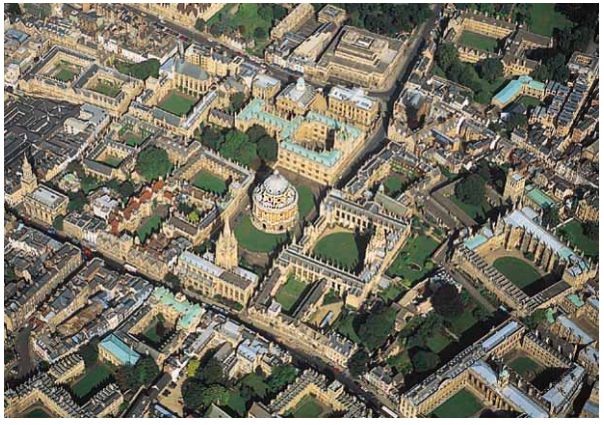
abc - the AspectBench Compiler for AspectJ

McGill

Laurie Hendren
Jennifer Lhoták
Ondřej Lhoták
Chris Goard



Oxford

Oege de Moor
Ganesh Sittampalam
Sascha Kuzins
Pavel Avgustinov
Julian Tibble
Damien Sereni

Aarhus

Aske Simon Christensen

Outline

- AspectJ introduction for compiler writers
- Challenges of building a compiler for AspectJ
- **abc** as an extensible and optimizing compiler
- How **abc** tackles performance issues
- Future Work

AspectJ Programming Language

- a seamless *aspect-oriented* extension to Java
- originally developed at Xerox PARC
- tools for AspectJ now developed and supported by the Eclipse AspectJ project
  - **ajc** compiler for the AspectJ language

(**`http://eclipse.org/aspectj`**)

AspectJ Programming Language

- a seamless *aspect-oriented* extension to Java
- originally developed at Xerox PARC
- tools for AspectJ now developed and supported by the Eclipse AspectJ project
  - **ajc** compiler for the AspectJ language

  (**`http://eclipse.org/aspectj`**)

- **abc**, the **AspectBench Compiler**, is a new, alternative compiler for the AspectJ language, designed for *extensibility* and *optimization* (**`http://aspectbench.org`**)

AspectJ Introduction

- introduce a small Java program, a little expression interpreter
- illustrate three main uses of AspectJ by applying it to this small example
  - aspects for additional static checking at compile time
  - adding fields/classes/constructors to classes via aspects
  - dynamic aspects for applying advice (code) at specified run-time events

Example Java Program - expression interpreter

Consider a small interpreter for an expression language, consisting of:

- SableCC-generated files for scanner, parser and tree utilities in four packages: **`parser`**, **`lexer`**, **`node`** and **`analysis`**.
- main driver class, **`tiny/Main.java`**, which reads the input, invokes parser, evaluates resulting expression tree, prints input expression and result.
- expression evaluator class, **`tiny/Evaluator.java`**

```
> java tiny.Main
Type in a tiny exp followed by Ctrl-d :
3 + 4 * 6 - 7
The result of evaluating: 3 + 4 * 6 - 7
is: 20
```

AspectJ for Static (compile-time) Checking

- Programmer specifies a pattern describing a static program property to look for and a string with the warning text.
- An AspectJ compiler must check where the pattern matches in the program, and issue a compile-time warning (string) for each match.

```
public aspect StyleChecker {
  declare warning :
    set(!final !private * *) &&
    !withincode(void set*(..) ) :
    "Recommend use of a set method.";
}
```

Using the `StyleChecker` aspect

The compilation:

```
abc StyleChecker.java */*.java
```

produces the compile-time output:

```
  parser/TokenIndex.java:34:
  Warning -- Recommend use of a set method.
    index = 4;
    ^-------^
...
```

AspectJ for Intertype Declarations

- Programmer specifies, in a separate aspect, new fields/methods/constructors to be added to existing classes/interfaces.
- An AspectJ compiler must weave in code to implement these additions.
- Other classes in the application can use the added fields/members/constructors.
- In our example, we can use an aspect to add fields and accessors to the code generated by SableCC, without touching the generated classes.

