Compiling Conditional Pointcuts for User-Level Semantic Pointcuts

Tomoyuki Aotani
Graduate School of Arts and Sciences,
University of Tokyo
aotani@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences,
University of Tokyo
masuhara@acm.org

ABSTRACT

We propose a compilation framework that compiles conditional pointcuts (i.e., `if` pointcuts) in AspectJ for allowing the programmer to define expressive pointcuts without runtime overheads. The framework automatically finds conditional pointcuts that are static with respect to join point shadows, evaluates them at compile-time, and generates compiled code that performs no runtime tests for those pointcuts. By writing conditions that examine static properties of a program, the programmer can simulate many semantic pointcuts within current AspectJ's semantics yet without runtime overheads. Our compiler implementation identifies static conditional pointcuts by using a technique similar to the binding-time analysis in partial evaluation, and employs *double-compilation* scheme to guarantee the same behavior to the code generated by the existing AspectJ compilers. Our experiments confirmed that the compiler is capable of compiling several semantic pointcuts, such as the ones testing class membership (e.g., `has` and `hasfield`), testing join point location (e.g., `withinstaticinitialization`), matching by using regular expressions, and checking high-level coding rules (e.g., the Law of Demeter).

1. INTRODUCTION

Aspect-oriented programming (AOP) helps modularization of crosscutting concerns [17], which can not be well modularized by using existing modularization mechanisms such as procedures and classes. Typical crosscutting concerns that can be modularized with AOP include logging, synchronization, persistence and profiling.

One of the important mechanisms in AOP languages is the *pointcut and advice* mechanism. In fact, many AOP languages including AspectJ [16], AspectWerkz, JBoss AOP, AspectS and AspectC++ support the mechanism. It can be explained in terms of the three elements: *join points, pointcuts* and *advice*. A join point is a point in execution whose behavior can be affected by advice. A pointcut is an expression that selects join points. An advice declaration, which has a pointcut and body statements, is to run the body statements in addition to or in place of the join points matching the pointcut.

Pointcuts are the key element in the pointcut and advice mechanism for making aspects more declarative—robust against program changes, reusable and easy to understand [15]—as they actually specify how the effects of advice should crosscut the base program. With pointcuts that specifies join points by using fundamental properties (e.g., the method signatures or the declaring types), an advice declaration for a complicated crosscutting behavior would have to enumerate each individual join point to be advised.

Expressive pointcut primitives, which identify join points based on higher-level information, offer solutions to the declarativeness of aspects by allowing programmers to write pointcuts that reflect their intention more straightforwardly. Such pointcut descriptions are called *semantic pointcuts*. There are a number of proposals to provide expressive pointcut primitives including the ones to reason about calling context (`cflow`), execution history [27], method/field membership (`has` and `hasfield`) [4], information flow (`dflow`) [20], and relationship among objects (`associated`) [24].

Languages that allow user-defined pointcut primitives could advance the declarativeness of aspects beyond the limitations of the fixed set of expressive pointcut primitives. Since semantic pointcuts tend to be application-specific or to require heavy program analysis, they might not be expressed by merely combining existing pointcut primitives. There have been proposed several extensible AOP languages in which the programmer can define new pointcut primitives [6, 9, 26].

Instead of developing extensible languages, we propose an alternative approach to *simulate* expressive pointcut primitives by using the conditional pointcut primitive in AspectJ. The key idea is to let the programmer write a conditional pointcut with an expression that examines the static properties of join points with the help of reflection APIs in Java and AspectJ. Our proposed compilation framework evaluates the expression at compile-time and generates code without runtime overheads.

The rest of the paper is organized as follows. Section 2 de-

scribes usefulness of the expressive pointcuts, and how they can be provided. Section 3 shows an overview of our proposed approach. Section 4 presents our compilation framework. Section 5 evaluates the expressiveness and performance of our approach with example pointcuts and benchmark tests over numeric applications. Section 6 discusses the related work. Section 7 concludes the paper.

