abc: an Implementation of AspectJ

Oege de Moor

Programming Tools Group

University of Oxford

joint work with

Ganesh Sittampalam, Sascha Kuzins, Chris Allan,
Pavel Avgustinov, Julian Tibble, Damien Sereni (Oxford)
Laurie Hendren, Jennifer Lhoták, Ondrej Lhoták, Bruno Dufour,
Christopher Goard, Clark Verbrugge (McGill)
Aske Simon Christensen (Århus)

What is AspectJ?

disciplined metaprogramming

The bluffer's guide to aspect-lingo

Static:

Intertype declarations:
inject new members into
existing classes at compile-time

Dynamic:

aspect observes base program
when certain patterns of events happen,
run some extra code

“join point” = event = node in (dynamic) call graph

“pointcut” = pattern of events = set of nodes in call graph

“shadow” = program point that corresponds to join point

“advice” = extra code

EJB policy enforcement

```
public aspect DetectEJBViolations {

    pointcut uiCalls() : call(* java.awt.*+.*(..));

    before() : uiCalls() && cflow(call(* EnterpriseBean+.*(..))) {
      System.err.println("UI call from EJB");
    }

}
```

Memoisation

```
abstract aspect Tabling {
    Hashtable table;
    abstract pointcut toMemo();

    before() : toMemo() && !cflowbelow(toMemo()) {
            table = new Hashtable();
    }

    Object around(Object n) : toMemo() && args(n) {
            Object entry = table.get(n);
            if (entry == null) {
                  entry = proceed(n);
                  table.put(n, entry);
            }
            return entry;
    }
}
```

Larger example: Ant Tournaments

Original task: ICFP 2005 programming contest
(won with Haskell by team from progtools group at Oxford)

Two ant hills do combat:
hill with most food wins

Practical assignment for 3rd year / MSc course:
construct pure Java simulator
add aspects for:
- tracing
- checking invariants
- viewer

No Allocations in Inner Loop

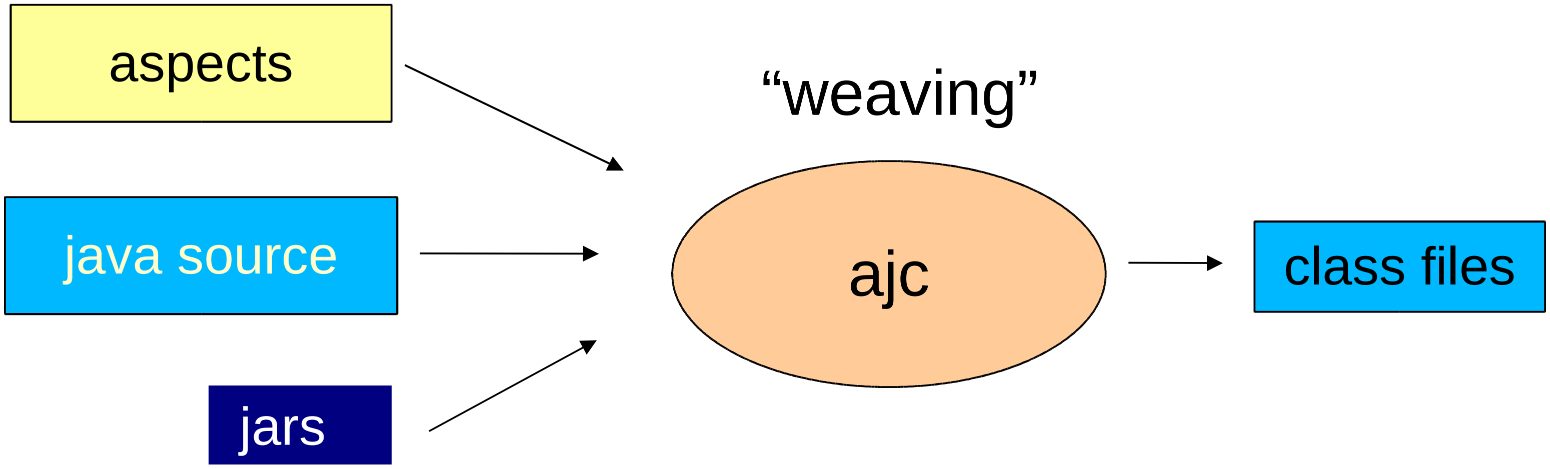
```
aspect NoNewInRound {
    private int allocations;

    before() : call(* World.play(..)) {
        allocations = 0;
    }

    before() : cflow(call(* World.play(..))) && call(*.new(..)) &&
           !call(java.lang.StringBuffer.new(..)) {
        System.err.println("allocation during play: "+
                            thisJoinPoint.getSourceLocation());
        allocations++;
    }

    after() : call(* World.play(..)) {
        if (allocations > 0)
            System.err.println("allocations per game "+allocations);
    }
}
```