Intertype Declarations - example

All AST nodes generated by SableCC are subclasses of **`node.Node`**.
We must **not** directly modify the code generated by SableCC.

```
public aspect AddValue {
  int node.Node.value; // a new field

  public void node.Node.setValue(int v)
     { value = v; }

  public int node.Node.getValue()
     { return value; }
}
```

Using the AddValue aspect

```
abc AddValue.java */*.java
```

where, the evaluator visitor can be now written using the **value** field to store intermediate values.

```
public void outAMinusExp(AMinusExp n)
  { n.setValue(n.getExp().getValue() -
                  n.getFactor().getValue());
  }
```

instead of the “old" way of storing intermediate values in a hash table. The aspect-oriented method is more efficient because fewer objects are created during the evaluation.

AspectJ for Dynamic Advice

- Programmer specifies a pattern describing run time events, and some extra code (advice) to execute before/after/around those events.
- An AspectJ Compiler must weave the advice into the base program for all potentially matching events.

AspectJ for Dynamic Advice

- Programmer specifies a pattern describing run time events, and some extra code (advice) to execute before/after/around those events.
- An AspectJ Compiler must weave the advice into the base program for all potentially matching events.
- Since events can depend on dynamic information:
  - some execution state may need to be tracked, and
  - some advice may be conditional on the result of a *dynamic residue test*.

Dynamic Advice - counting runtime events

```
public aspect CountEvalAllocs {
  int allocs;  // counter

  before () : call(* *.eval(..)) &&
              within(*.Main)
    { allocs = 0; }

  after () : call(* *.eval(..)) &&
             within(*.Main)
    { System.out.println(
        "*** Eval allocs: " + allocs); }

  before () : call(*.new(..)) &&
              cflow(call(* *.eval(..)))
    { allocs ++; }
}
```

Using the `CountEvalAllocs` aspect

- Using the interpreter with the **`CountEvalAllocs`** aspect included.

```
The result of evaluating:
3 + 4 * 6 + 9 / 3
*** Eval allocations: 17
is: 30
```

- Using the interpreter with the **`CountEvalAllocs`** aspect, and the improved evaluator enabled by the **`addValue`** aspect.

```
The result of evaluating:
3 + 4 * 6 + 9 / 3
*** Eval allocations: 2
is: 30
```

Dynamic Advice - example 2

```
public aspect ExtraParens {
String around() :
 execution(String node.AMultFactor.toString()) ||
 execution(String node.ADivFactor.toString())
 { String normal = proceed();
   return "(" + normal + ")";
 }
}
```

Compile: **abc ExtraParens.java */*.java**
Run: **java tiny.Main**

```
The result of evaluating:
3 + (4 * 6) + (9 / 3)
is: 30
```

Recap: uses of AspectJ for example

- **Static (compile-time) check:** Check that accessor methods are always used to set non-private non-final fields.
- **Intertype declaration:** Add a new field and associated accessor methods to the SableCC-generated **`node.Node`** class.
- **Dynamic advice:**
  - Count the number of allocations peformed during a an expression evaluation.
  - Intercept calls to **`toString()`** for factors and add surrounding parentheses, if they are not already there.

Challenges: front-end

- AspectJ-specific language features, including relatively complex pointcut (patterns) language.
- Intertype declarations, need to be able to extend the type system in non-trivial ways.

Challenges: front-end

- AspectJ-specific language features, including relatively complex pointcut (patterns) language.
- Intertype declarations, need to be able to extend the type system in non-trivial ways.
- **abc**'s solution:
  - use Polyglot, an extensible framework for Java compilers (Cornell)
  - express AspectJ language via LALR(1) grammar: base Java grammar + additional grammar rules for AspectJ
  - use Polyglot's extension mechanisms to override key points in type system to handle intertype declarations.