2. POINTCUT AND ADVICE AND EXPRESSIVE POINTCUTS

Expressive pointcut primitives are the key to provide reusable and comprehensible aspects by allowing the programmer to describe pointcuts that reflect their intention more directly. This section briefly presents how the pointcut and advice mechanism works, and how expressive pointcut primitives can improve the reusability and comprehensibility of aspects. We finally review the existing techniques to provide new pointcut primitives with their problems.

2.1 Pointcut and Advice

As an example of the pointcut and advice mechanism in AspectJ, we first show an aspect that profiles number of method invocations in program execution. Among many existing profiling techniques, AOP based techniques run on standard execution platforms with flexibility of selecting profiling targets [7, 14, 16].

Consider simple profiling tasks that count the total number of specific method invocations in a program execution. The set of methods whose invocations are counted can vary. The next definition is an aspect that counts the invocations of the methods that are defined in `EventListener` interface:

```
1 aspect ProfilingEventListenerInvocations {
2   int counter=0;
3   pointcut profiliedCall() :
4     call(* EventListener.*(..));
5   after(): profiliedCall(){ ++counter; }
6 }
```

The second line declares an instance variable `counter`, which acts as similar as instance variables in classes. The third and fourth lines declare a pointcut named `profiledCall`, that identifies any method call whose method signature belongs to `EventListener` interface. The fifth line is an advice declaration that increments `counter` after `profiledCall` happens.

When an application program runs with the aspect, the program runs the body of the advice after invoking any method in `EventListener` interface.

Compared with an approach that specifies each method call expression to profile, the above approach specifies all method call expressions to profile in a concise manner.

2.2 Expressive Pointcut Primitives

Pointcuts largely affect the maintainability, reusability and comprehensiveness of aspects. Expressive pointcut primitives are the key to improve pointcuts.

In the above example, `profiledCall` pointcut abstracts the policy that specifies the set of methods to profile. For example, in order to profile the number of interface calls to either `EventListener` or `Runnable`, we merely need to replace `profiledCall` pointcut with the following one, which combines two pointcut expressions by `||` operator:

```
1 pointcut profiledCall():
2   call(* EventListener.*(..)) ||
3   call(* Runnable.*(..));
```

AspectJ offers many pointcut primitives to specify complicated conditions. For example, we can profile method invocations that are performed in specific packages by using `within` pointcut.

More expressive pointcut primitives contribute to define more robust, easier to understand, and more concise pointcuts, which often scalled *semantic* pointcuts as they reflect programmer's intention at higher-levels. Assume one wants to profile the number of calls to all interface methods[1]. One approach is to declare `profiledCall` pointcut by enumerating all interface names:

```
1 pointcut profiledCall():
2   call(* EventListener.*(..)) ||
3   call(* Runnable.*(..)) ||
4   call(* Clonable.*(..)) ||
5   call(* Comparable.*(..)) ||
6   ...;
```

If there is a hypothetical pointcut primitive `isInterface()` that identifies whether the method to be called is an interface method, the pointcut becomes as follows:

```
1 pointcut profiledCall():
2   call(* *.*(..)) || isInterface();
```

We can observe a number of advantages in the latter pointcut declaration. It is more robust against changes in the base program; the pointcut declaration need not be changed even when interfaces are added to the base program. It is also more easy to understand the intension of the pointcut, which is to profile all interface calls. (In the enumeration based approach, the intention should be guessed by the fact that all the interested types are interface.) It is clearly more concise.

A number of recent AOP language studies propose various expressive pointcut primitives. They include the ones to express properties in the calling context [3, 16], execution history [8, 27], speculative execution [15], data flow [20], and relation between objects [24]. There are also requirements of new pointcut primitives to express conditions that can not be described by merely combining existing pointcut primitives [12].

2.3 User-defined Pointcut Primitives

Although many studies attempt to provide new expressive pointcut primitives built into AOP languages, some AOP languages offer ability mechanisms to define new pointcut

[1]This would be useful to analyze performance characteristics of Java programs as interface calls are usually slower than virtual calls in Java [23].

primitives by the programmer. The mechanisms can be classified into the following two categories:

Extensible languages allow the application programmer to define rules to parse and compile new pointcut primitives. The rules are written in the same language by using a dedicated API [6], in different languages like a logic programming language [10, 26] or a query language [9].