Aspects in Ants Tournaments

- command:
  - Introduce comments
- debugging:
  - Check Scores
  - Command Tracer
  - Live Ants
  - World Dumper
- model:
  - Combat rules
  - Resting rules
- profile:
  - No allocations in inner loop
- style:
  - Use getters and setters
- viewer:
  - Updating of hexagons

can all be included or excluded at will

ajc: “standard” AspectJ compiler



- builds on Eclipse compiler
- weaving with BCEL
- aims to be fast
- about 45KLOC, excluding IDE support

- initially developed at Xerox Parc
- now part of Eclipse
- development mostly at IBM

Daniel Sabbah (VP of development@ IBM): “critical to our survival”

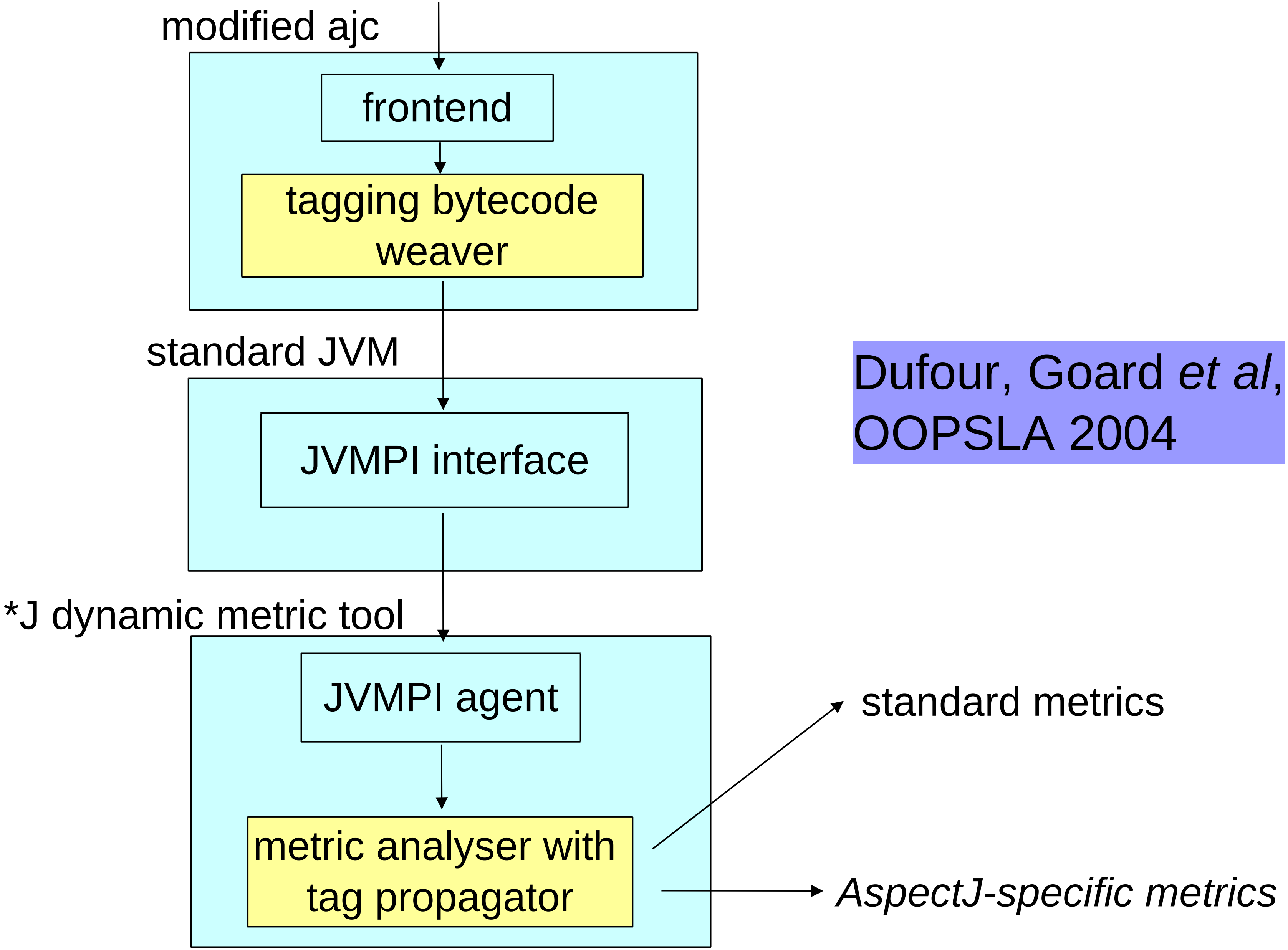
What do you pay at runtime?

