Preexistence revisited

R. Ducournau, J. Pagès, J. Privat, C. Vidal

LIRMM – Université de Montpellier & CNRS
Université du Québec à Montréal

ICOOOLPS, Prague, July 2015
Plan

1. Motivations
2. Preexistence
3. Experiments with preexistence
4. Future work and conclusions
Motivations: Virtual Machines

Work under the Open-World Assumption
- dynamic class loading
- lazy method compilation

Performance results from greedy Current-World optimizations
- devirtualization
- inlining

Consequence
- need for dynamic recompilations
Motivations: the Repair Problem

How to recompile a method ...

... while it is running
Motivations: the Repair Problem

3 well-known techniques

- guards
- stack-patching (aka OSR)
- code-patching

Still better to avoid them
**Motivations: Multiple Inheritance**

**In a Java-like language**

Optimisations apply to
- method invocation (mainly)
- subtyping tests (marginally)
- and interfaces

**In multiple inheritance**

Optimisations apply to
- attribute access, too
Motivations: Multiple Inheritance

**Inlining mechanisms**
- attribute access
- subtyping tests

**Repair techniques**
- stack-patching: does not apply at all
- code-patching: does not apply efficiently
## Motivations: Multiple Inheritance

### Object representation

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Method</th>
<th>Attribute</th>
<th>Subtyping</th>
</tr>
</thead>
<tbody>
<tr>
<td>inlining</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>static</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>single subtyping (SST)</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>perfect hashing (PH)</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>unknown</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

### Optimizations involve substituting

- inlining to static (methods only)
- static to SST (except attributes)
- SST to PH
Plan

1. Motivations
2. Preexistence
3. Experiments with preexistence
4. Future work and conclusions
A property ensuring that
- a reference will remain compatible
- with the current compiled code of a method
- during the current activation of this method

Preexistence of the receiver avoids the need for hot repair
Original preexistence of value

The referenced object has been created before entering the method.

2 original rules [Detlefs and Agesen, Ecoop’99]
- an input parameter is preexisting,
- an immutable attribute of a preexisting object is preexisting

Example

```python
def bar(x) { x.foo()}
```

x is preexisting and the call to `foo` can be safely optimized.

Assessment

Between 20% and 60% of call sites have a preexisting receiver.
Extended preexistence of type

The object’s class has been loaded before entering the method

The main rule

- new A() is preexisting iff A is already loaded

Example

```python
def bar(x) {
    if condition then y=x else y=new A() end
    y.foo()
}
```

y is preexisting and the call to foo can be safely optimized
Extended preexistence of type

Auxiliary rules

- any expression typed with a `final` type is preexisting
- a `method-invocation` expression is preexisting iff
  - each invoked method has a preexisting return
  - each argument corresponding to a returned parameter is preexisting

Consequence

- a call to a `factory` method is preexisting
- provided that all the invocable methods are compiled!
Extended preexistence

Pros

- extended ⇒ (hopefully many) more preexisting receivers
- applied to attribute access and subtyping tests, too

Cons

- preexistence is no longer immutable
- a preexisting method-invocation becomes non-preexisting when a class redefining this method is loaded
- a method must be recompiled when a site of it switches to non-preexistence
Plan

1. Motivations
2. Preexistence
3. Experiments with preexistence
4. Future work and conclusions
Experiments with preexistence

The testbed

- the **Nit** language (Jean Privat, UQAM, formerly **Prm**, LIRMM)
- the **Nit** Closed-World interpreter
- the **Nit** Open-World VM, based on the interpreter
- a meta-evaluator benchmark
  - the Nit interpreter
  - run in the Nit VM
  - on a small Nit program (eg **fibonacci(4))**
- statistics at the end of the computation
### Statistics of preexistence