Conditional pointcut with reflection API allows to evaluate arbitrary expressions, including the ones that investigate properties of a program, at runtime.

As this effectively makes it possible to describe expressive pointcuts without extending the language, we propose to use this mechanism as an alternative to the extensible language approach for providing expressive pointcuts.

AspectJ offers a conditional pointcut primitive called `if` pointcut in which arbitrary Boolean expressions can be written inside. By using the reflection API in Java and AspectJ language, it is possible to define pointcuts that examines various properties of a program. The following pointcut tests the same condition as the above hypothetical `isInterface()` pointcut by accessing meta-level information through `thisJoinPoint` object in AspectJ:

```
1 pointcut profiledCall(): call(* *(..)) &&
2   if(isInterface(thisJoinPoint));
3
4 static boolean isInterface(JoinPoint tjp){
5   return tjp.getSignature()
6         .getDeclaringType().isInterface();
7 }
```

When a base program is to call a method, the program evaluates the expression in the `if` pointcut, which invokes `isInterface` method with argument `thisJoinPoint`, which is an object containing meta-level information about the current join point. Method `isInterface` tests whether the method being invoked at the join point is declared in an interface, through reflection APIs in AspectJ (i.e., `getSignature()`, `getDeclaringType()`, and `isInterface()`).

It would be possible to describe a large part of expressive pointcuts by replacing expressions inside the `if` pointcut.

2.4 Comparison of Approaches to Provide User-defined Pointcut Primitives

We identify that the following four criteria illustrate the differences between the above two approaches to user-defined pointcuts. The differences, which are summarized in Table 1, are discussed in the following subsections.

criteria	extensible languages	cond. pointcut +reflection API
evaluation time	compile-time	runtime
evaluation context	before weaving	after weaving
definition language	different	same
information source	compiler	reflection APIs

Table 1: Comparison of approaches to define new pointcut primitives

2.4.1 Evaluation Time

When a language evaluates the new pointcut primitives against join points determines the runtime performance. Whereas the extensible languages do at compile-time for avoiding runtime overheads, conditional pointcuts in AspectJ are evaluated at runtime. This makes it possible to describe dynamic conditions, but the compiled code is tremendously sluggish even if the conditions merely examines static properties of a program.

2.4.2 Evaluation Context

Evaluation context is the environment under which the language evaluates new pointcut primitives. In other words, it is the code base examined by the new pointcut primitives. It can be the code before weaving or after weaving (or in the middle of weaving).

The choice of the evaluation context greatly affects the semantic model for the user-defined pointcuts, and also the implementation. If the evaluation context is the code before weaving, the new pointcut primitives, which might examine class membership or control reachability, can not observe effects from aspects. If the evaluation context is the code after weaving, the implementation of the language would become difficult. The following example explains those trade-offs.

Assume there is a pointcut written by using a user-defined pointcut primitive that judges whether the current join point has any control flow to a constructor of a class that implements the `Runnable` interface. Given the following class definitions, `Starter` implements `Runnable` but `AppFrame` does not. The pointcut hence matches to calls to `initialize` but not to `makeFrame` and `terminate`:

```
1 class Starter implements Runnable { ... }
2 class AppFrame { ... }
3 class Main {
4   void initialize() {...new Starter();...}
5   void makeFrame() {...new AppFrame();...}
6   void terminate() {/* empty */}
7 }
```

When there is an aspect that adds a class member and modifies control flow, the pointcut would match differently. For example, when the above program runs with the aspect shown in Listing 1, the inter-type declaration (ll.2–4) makes `AppFrame` class to implement `Runnable`, and the advice (ll.6–8) makes executions of `terminate` method to create a `Starter` object. Consequently, the abovementioned pointcut should match calls to `makeFrame` and `terminate` as well.