From the FAQ on aspectj.org:

We aim for the performance of our implementation of AspectJ to be on par with the same functionality hand-coded in Java. Anything significantly less should be considered a bug.

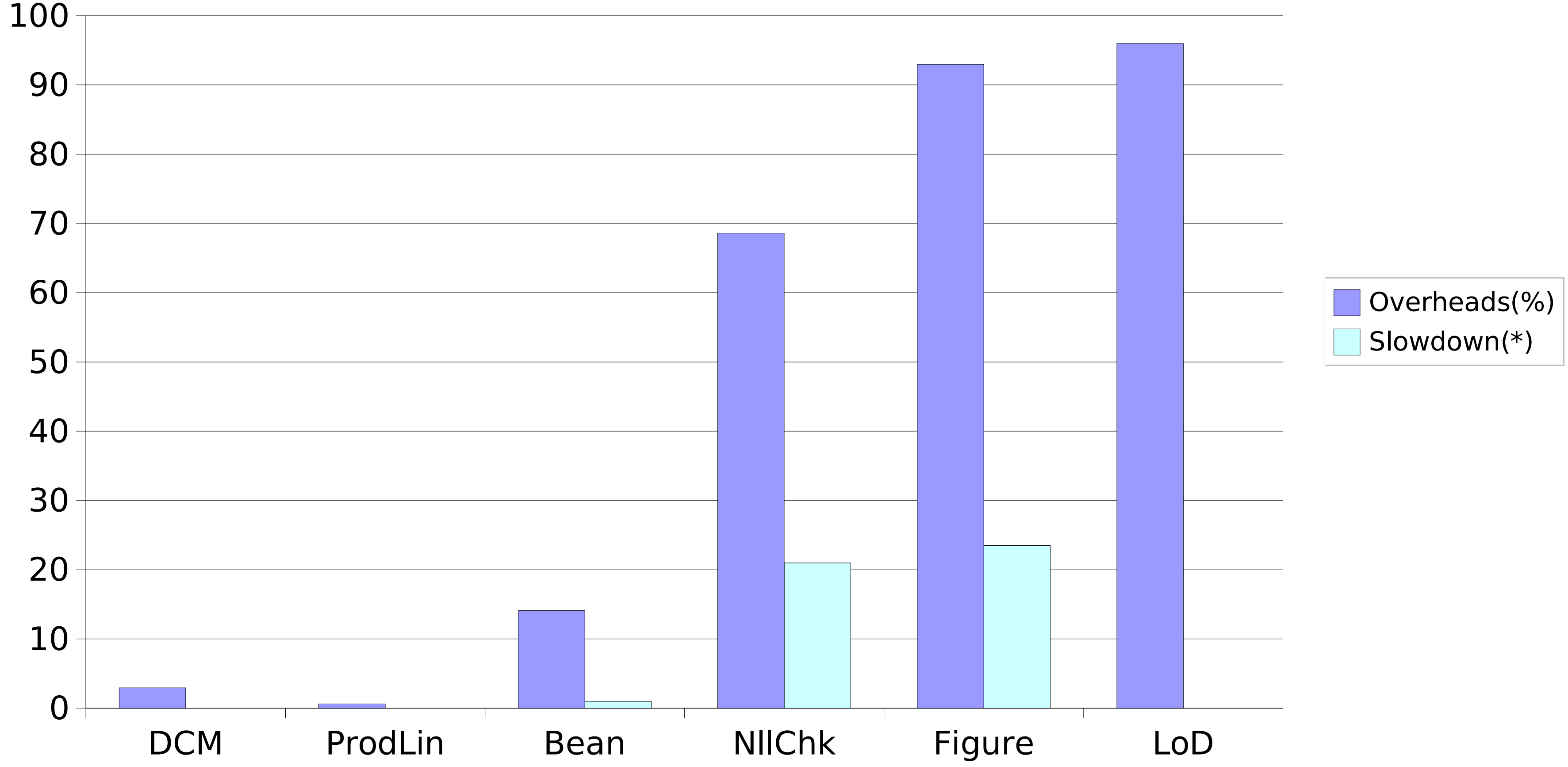
...we believe that code generated by AspectJ has negligible performance overhead.