Challenges: back-end

- Need to handle input from .java and .class files.
- AspectJ compilers need additional modules: matcher, weaver
- need to produce **efficient** woven code (.class files)

Challenges: back-end

- Need to handle input from .java and .class files.
- AspectJ compilers need additional modules: matcher, weaver
- need to produce **efficient** woven code (.class files)
- **abc**'s solution:
  - clean design of matcher and weaver using a simplified and factored pointcut language
  - use Soot, which provides Jimple IR (typed 3-addr), standard optimizations, and an optimization framework

The abc approach

abc has been designed to be an:

- **extensible compiler:**
  - easy to implement language extensions
  - build on two extensible frameworks, Polyglot and Soot
  - see AOSD 2005 submission at **`http://aspectbench.org/techreports`**

The abc approach

abc has been designed to be an:

- **extensible compiler:**
  - easy to implement language extensions
  - build on two extensible frameworks, Polyglot and Soot
  - see AOSD 2005 submission at **`http://aspectbench.org/techreports`**
- **optimizing compiler:**
  - convenient IR
  - good weaving strategies
  - standard compiler optimizations
  - AspectJ-specific optimizations

Does the weaving strategy matter?

- Studied the code produced by **ajc** by tagging instructions that are introduced by the **ajc** weaver and using *J tool to measure dynamic metrics. (OOPSLA 2004)
- When there is **not** a lot of overhead:
  - very simple **before** and **after** advice
  - when the aspect only applies to a small, cold, part of the program
  - when the aspect body is a large computation
- When there can be overhead:
  - frequent (hot) aspects with small bodies
  - frequent (hot) use of cflow and/or around advice

How abc reduces overhead

- use Soot in back-end, so can optimize generated code
- new around weaving strategy
- new cflow implementation

Reducing overhead by using Soot

- the **abc** backend uses Jimple, a typed 3-address IR (**ajc** use stack-based Java bytecode)
  - **abc** weaver does not need to save implicit values on the stack, leads to fewer locals in generated code
  - **abc** weaver can use def-use and variable types to generate better code
- **abc** uses the Soot basic optimizations to clean up generated code
- **abc** can use Soots intra- and inter-procedural analysis frameworks to implement AspectJ-specific optimizations.

Weaving in bytecode (ajc)

```
public int foo(int x, int y, int z)
0:    aload_0
1:    iload_1
2:    iload_2
3:    iload_3
4:    istore          %4
6:    istore          %5
8:    istore          %6
10:   astore          %7
12:   invokestatic    A.aspectOf ()LA; (52)
15:   aload           %7
17:   invokevirtual   A.ajc$before$A$124 (LFoo;)
20:   aload           %7
22:   iload           %6
24:   iload           %5
26:   iload           %4
28:   invokevirtual   Foo.bar (III)I (37)
31:   ireturn
```

Weaving in Jimple (abc)

```
public int foo(int, int, int)
{ Foo this;
    int x, y, z, $i0;
    A theAspect;

    this := @this;
    x := @parameter0;
    y := @parameter1;
    z := @parameter2;
    theAspect = A.aspectOf();
    theAspect.before$0(this);
    $i0 = this.bar(x, y, z);
    return $i0;
}
```

Sascha's strategy for around weaving

- **ajc** has two strategies, inlining around advice, and using closures
  - the inlining method will work well for small advice bodies, or advice the applies in few places
  - the closure strategy is very inefficient, but must be used in some situations (i.e. when an advice applies to itself)
- **abc** has another strategy (**`http://aspectbench.org/theses`**):
  - doesn't inline (no code bloat), but uses generic advice methods
  - replaces polymorphism with lookup tables
  - avoids object creation
  - no closures in the general case
  - uses closures only at specific points, degrades gracefully

