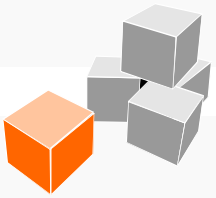
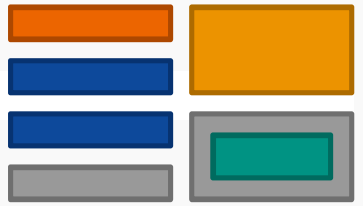
Twente Research and Education on Software Engineering, Universiteit Twente



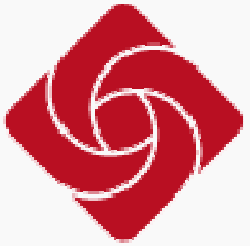
Software Technology Group
TU Darmstadt | FB Informatik



www.alia4j.org

Optimizing the Evaluation of Patterns in Pointcuts

Remko Bijker, r.bijker@student.utwente.nl
Christoph Bockisch, c.m.bockisch@cs.utwente.nl
Andreas Sewe, sewe@st.informatik.tu-darmstadt.de



CASED

Aspect-Oriented Programming

- Localizing implementation of behavior common to multiple modules
- Common core concepts
  - Definition of functionality (**advice** in AspectJ)
  - Definition when functionality is applicable (**pointcut** in AspectJ)
- Pointcuts **evaluate** to set of **join points**
  - Generally regions in time (e.g., beginning of method invocation till returning from method)
- Join points described in terms of
  - **Static properties**
  - **Dynamic properties**

Aspect-Oriented Execution Environments

- Aspect deployment
  - Aspect must become active
  - **Partially evaluate pointcuts** to **join-point shadows** (code locations)
  - Insert instructions
    - Evaluating dynamic properties
    - Executing advice
- Support for deployment at
  - Compile-time
  - Class-loading-time
  - Run-time

Partial pointcut evaluation affects overall application performance.

Partial Pointcut Evaluation

- Static part of pointcut includes
  - Kind of join point (e.g., method call, field write)
  - **Pattern** over **signature** of associated member
- (Naïve) partial evaluation
  - **Iterate** over list of potential join-point shadows
  - Match pattern against each
- Hypotheses
  - Naïve partial evaluation is **slow**
  - Patterns only match few join-point shadows
  - Data (signatures and patterns) is **structured**

Optimization Strategies for Pattern Evaluation

- Observation
  - **Signatures** and **patterns** made up of **parts**
  - Match when all sub-patterns match

Method	Declaring Class	Modifiers	Result Type	Name	Parameter Types	Exceptions
Constructor	Declaring Class	Modifiers			Parameter Types	Exceptions
Static Initializer	Declaring Class					
Field (Read/Write)	Declaring Class	Modifiers	Type	Name		

- List of join-point shadows as **database table**
  - Signature parts as columns, patterns as queries
  - **Index** on signature part speeds up sub-pattern
  - **Search-plan optimization** orders sub-pattern evaluation

Research Questions

- Which evaluation order is (heuristically) most efficient?
  - What is the selectivity of each sub-pattern?
- Which is the best data structure for an index?
- Is the additional effort of maintaining an index paying off?

Experimental Setup

- Survey **real-world programs**
  - OO programs for understanding signatures
    - ANTLR, FreeCol, LIAM, TightVNC
    - Total: 2432 classes, 28065 signatures, 150432 join-point shadows
  - AO programs for understanding patterns
    - ajlib-incubator, Contract4J5, Glassbox, Nversion, Sable Benchmarks
    - Total: 242 aspects, 170 different patterns
  - Pattern evaluation against Java tools and Java Runtime library
- Based on ALIA4J language-implementation approach (shown to support AspectJ, Compose*, ConSpec, etc.)
  - Used parts of ALIA4J to extract signatures
  - **Prototyped optimizations** in ALIA4J execution environment
  - Measured evaluation times

Survey Results (Method Patterns)

- Most patterns match methods (149/170)
- Usage of sub-pattern kinds

Sub-patter	Any	Wildcard pattern	Exact pattern
Declaring class	24%	1%	75%
Name	12%	16%	72%
Parameter Types	60%	4%	36%
Return type	82%	1%	17%
Modifiers	91%	—	9%
Exceptions	96%	2%	2%

Survey Results (Method Patterns)

- Determine **selectivity** of sub-patterns
  - Evaluate all patterns against
  - All signatures in surveyed OO applications
- Optimal **evaluation order**

Declaring class	Name	Return type	Modifiers	Parameters	Exceptions

Application of Survey Results

- ALIA4J implements pattern evaluation algorithm
- Modified to consider selectivity
- Compared evaluation time to original algorithm

TighVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
107%	108%	105%	99%	119%	112%

- Must weight selectivity with evaluation performance!

Name	Declaring Class	Modifiers	Return type	Exceptions	Parameters

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
29%	38%	28%	27%	33%	36%

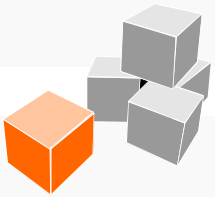
Additionally Using Index

- Pattern evaluation less than **10% of original algorithm**
- Index must be maintained
- Additional **effort pays off** with **fifth pattern evaluation**

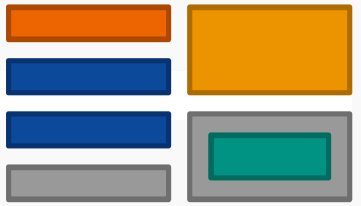
Conclusions & Future Work

- Optimizing pattern evaluation has potential
  - Pattern evaluation is **inherent to AO** execution environments
  - Performance significant in **dynamic AOP**
  - **Survey** shows opportunities
  - **Indexes** and **search-plan optimization** improve performance
- **Extend survey** to confirm results
- Confirm performance gain in **other execution environments**
- Make search-plan optimization **context-aware** (e.g., return type pattern together with “get*” name pattern is very selective)

Twente Research and Education on Software Engineering, Universiteit Twente



Software Technology Group
TU Darmstadt | FB Informatik



www.alia4j.org

http://www.alia4j.org

CASED