Measuring the cost with *J



Dufour, Goard *et al*, OOPSLA 2004

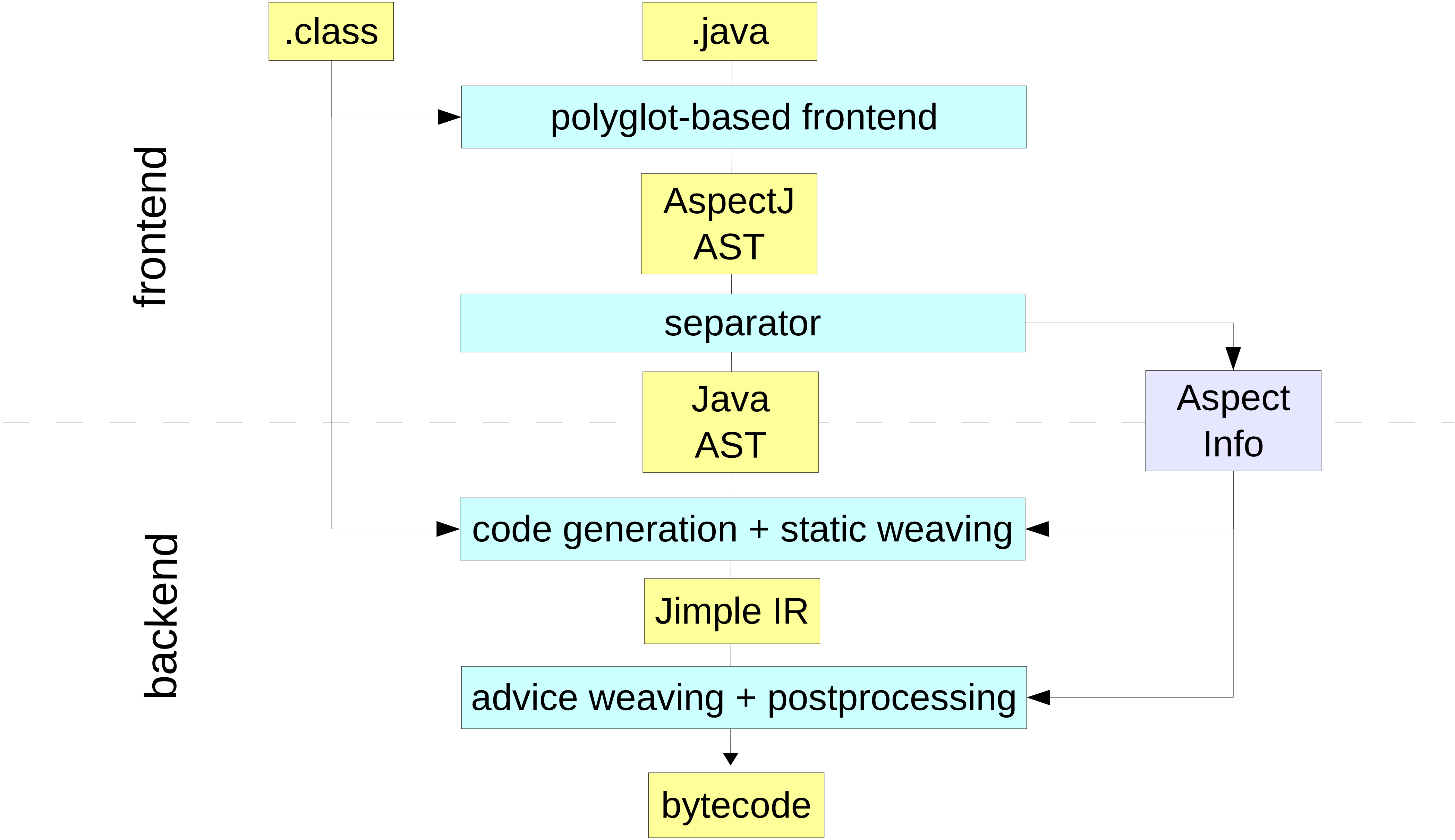
ajc 1.2 performance



The need for a second compiler

- language definition other than test suite
- explore AOP language design space
- experiment with better code generation
- experiment with static analyses

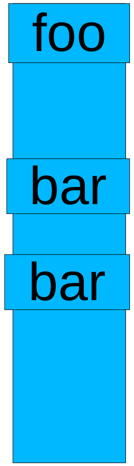
Architecture of abc



Focus on optimising *cflow*

```
pointcut fooFromBar(int x) :
      call(* foo()) &&
      cflow( call(* bar(*)) && args(x) )
```

call stack:



bind x to the argument of the last call to bar

Obvious implementation:
maintain stack of bindings

- push before each call to bar
- pop after each call to bar
- check top upon each call to foo

Intraprocedural optimisations

no variable binders?

