREAD LOCALITY

Numerous uses:
- Simplify reasoning, enable optimizations (worst-case analysis), concurrent real-time programming, etc.
- Minimal support through the ThreadLocal API
- thread-local is unsafe, little programmer aid
- GCing of thread-local data, etc.

Java:
- Using the ThreadLocal API

Plain Old Java:
```java
class Foo {
    private Bar bar = null; // Should be thread-local!!
    void foo() {
        ...
    }
}
```

Using the ThreadLocal API:
```java
class Foo {
    private ThreadLocal<Bar> bar = new ThreadLocal<Bar> ()
    synchronized void foo() {
        // Local!
        // Should be thread-local
        bar = null;
    }
}
```

LOCI:
SIMPLE THREAD-LOCALITY FOR JAVA
Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, Jan Vitek
Purdue University
23rd European Conference on Object-Oriented Programming
Genoa, Italy 2009

• Numerous uses:
- Simplify reasoning, enable optimizations, concurrent real-time programming (worst-case analysis), etc.
- Minimal support through the ThreadLocal API
- thread-local is unsafe, little programmer aid
Leak

HOWEVER…

class Foo {
    private ThreadLocal<Bar> bar = new ThreadLocal<Bar>();
    private Bar baz = null;

    synchronized void foo() {
        baz = bar.get();
    }
}

torsdag den 5 november 2009

HOW CAN WE MAKE THREAD-LOCALITY SAFE YET KEEP PROGRAMMING SIMPLE?

SEARCHING FOR A SIMPLE SOLUTION

• Idea:
  - Divide classes up in two categories: those that are used to create thread-local entities and those used to create shared entities
  - Let thread-local classes be used as shared too

• Problem:
  - Reuse and libraries

• Solution:
  - Two annotations: @shared, @thread-local
  - One simple rule: dataflow must preserve thread-locality
  - One magic class: the thread-local API

• Defaults: @shared is implicit for classes, all other annotations are implicitly the same as the current this

THE POWER OF SIMPLICITY
Foo creates "shared" instances

```
@shared
class Foo {
  @thread
  private Bar bar = null;
  @shared
  private Bar baz = null;
  void synchronized foo() {
    baz = bar;
  }
}
```

safe points to a thread-local object

e.g.

```
{ baz = bar; }
{ baz = bar; }
```

unsafe points to a "shared" object

Foo creates "shared" instances
One, crucial, difference from first example

```java
@shared
class Foo {
  @shared
  private Bar bar = null;
  @shared
  private Bar baz = null;

  void synchronized foo() {
    baz = bar; // will not compile
  }
}
```

unsafe points to a "shared" object

```
new ThreadLocal<Bar> bar = new ThreadLocal<Bar>();
```

```
private ThreadLocal<Bar> bar = new ThreadLocal<Bar>();
```

Foo creates "shared" instances

```
new ThreadLocal<Bar> bar = new ThreadLocal<Bar>();
```

```
private ThreadLocal<Bar> bar = new ThreadLocal<Bar>();
```

safe points to a thread-local object

```
baz = bar.get(); // will not compile
```

```
baz = bar.get(); // will not compile
```

```
baz = bar.get(); // will not compile
```

```
baz = bar.get(); // will not compile
```

```
```
Accessing hashmaps is about 8x the speed of accessing a field. Plus the overhead of creating and destroying hashmap aggregate objects. Unacceptable overhead if many fields are thread-local (likely to be a factor 50-100 size overhead). thread-local if the current thread is the current thread-local.

However...

```java
@thread
class Foo {
    private Bar bar = null;
    private Bar baz = null;
    void synchronized foo() {
        baz = bar; // OK
    }
}
```
class Foo {
    private Bar bar = null;
    private Bar baz = null;

    void synchronized foo() {
        baz = bar; // OK
    }
}

no expansion needed

synchronized still needed if Foo used to create "shared" object

LOGICAL MEMORY MODEL

{ }

volatile synchronized Foo() {
    private Bar bar = null;
    private Bar baz = null;
}
SIMPLE TYPE SYSTEM

Shared to thread unless guarded by thread-local fields

Intra-thread & intra-shared

Allowed

Intra-thread

Disallowed

Thread to shared

Inter-thread
tated classes are implicitly
y
n this would have had the drawback of requiring invasive changes to
@Shared
::= (]
::= []
@Thread
annotationp
Javan modulo genericsn checking of native coden and reflectionp The tool impleo
We have implemented
the initial stack frame and heap are emptyw The relation p
Javan modulo genericsn checking of native coden and reflectionp The tool impleo
v
H
'S or
x=
w
Lemma 1. Local matching point in shared heaplet of current humplet.