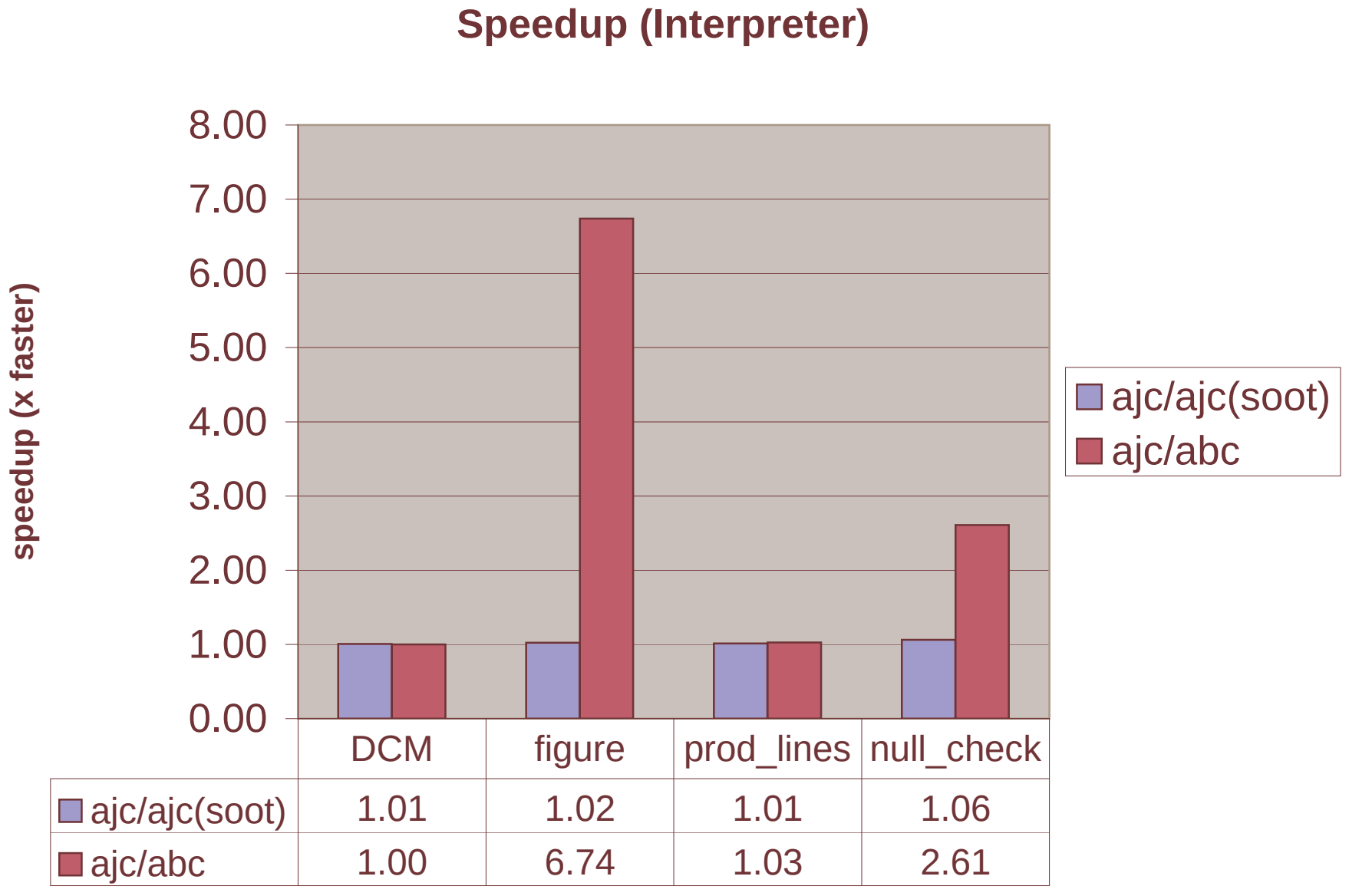
Improving the implementation of cflow

- to track if a runtime computation is within the cflow of some event, the compiler has to generate code to track when that event begins and when it ends
- in general the event may have some state, but most often it does not
- **ajc** uses a stack of states that must be thread-safe
- **abc** improves upon this by:
  - recognizing when there is no state and using a counter instead of a stack of empty states
  - recognizing when counters (or stacks) are equivalent and can be shared
  - only peforming thread-specific operations once per method body
  - will soon use interprocedural analysis to determine if the cflow can be decided statically (and thus no runtime book-keeping is necessary)

Peformance Improvement(1)