Most existing extensible languages use the code before weaving as the evaluation context, although the authors failed in finding clear semantics in terms of evaluation context. It seems to be reasonable to provide the code before weaving as only the code after weaving is not usually available at

Listing 1: An aspect which changes the class hierarchy and control flow

```
1 aspect InterfaceAndControlFlowChanging {
2   declare parents:
3     AppFrame implements Runnable;
4   void AppFrame.run() { ... }
5
6   before():execution(void Main.terminate()){
7     ... new Starter() ...
8   }
9 }
```

the compile-time. An alternative semantics would provide the code after weaving only inter-type declarations. In this case, `initialize` and `makeFrame` match the pointcut but `terminate` does not. However, it would be difficult to provide the code after weaving advice declarations. In fact, such a system could easily cause a paradox (e.g., when the control flow from the join point can reach a call to this method, advise so as not to perform the operation [15]).

Contrary, the conditional pointcuts are evaluated under the evaluation environment with the code after weaving. This is simply because the evaluation of conditions is taken place at runtime. In other words, the effects of other aspects are visible to the conditional pointcuts.

Although both evaluation contexts have advantages and disadvantages, we believe that the code after weaving gives a comprehensible model to the programmers who define new pointcut primitives.

2.4.3 Definition Language

In order to define a new pointcut primitives, most extensible languages use different languages (e.g., logic programming languages or query languages) from the host (i.e., AspectJ) language. Some (e.g., Josh) use a similar (i.e., Java) language, but require to use a special-purpose API to obtain properties of a program. Conversely, the conditions in the conditional pointcuts are written in the same (i.e., AspectJ) language and can use standard reflection APIs (i.e., those in Java and AspectJ).

Both choices of definition languages have advantages and disadvantages. Special-purpose languages/APIs are good for defining pointcut primitive concisely, but the programmers may need to learn a new language or API.

Another difference between the choices of definition languages: whether aspects affect the definitions of new pointcut primitives. If we had a rich aspect library such as for optimizations, we may want to use the library for implementing new pointcut primitives. Defining new pointcut primitives within the same language gives a chance to apply such aspects to the definitions being written.

2.4.4 Information Source

Where the new pointcut primitives obtain information about program properties can affect the expressiveness of the pointcuts. The extensible languages usually provides information from internals of the compiler. On the other hand, conditional pointcut rely on the standard reflection APIs in Java and AspectJ as the means of obtaining program properties.

Conditional pointcuts are less expressive since the standard reflection APIs do not provide detailed information about the program, such as control and data dependency, which is basically available inside the compiler.

However, we believe that the standard reflection APIs already provide information to describe many interesting pointcut primitives, including the one to test interface methods (e.g., `isInterface`), field and method membership (e.g., `has` and `hasfield` proposed for JBoss AOP and AspectWerkz) and to check most of the Law of Demeter [18]. In addition, if we plugged in a richer reflection API such as Javassist [5], more expressive pointcut primitives could be described by using conditional pointcuts.

3. OUR APPROACH

Our compilation framework evaluates conditional pointcuts at compile-time when the conditions inside the conditional pointcuts are *static*. With this framework, we aim to let the programmer to simulate new pointcut primitives by writing conditional pointcuts without introducing runtime overheads. Even though our approach evaluates conditional pointcuts at compile-time, it has the same characteristics to the conditional pointcuts approach discussed in Section 2.4.

Before explaining the compilation framework, we discuss several prerequisites in our approach in this section.

3.1 Compilation of AspectJ Programs

We explain our compilation framework as a modified version of an AspectJ compiler. An existing AspectJ compiler processes a program consisting of base class definitions and aspect definitions in the following ways [1, 11, 21]:

1. After parsing the program, it visits each parse tree node that creates join points at runtime. Such a node is called *a join point shadow* [11, 21].
2. For each advice declaration in the aspect definitions, there are the next two cases:
   (a) If the pointcut of the advice tests only static properties of the join point (e.g., `call` or `within` pointcut), it decides whether the pointcut matches, and when matches, it inserts instructions that run the advice body around the join point shadow.
   (b) If the pointcut can test dynamic properties (e.g., `args`, `cflow`, or `if` pointcut), it inserts instructions that run the advice body around the shadow with a guard that tests the dynamic properties. When the advice has an `if` pointcut, it generates a boolean method that returns a value of the expression in the pointcut, and the guard invokes the method with the values of the free variables in the expression. We call the generated method *the conditional method* of the pointcut.