Meta Theory

Type System 2/2

VIEWPOINT ADAPTATION

Type System 1/2
Pragmatic: warnings rather than errors

In classes that are never used to create shared instance, the
compiler
removes method with all synchronization on thread-local variables

Every \( \text{@thread} \) class gets an extra method entry for each

SUPPORTING “ALL” OF JAVA

AVOIDING UNNECESSARY SYNCHRONIZATION

Extra methods are not needed (loc0 2.0)

(\un)expand \( \text{@thread} \) annotations

What I haven’t shown

(\un)covered in the paper

Extra methods are not needed (loc0 2.0)

In classes that are never used to create shared instance, the
compiler
removes method with all synchronization on thread-local variables

Every \( \text{@thread} \) class gets an extra method entry for each

(\un)expand \( \text{@thread} \) annotations

(\un)covered in the paper

Extra methods are not needed (loc0 2.0)

In classes that are never used to create shared instance, the
compiler
removes method with all synchronization on thread-local variables

Every \( \text{@thread} \) class gets an extra method entry for each

(\un)expand \( \text{@thread} \) annotations

(\un)covered in the paper

Extra methods are not needed (loc0 2.0)

In classes that are never used to create shared instance, the
compiler
removes method with all synchronization on thread-local variables

Every \( \text{@thread} \) class gets an extra method entry for each

(\un)expand \( \text{@thread} \) annotations

(\un)covered in the paper

Extra methods are not needed (loc0 2.0)

In classes that are never used to create shared instance, the
compiler
removes method with all synchronization on thread-local variables

Every \( \text{@thread} \) class gets an extra method entry for each

(\un)expand \( \text{@thread} \) annotations
Most objects are thread-local.

```
DEEP COPYING CODE

DEEP COPYING CODE
```

### Analysis and inference by Filip Pizlo

### Pragmatic Evaluation

- **Note:** Deep copying code can be type checked.
  - Copy-on-write
  - Deep copying (current solution)
  - (extramal) ungueness
- Open problem (with some known solutions)
We have also used our Jikes RVM instrumentation to check that all objects are shared because of sparse uses of native code in some collection implementations. For simplicity, we assumed that native code did not leak, classes of all collection standard API could be annotated. We have also used our Jikes RVM instrumentation to check that all objects are shared because of sparse uses of native code in some collection implementations. Native code always leaks, so the number is 77, and ergo all collection classes are marked.

The remaining classes were marked @Thread.

To further test our assumption that most classes can be annotated with thread-locality polymorphism, we implemented a conservative backward flow analysis to detect leakage of threads. Applying the analysis on the GNU Classpath version of the Java standard API, we measured both the average rate at which objects are thread-local and the average number of bytes that belong to thread-local objects in the entire DaCapo suite. The bottom shows results of dynamic analysis for DaCapo benchmarks.

Results from experiments with annotating Java programs with Class-Level Annotation Inference.

• Re-run experiments with improved tool (we need help!)
• Improved Eclipse tool support (we need help!)
• Improved eclipse tool support (we need help!)

<table>
<thead>
<tr>
<th>Locale</th>
<th>LOC</th>
<th>@Thread</th>
<th>@Shared</th>
<th>@Thread &amp; @Shared</th>
<th>@Shared</th>
<th>@Thread</th>
<th>@Thread &amp; @Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lucene Search</td>
<td>30</td>
<td>14</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Raytracer</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

BLOAT, ANTLR, and others, the rate is not uniform. As Table 2 shows, all benchmarks have at least half of their heaps occupied by thread-local objects—significantly, our results show that small objects tend to be more likely to be thread-local.

Conclusions:

- Experiments: annotated Java API is a must.
- Guinea and thread-locality polymorphism exist, Java classes work despite of (although as shared).
- Simple: very few rules and annotations, few special cases.
- Room for improvement: transfer across threads, local objects, most classes can be used to create threads.
- Compatible: most classes can be used to create threads.

Results will be much better with annotated Java API.

Added local annotations to ~44 KLOC Java.

WHAT'S AHEAD