Empirical Assessment of C++-like Implementations for Multiple Inheritance

Floréal Morandat ∗ Roland Ducournau

LIRMM — The University of Montpellier & CNRS

MASPEGHI/ICOOOLPS’10, June 21–15, 2010, Maribor — Slovenia
Empirical Assessment of C++-like Implementations for Multiple Inheritance

Outline

1. Introduction
2. Implementation Techniques
3. Execution Testbed
4. Results and Discussion
5. Conclusion and Prospects
Introduction

Context & Motivation
Compilation of multiple inheritance with static typing:
- Constant time
- Reasonable space
- Compatible with dynamic loading (OWA)

Objective
Only two implementation techniques satisfy these requirements:
- C++ Subobjects (virtual inheritance)
- Perfect Class Hashing

Which is the best one?
Outline

1. Introduction

2. Implementation Techniques
   - Single Subtyping
   - C++ Subobjects
   - Perfect Class Hashing
   - Abstract assessment

3. Execution Testbed

4. Results and Discussion

5. Conclusion and Prospects
Attribute Access

load \([\text{object} + \#\text{attributeOffset}] \rightarrow \text{value}\); \textit{Read}

store \text{value} \rightarrow [\text{object} + \#\text{attributeOffset}]; \textit{Write}
Attribute Access

load [object + #attributeOffset] -> value ; Read

store value -> [object + #attributeOffset] ; Write
Single Subtyping (SST)

Method invocation

load [object + #tableOffset] -> table
load [table + #methodOffset] -> method
call method
Single Subtyping (SST)

Subtype test

load [object + #tableOffset], table
load [table + #classIdPosition], class
cmp #classId, class
bne #checkFailed
Advantages

- Compatible with OWA
- Optimal time/space for dynamic loading
  - Time: Constant for all mechanism
  - Space: Linear in the specialization relationship size
    - i.e. quadratic in the class number
- Simple at compile-time

Invariants

- References don’t depend on their static type
- Positions independent of receiver’s dynamic type
## From SST to MI

MI can’t preserve both OWA and SST invariants

### Preserving SST invariants

Coloring

- Requires CWA at link-time

[Dixon et al. 1989]

### Preserving OWA and position invariant

C++ Subobjects (virtual inheritance)

- References depend on their static types
- Overhead: **Cubic** table size, pointer adjustments, ...

### Preserving OWA and reference invariant

Perfect Class Hashing

- Positions depend on the dynamic type
- Overhead: Hashing + Hashtable
Method invocation

load [object + #tableOffset], table
load [table + #methOffset + fieldLen], delta2
load [table + #methOffset], method
add object, delta2, object2
call method
Method invocation

load [object + #tableOffset], table
load [table + #methOffset + fieldLen], delta2
load [table + #methOffset], method
add object, delta2, object2
call method
Method invocation

load [object + #tableOffset], table
load [table + #methOffset + fieldLen], delta2
load [table + #methOffset], method
add object, delta2, object2
call method
Method invocation

load [object + #tableOffset], table
load [table + #methOffset + fieldLen], delta2
load [table + #methOffset], method
add object, delta2, object2
call method
**Attribute access**

load [object + #tableOffset], table
load [table + #castOffset], delta1
add object, delta1, object1
load [object1 + #attrOffset], value
**Attribute access**

load [object + #tableOffset], table
load [table + #castOffset], delta1
add object, delta1, object1
load [object1 + #attrOffset], value
### Attribute access

- `load [object + #tableOffset], table`
- `load [table + #castOffset], delta1`
- `add object, delta1, object1`
- `load [object1 + #attrOffset], value`
C++ Subobjects (SO)

Attribute access

load [object + #tableOffset], table
load [table + #castOffset], delta1
add object, delta1, object1
load [object1 + #attrOffset], value
Attribute access

load [object + #tableOffset], table
load [table + #castOffset], delta1
add object, delta1, object1
load [object1 + #attrOffset], value
**C++ Subobjects (SO)**

**Pointer adjustment**

- `load [object + #tableOffset], table`
- `load [table + #castOffset], delta1`
- `add object, delta1, object1`
- `load [object1 + #attrOffset], value`
Perfect Class Hashing (PH)

Preamble

load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable
Perfect Class Hashing (PH)

Preamble

load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable
Perfect Class Hashing (PH)

Preamble

load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable
Subtype testing

```
load [htable + #htOffset-fieldLen], id
comp #classId, id
bne #fail
% succeed
```
Perfect Class Hashing (PH)

Subtype testing

load [htable + #htOffset-fieldLen], id
comp #classId, id
bne #fail
% succeed
Perfect Class Hashing (PH)