For example, the compilation of the `Logger` class in Listing 2 with the profiling aspect in Section 2.1 with the conditional pointcut in Section 2.3 yields the code in Listing 3.

Listing 2: Base Class Definition

```
1 class Logger {
2   List list;
3   public void addMessage(String msg){
4     list.add(msg);//interface call
5   }}
```

Since the line 4 in Listing 2 is an interface call, and the pointcut tests by calling `isInterface` method inside an `if` pointcut, the compiled code in Listing 3 has instructions[2] that creates a (dynamic) join point (ll.3–4), and calls (l.6) the conditional method (ll.11-12) that is generated from the expression inside the conditional pointcut. If the result is true, it runs the body of advice by calling (ll.7–8) a method generated from the advice body.

Listing 3: Compiled Code by Plain AspectJ Compiler

```
1 class Logger {
2   public void addMessage(String msg){
3     JoinPoint jp =
4       Factory.makeJP(ajc$tjp_0, this, list, msg);
5     list.add(msg);//interface call
6     if (Profiling.ajc$if_0(jp))
7       Profiling.aspectOf().
8               ajc$after$Profiling$1$bc0f1d0d();
9   }}
10 class Profiling {
11   boolean ajc$if_0(JoinPoint jp) {
12     return isInterface(jp); }
13   void ajc$after$Profiling$1$bc0f1d0d() {
14     ...}}
```

3.2 Static Conditional Pointcuts

A conditional pointcut is *static* when the expression in the pointcut always has the same value with respect to each join point shadow. For example, `profiledCall` pointcut in Section 2.3 is static because the expression inside always gives true for calls to interface methods.

Although the definition of staticness is simple, we should note the next two points:

- In order to evaluate conditional pointcuts that use essential reflection APIs, our compilation framework assumes that the same set of classes are given at compile-time and at runtime. Without this assumption, the reflection APIs, whose result may depend on the classes provided to the virtual machine, could return different results over different runs.
- Our compilation framework *automatically* finds static conditional pointcuts and evaluates them at compilation. This is done by analysis similar to binding-time analysis in partial evaluation techniques. An alternative approach is to let the programmer explicitly declare static pointcuts by adding annotations. We chose the automatic approach because (1) it is easier to programmers, and (2) the alternative approach would also need a similar analysis in order to reject incorrectly annotated pointcuts.

[2]We decompiled and edited the actual compiled code for readability.

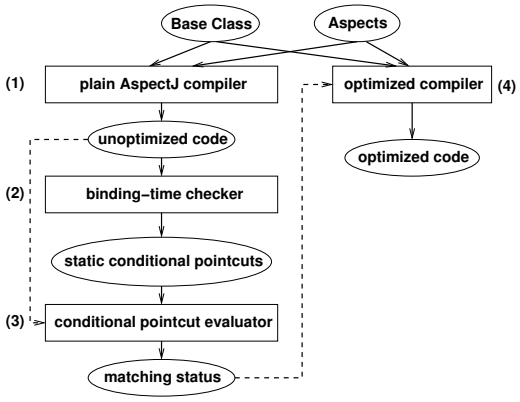

Figure 1: Overview of Double Compilation

4. DOUBLE COMPILATION FRAMEWORK

We propose a *double compilation* technique in order to ensure that the analysis and evaluation of pointcuts correctly uses the code after weaving as the evaluation context. The compilation takes place in the following steps, as also illustrated in Figure 1:

1. A plain AspectJ compiler compiles the whole program and generates the unoptimized code.
2. By analyzing the unoptimized code, the binding-time checker selects all static conditional pointcuts.
3. The conditional pointcut evaluator evaluates the expressions in the static conditional pointcuts with respect to each join point shadow. The evaluation use the unoptimized code as the evaluation context.
4. The customized compiler recompiles the whole program. For advice declarations with static conditional pointcuts, the compiler does/does not insert instructions to call advice body based on the matching status computed in the previous step.