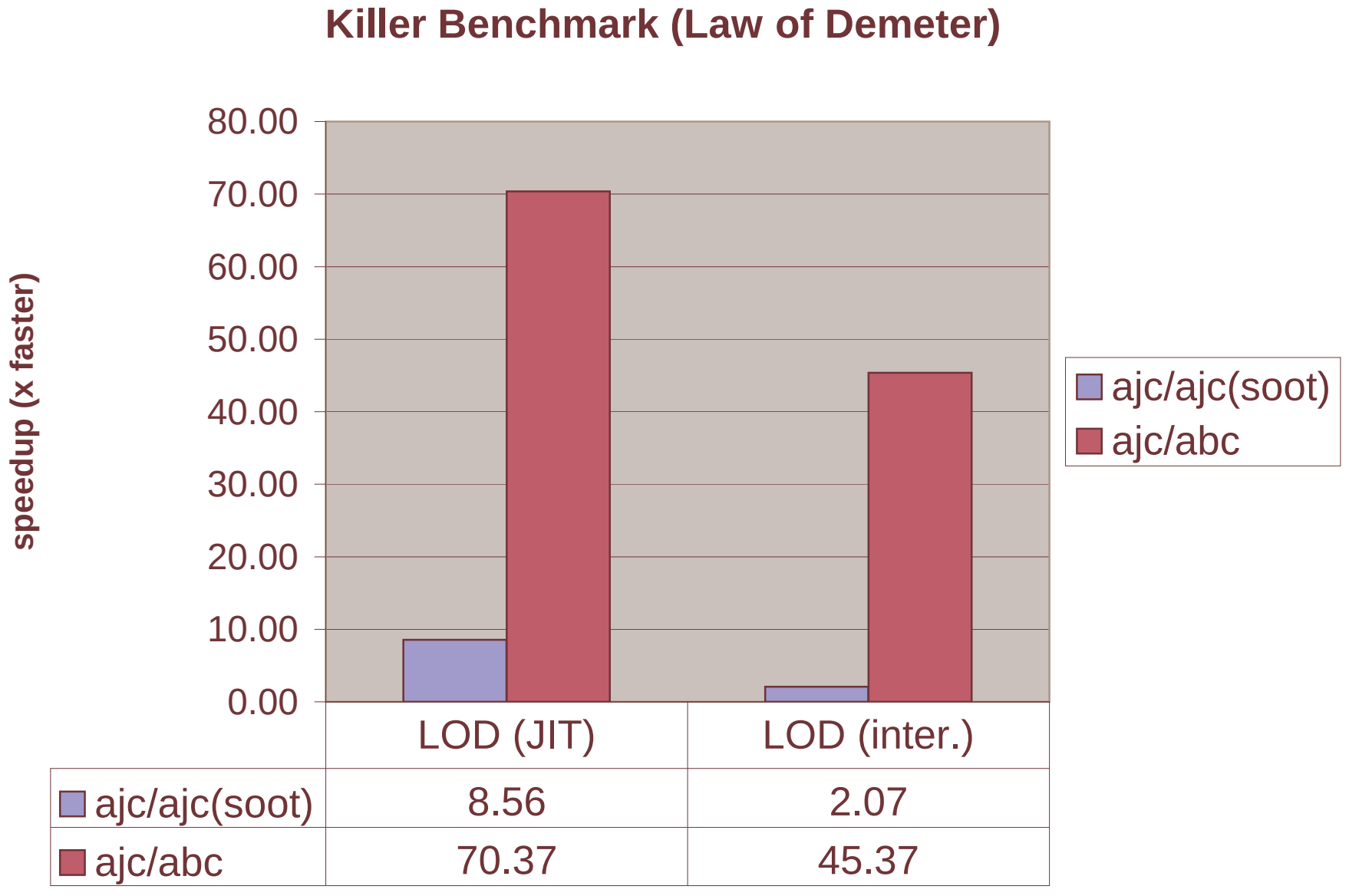
	DCM	figure	prod_lines	null_check
ajc/ajc(soot)	1.00	1.00	1.01	1.02
ajc/abc	1.00	7.45	1.04	4.65

Peformance Improvement(2)



	DCM	figure	prod_lines	null_check
ajc/ajc(soot)	1.01	1.02	1.01	1.06
ajc/abc	1.00	6.74	1.03	2.61

Peformance Improvement(3)



	LOD (JIT)	LOD (inter.)
ajc/ajc(soot)	8.56	2.07
ajc/abc	70.37	45.37

Conclusions

- AspectJ is very useful for many tasks - illustrated with tiny interpreter
- **abc** is a new compiler for AspectJ which is extensible and optimizing.
- You can use **abc** as an alternative AspectJ compiler, or you can use it for research into language extensions and new optimizations.
- It is worth thinking about AspectJ-specific optimizations, and **abc** has already implemented some of these.
- Lots more work by the **abc** team to come ... we welcome users!

`http://aspectbench.org`