use an integer counter instead of stack

share stacks for multiple pointcuts:

e.g. unify cflows in

```
call(* bar(..)) && cflow (call(* foo(..)) && args(t,*,*))
call(* bar(..)) && cflow (call(* foo(..)) && args(*,s,*))
```

to

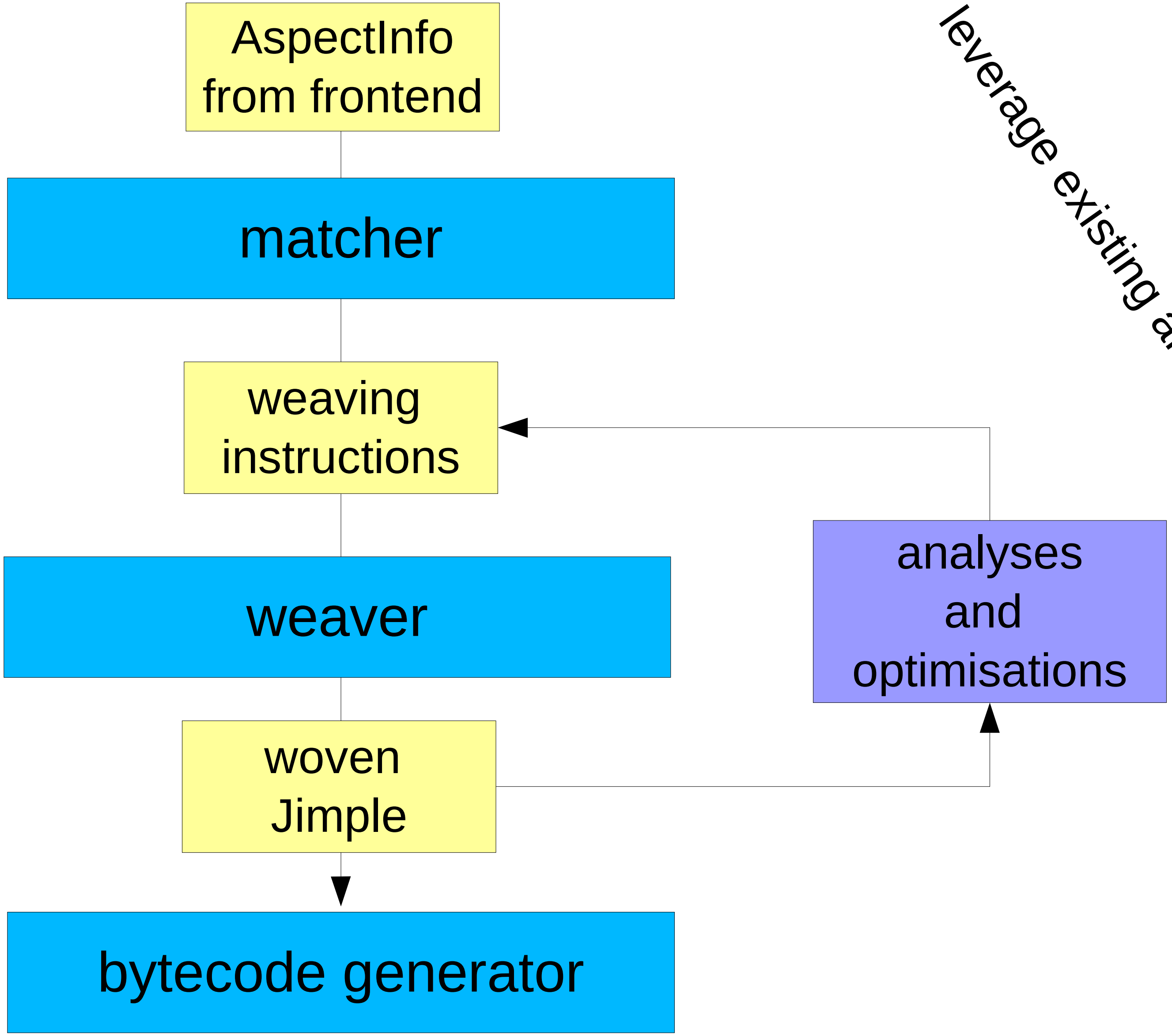
```
cflow(call(* foo(..)) && args(x,y,*))
```

each cflow stack is local to a thread

perform CSE on stack retrieval within method

reduce overheads of cflow, but do not eliminate them

Analysis in abc


AspectInfo from frontend

matcher

weaving instructions

weaver

woven Jimple

bytecode generator

analyses and optimisations

leverage existing analyses for pure Java

Desired cflow optimisations

to implement cflow(p)

update shadow:

push/pop stack at each shadow matching p

query shadow:

test whether stack nonempty

at query shadow:

predict emptiness:

if yes or no, remove test

at update shadow:

predict whether observed by any query:

if not, remove push/pop

Analysis information required

For each update shadow sh:

st ∈ *mayCflow*(sh) :

at statement st, we *may* be in the dynamic scope of sh

st ∈ *mustCflow*(sh) :

at statement st, we *must* be in the dynamic scope of sh

sh ∈ *necessaryShadows*:

∃ qsh ∈ mayCflow(sh) (it's queried)

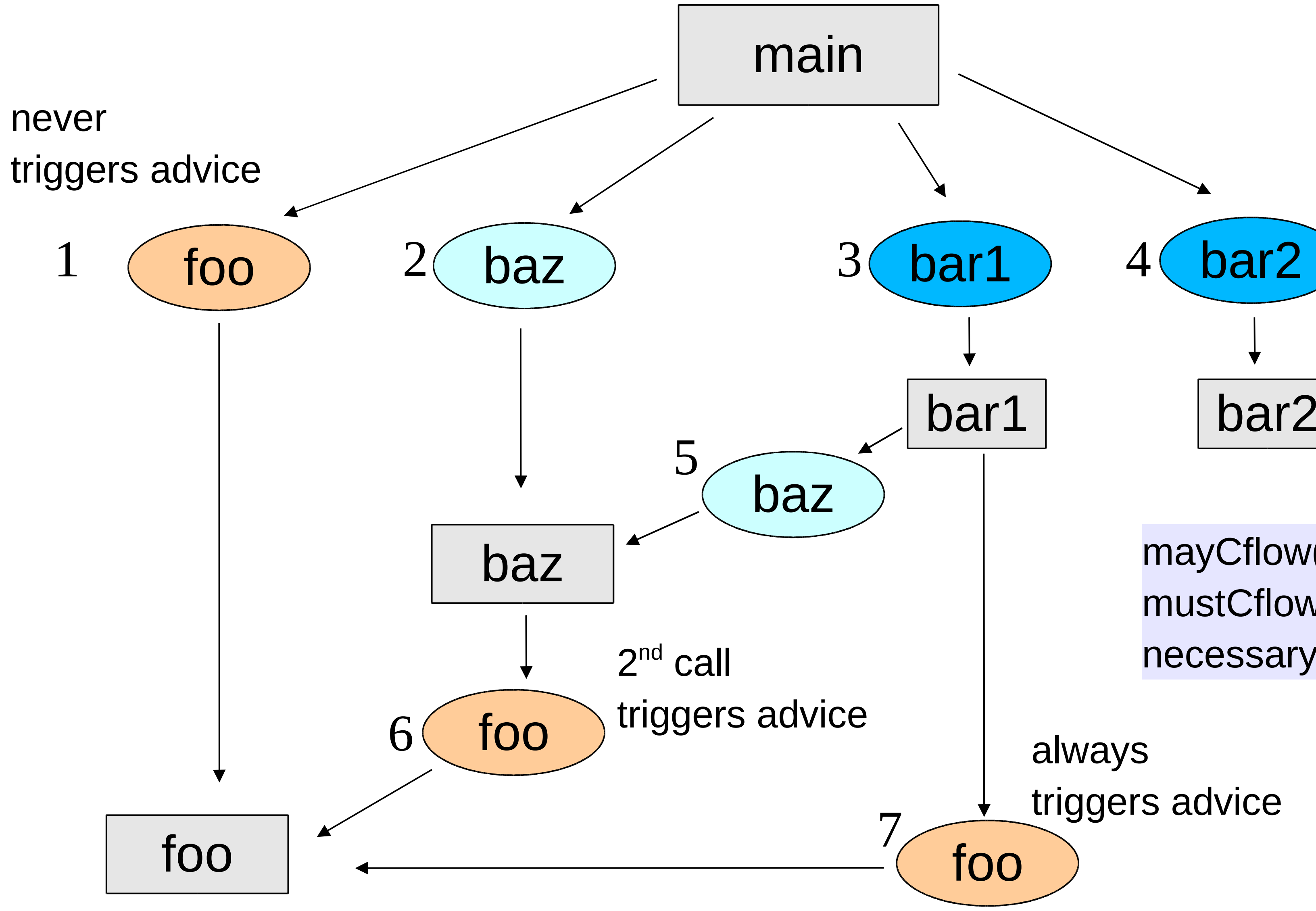
∧ ¬ (∃ sh' : sh ∈ mustCflow(sh')) (otherwise it's guaranteed to be nonempty)

Example

```
aspect Aspect {
    pointcut fooFromBar(int x) :
        call(* foo()) &&
        cflow( call(* bar*(*)) && args(x) );

    before(int x) : fooFromBar(x) {
        System.out.println("foo from bar, x="+x);
    }
}
```

```
public class Cflow {

    void foo() {}
    void bar1(int x) { foo(); baz(); }
    void bar2(int x) {}
    void baz() { foo(); }

    public static void main(String[] args) {
        Cflow c = new Cflow();
        c.foo();
        c.baz();
        c.bar1(3);
        c.bar2(4);
    }

}
```