4.1 First Compilation

We use the AspectBench Compiler (abc) [1] to generate unoptimized code. We extended the compiler so that it records each conditional pointcut and the free variables in the conditional expression. Note that the expression is compiled into a method in the unoptimized code.

4.2 Binding-Time Checking

The binding-time checker judges whether the conditional pointcuts are static or not by applying the following rules. The rules approximate the intuitive notion of static conditional pointcuts in Section 3.2:

A conditional pointcut `if(`e`)` is static when the expression e satisfies the next two:

- e accesses variables other than `thisJoinPoint`, `thisJoinPointStaticPart` and `thisEnclosingJoinPointStaticPart`. (In other words, e accesses no global variables and variables bound by other pointcuts.)

- All methods called in e are static[3], as defined below.

A method is *static* (as opposed to *dynamic*) when the body satisfies the next three:

- It accesses no global variables.
- It accesses no dynamic fields/methods of specific classes (e.g., `JoinPoint.getThis()`).
- All methods called in the body are static.

We can intuitively understand that an expression that satisfies the above rules does not have different values with respect to the same join point shadow, because the evaluation of the expression can use only values derived from constants or the specific fields in `thisJoinPoint` object that are constant with respect to the join point shadow.

The formal rules are given in a similar style to typing rules, whose details are omitted in the paper.

4.2.1 Class Set

Since a naive implementation of the binding-time checking is not precise enough and inefficient, we devise an improved technique to minimize the number of classes examined by the analysis.

The above rules need to examine all the methods that can be reachable from the expression in a conditional pointcut. If there is a call to a method defined in a class that has many descendants, it has to examine every respective method defined in each subclass. This not only degrades the efficiency of the analysis, but also degrades the precision of the analysis.

Our technique reduces the number of methods to be investigated by analyzing a set of classes that can be created during the evaluation of the expression in the conditional pointcut. Since an expression in a static conditional pointcut has no access to objects that are created outside the expression, we can have a set of classes that are possibly created during the evaluation of the expression. By using the set, the target of the analysis can be limited to the methods defined in the classes included in the set.

4.3 Evaluation of Conditional Pointcuts

Given a static conditional pointcut, the evaluator matches the pointcut at each join point shadow, and records the matching status in a table for later use. To do so, the evaluator calls a method that is generated from the expression in the conditional pointcut with appropriate parameters. The generated method takes parameters for `thisJoinPoint` etc., the evaluator builds a `JoinPoint` filled with only static information such as signatures and source code locations. This information is also generated at first compilation time.

The evaluator is responsible to provide an accurate evaluation context to the expressions in the conditional pointcuts.

[3]Note that this definition includes both instance methods and class methods.

Since evaluation of a conditional pointcut is done by calling a method in the unoptimized code, investigation through reflection APIs automatically gives information in the unoptimized, or after weaving, code, rather than the code before weaving.

When the evaluator catches an unhandled exception during the evaluation a conditional pointcut with respect to a join point shadow. In this case, it records the matching status as "dynamic" so that the later process do not optimize the pointcut at the shadow at all. An alternative design would report a compile error for an unhandled exception, but it would be annoying if the exception is raised by dead code.

4.4 Second Compilation

The customized compiler for the second compilation generates optimized code. When it visits a join point shadow that can match an advice declaration with a static conditional pointcut, it retrieves the matching status of the pointcut with respect to the join point shadow. If the matching status is "matched" it inserts the instructions to invoke advice body without guards. If the matching status is "unmatched", it just does nothing. If the matching status is "dynamic", it inserts the instructions with guards, as if no optimizations are done.