Subtype testing

load [htable + #htOffset-fieldLen], id
comp #classId, id
bne #fail
% succeed
Perfect Class Hashing (PH)

Method invocation

load [htable + #htOffset], cOffset
load [ctable + #methOffset], method
call method
Perfect Class Hashing (PH)

Method invocation

load [htable + #htOffset], cOffset
load [ctable + #methOffset], method
call method

Empirical Assessment of C++-like Implementations for Multiple Inheritance
Method invocation

load [htable + #htOffset], cOffset
load [ctable + #methOffset], method
call method
Perfect Class Hashing (PH)

Method invocation

load [htable + #htOffset], cOffset
load [ctable + #methOffset], method
call method
Attribute access

load [htable + #htOffset], aOffset
add objet, aOffset, aGroup
load [aGroup + #attributeOffset], value
**Attribute access**

load [htable + #htOffset], aOffset
add objet, aOffset, aGroup
load [aGroup + #attributeOffset], value
Perfect Class Hashing (PH)

Attribute access

load [htable + #htOffset], aOffset
add objet, aOffset, aGroup
load [aGroup + #attributeOffset], value
Attribute access

load [htable + #htOffset], aOffset
add objet, aOffset, aGroup
load [aGroup + #attributeOffset], value
### Abstract assessment

#### Subobjects (SO)
- Pointer adjustments
- Cubic space
- Overhead in object layout

#### Perfect Class Hashing (PH)
- Overhead for hashing
- Quadratic space

### Abstract cycle count

<table>
<thead>
<tr>
<th></th>
<th>Subtype test</th>
<th>Method call</th>
<th>Attr access</th>
<th>Ptr adjust</th>
</tr>
</thead>
<tbody>
<tr>
<td>SST</td>
<td>2L + 2</td>
<td>2L + B</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>SO</td>
<td>4L + 4</td>
<td>2L + B + 2</td>
<td>L</td>
<td>2L + 1</td>
</tr>
<tr>
<td>PH</td>
<td>3L + 4</td>
<td>L + B + 3</td>
<td>4L + 3</td>
<td></td>
</tr>
</tbody>
</table>
Empirical Assessment of C++-like Implementations for Multiple Inheritance

1. Introduction
2. Implementation Techniques
3. Execution Testbed
4. Results and Discussion
5. Conclusion and Prospects
The Prm Language

- Full multiple inheritance (methods & attributes)
- Genericity
- Primitive types are subtypes of Object

The PrmC Compiler

- A Prm program
- Modular
- Generates C code
Meta-Compiling Test Protocol

- All boxes are compilers
- Time measurement of the red path
Meta-Compiling Test Protocol

- All boxes are compilers
- Time measurement of the red path
**Runtime Reproducibility**

- Deterministic code generation
  - Hashmap with predictable iteration order
  - Produces diff-equivalent binaries
- Bootstrap = actual fix point

**Measurements**

- Time spent by the **Prm to C** process
- Best time among severals tens of runs
Empirical Assessment of C++-like Implementations for Multiple Inheritance
Runtime Efficiency

Overhead wrt SST

Core2 E8500 3.16Gz

Xeon Irwindale 2.8Gz

Static space

Dynamic space

Perfect Hashing
Subobject
Without GC

Introduction Implementation Techniques Testbed Results Conclusion
Discussion

Subobjects
- Dynamic space overhead ⇒ Increase GC time
- Numerous pointer adjustments ⇒ Offset PH overhead

Perfect Class Hashing
- Time: Similar to SO
- Space:
  - Slight static overhead vs SST (Code length + Hashtable size)
  - No dynamic overhead
- Allow for Just-In-Time optimizations
Outline

1. Introduction
2. Implementation Techniques
3. Execution Testbed
4. Results and Discussion
5. Conclusion and Prospects
Conclusion

Comparisons
- First “language-independent” comparisons of SO implementation
- First comparison of PH vs OWA compatible implementation

Perfect Class Hashing
According to these tests:
- Time Efficiency similar to Subobjects
- Allow for Just-In-Time optimizations

The Prm Testbed
- Modular compiler open to new implementations
- Repeatable and reproducible tests
- Single benchmark, but intensive OO mechanism usage
Prospects

**Testbed extension**

- Improving Subobjects
  - Empty Subobjects optimization
  - Thunk implementation
  - Heterogeneous vs homogeneous genericity
- Other processors and architectures
- Other optimisations (garbage collector, ...)

**Virtual Machine application**

- Perfect Class Hashing on Production VM
- Full multiple inheritance VM (as efficient as Java/.NET)
Thanks ...

**Prm the languages**
- **Prm**: Dedicated to test  
  http://www.lirmm.fr/prm/
- **Nit**: User friendly language (recommended)  
  http://www.nitlanguage.org/