Call Graph



mayCflow(3) = {5,6,7}
mustCflow(3) = {7}
necessaryShadows = {3}

Computing Analysis Information

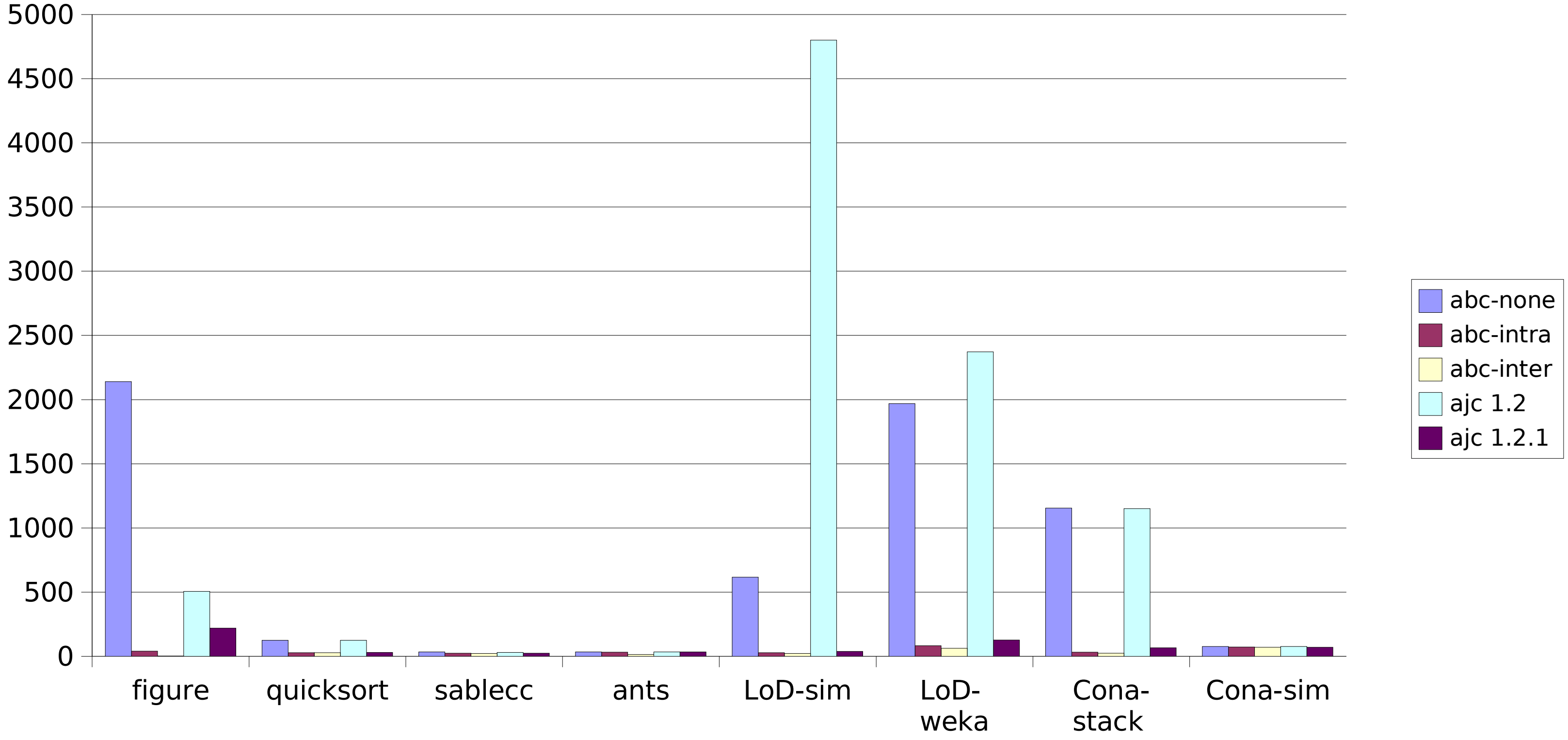
computation of mayCflow(sh):

```
mayCflow $\leftarrow$ { st | st is in intraprocedural shadow of sh}
repeat
    for all methods m | $\exists$ st $\in$ mayCflow : st may call m do
        mayCflow $\leftarrow$ mayCflow $\cup$ set of statements in m
until mayCflow does not change
```