<table>
<thead>
<tr>
<th></th>
<th>method</th>
<th>attribute</th>
<th>subtyping</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>preexisting</td>
<td>4044</td>
<td>3802</td>
<td>248</td>
<td>8094</td>
<td>58%</td>
</tr>
<tr>
<td>non-preexisting</td>
<td>4216</td>
<td>916</td>
<td>734</td>
<td>5866</td>
<td>42%</td>
</tr>
<tr>
<td>total</td>
<td>8260</td>
<td>4718</td>
<td>982</td>
<td>13960</td>
<td></td>
</tr>
</tbody>
</table>

- For methods, preexistence rate is in the middle-upper range of the original paper.
- Even higher for attributes (80%)
- There is potential for improvement.
# Statistics of preexistence

## Original non-preexistence

<table>
<thead>
<tr>
<th></th>
<th>method</th>
<th>attribute</th>
<th>subtyping</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>potential</td>
<td>4216</td>
<td>916</td>
<td>734</td>
<td>5866</td>
<td>100%</td>
</tr>
<tr>
<td>NewSite</td>
<td>1331</td>
<td>80</td>
<td>0</td>
<td>1411</td>
<td>24%</td>
</tr>
<tr>
<td>CallSite</td>
<td>1388</td>
<td>255</td>
<td>663</td>
<td>2306</td>
<td>39%</td>
</tr>
<tr>
<td>ReadSite</td>
<td>1426</td>
<td>551</td>
<td>68</td>
<td>2045</td>
<td>35%</td>
</tr>
</tbody>
</table>

---

Experiments with preexistence

(.LIRMM+UQAM) Preexistence revisited

ICOOOLPS'15
# Statistics of preexistence

## Original non-preexistence

<table>
<thead>
<tr>
<th></th>
<th>method</th>
<th>attribute</th>
<th>subtyping</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>potential</td>
<td>4216</td>
<td>916</td>
<td>734</td>
<td>5866</td>
<td>100%</td>
</tr>
<tr>
<td>NewSite</td>
<td>1331</td>
<td>80</td>
<td>0</td>
<td>1411</td>
<td>24%</td>
</tr>
<tr>
<td>CallSite</td>
<td>1388</td>
<td>255</td>
<td>663</td>
<td>2306</td>
<td>39%</td>
</tr>
<tr>
<td>ReadSite</td>
<td>1426</td>
<td>551</td>
<td>68</td>
<td>2045</td>
<td>35%</td>
</tr>
<tr>
<td>improvable</td>
<td>2719</td>
<td>335</td>
<td>663</td>
<td>3717</td>
<td>63%</td>
</tr>
</tbody>
</table>

(LIRMM+UQAM)
## Statistics of preexistence

### Extended preexistence

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>improvable</td>
<td>3717</td>
<td>100%</td>
</tr>
<tr>
<td>NewSite</td>
<td>1390</td>
<td></td>
</tr>
<tr>
<td>CallSite</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>improved</td>
<td>1398</td>
<td>38%</td>
</tr>
</tbody>
</table>
Experiments with preexistence

Pros and cons

- most of the improved sites have static concrete types
- inter-procedural analysis has marginal effect
Experiments with preexistence

**Pros and cons**

- most of the improved sites have **static concrete types**
- inter-procedural analysis has marginal effect
Plan

1. Motivations
2. Preexistence
3. Experiments with preexistence
4. Future work and conclusions
Experiments with inlining

But preexistence rate is meaningless
- preexistence depends on programming style
- any program can be transformed into a 100%-preexistence program
- preexistence is not preserved by inlining

\[
\text{def } \text{bar}(y) \{ \ y.\text{baz}() \ \} \\
\text{def } \text{bar}(x.\text{foo}()) \ \iff \ x.\text{foo}().\text{baz}()
\]

Next step involves experimenting inlining
- inline \text{bar} or \text{baz}, not both
Other perspectives

Assessing the recompilation cost
- method recompilations
- transitions between implementations
- transitions between preexistence and non-preexistence

Extended protocols
- with guards or patches?

Other benchmarks
Conclusion

Extended preexistence: an interesting idea
which needs a deeper study