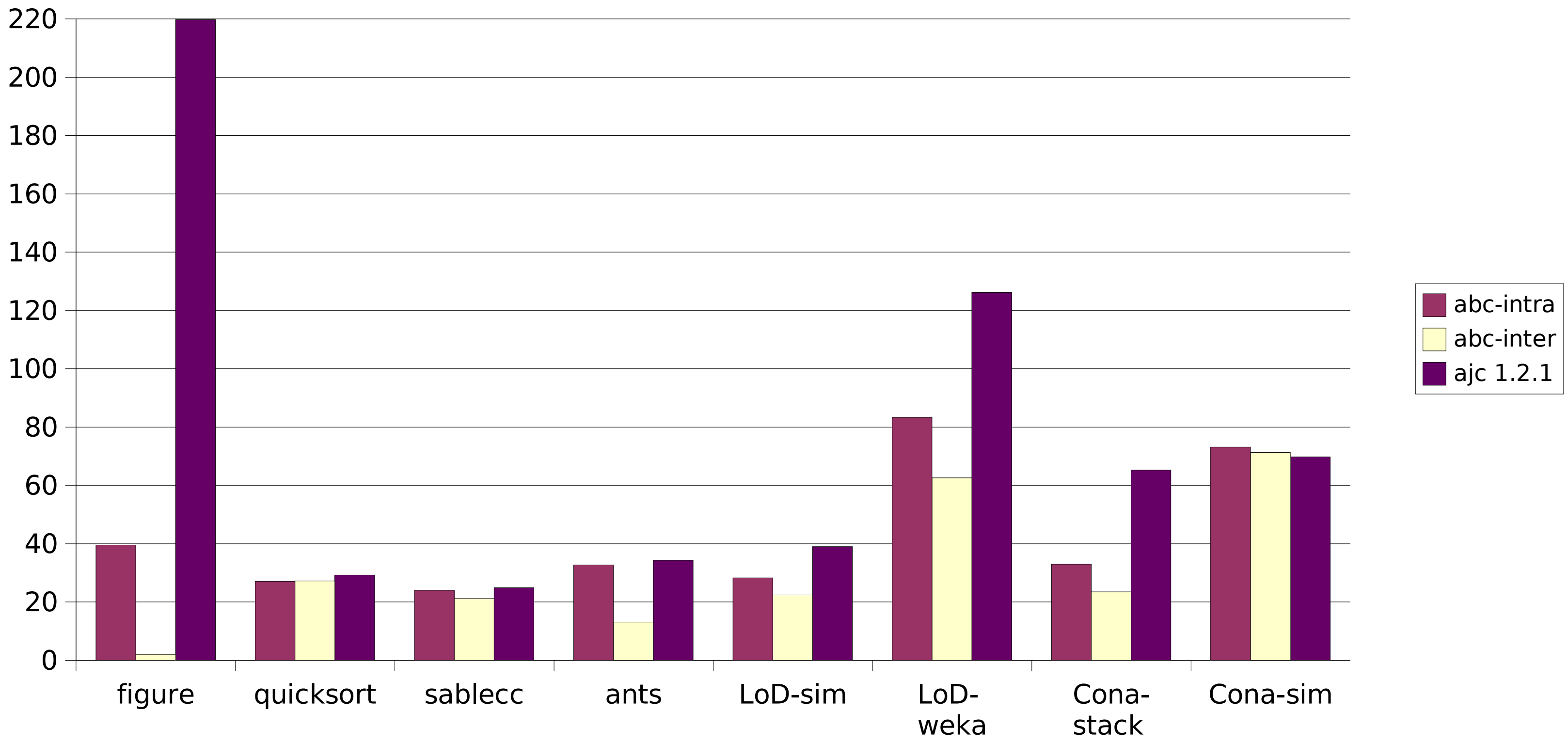
“may call” : use Paddle framework for callgraph construction

set representation: BDDs via Jedd
(extension of Java for programming BDD-based analyses)

abc *cflow* performance (1)



abc *cflow* performance (2)



Research Directions

- aspects are here to stay
- what might the next language look like?
- what are the main implementation challenges?

Where will AspectJ go?

LANGUAGE:

open classes:
- relaxed MultiJava, nested inheritance

pointcuts:
- match on semantic properties
- observation of traces
  - via regular patterns
- logic query language
- hiding events

static property checking

IMPLEMENTATION:

reduce weave time:
- matching automaton for set of pointcuts

incremental
- compilation

safety checks:
- “pure” aspects

EJB policy revisited

```
public aspect DetectEJBViolations {

    pointcut uiCalls() : call(* java.awt.*+.*(..));

    before() : uiCalls() && cflow(call(* EnterpriseBean+.*(..))) {
      System.err.println("UI call from EJB");
    }

    declare error : uiCalls() && within(EnterpriseBean+)
        : "UI call from EJB";

}
```

“declare error/warning”:
only static pointcuts
(no cflow, this, target, args...)