5. EVALUATION

An implementation of the proposed compilation framework, which is built by modifying AspectBench Compiler (abc), is evaluated in terms of expressiveness and efficiency.

5.1 Expressiveness Evaluation

In order to verify that the reasonably many expressive pointcuts can be described by means of conditional pointcuts, and to verify whether our compilation framework generates the compiled code with no runtime overheads for those pointcuts, we have described several expressive pointcuts including the ones proposed by the other researchers. We confirmed that the following pointcuts can be written and compiled out in our framework:

1. `isInterface`
2. `withinstaticinitialization` [12]
3. `has` and `hasfield` [4]
4. regular expressions
5. checking violations of the Law of Demeter [19]

As we already have seen the first pointcut, we present the implementations of the rest pointcuts below.

5.1.1 WITHINSTATICINITIALIZATION

This is a kind of lexical pointcut that matches join points which are in the body of a staticinitialization method. We can emulate this by using `thisEnclosingJoinPointStaticPart` as Listing 4. The after advice prints the static fields of the class `Target` which is initialized in the static initializer of the class.

Listing 4: withinstaticinitialization

```
1 static boolean withinstaticinitialization(
2           JoinPoint.StaticPart ejp,String cname){
3  return cname.equals(ejp.getSignature()
4                          .getDeclaringTypeName())&&
5      ejp.getKind()==JoinPoint.STATICINITIALIZATION;
6 }
7 after():if(withinstaticinitialization(
8             thisEnclosingJoinPointStaticPart,
9             "Target"))&&
10         set(static * Target.*){
11  System.out.println(thisJoinPoint);
12 }
```

Listing 5: hasmethod

```
1 static boolean hasMethod(Class c,String name,
2        Class return_type,Class[] param_types){
3  Class current=c;
4  do{
5  Method[] methods=current.getDeclaredMethods();
6   for(int i=0,end=methods.length;i<end;++i){
7    if(methods[i].getReturnType()==return_type&&
8     Arrays.equals(methods[i].getParameterTypes(),
9                   param_types)&&
10    name.equals(methods[i].getName()))
11     return true;}
12   current=current.getSuperclass();
13  }while(current!=null);
14  return false;}
```

5.1.2 HAS and HASFIELD

These two pointcut primitives check whether a class has the specific method or field. We can describe them as Listing 5. Though we ignore the modifiers and asterisk pattern matching, we can easily emulate them.

5.1.3 Pointcuts with Regular Expressions

We can define more flexible pointcuts with these than asterisks in AspectJ. For example, when counting up the method calls of which name is constructed with only lower-case alphabets, the aspect looks like follows.

```
1 public aspect Profiler{
2  static int counter_=0;
3  pointcut lower_case_methods_called():
4   call(* *(..)) && (!within(Profiler)) &&
5   if(thisJoinPoint.getSignature().
6            getName().matches("^[a-z]+$"));
7  before() : lower_case_methods_called(){
8   ++counter_;
9  }
10 }
```

We have a conditional pointcut with regular expressions in the fifth and sixth line. Since its result is statically available, we can evaluate it at compile time.

5.1.4 Checking Violations of Law of Demeter

The most violations of Law of Demeter (LoD) [19] is statically checkable with our compiler. The checked law excludes preferred-acquaintance classes from preferred-supplier classes since our compiler cannot evaluate the conditional pointcuts which use global variables statically.

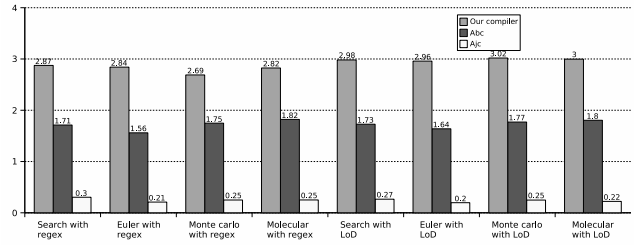

Figure 3: Compilation Times Relative to abc Compiler without Aspects

As [18] shows the messages which warn the violations are available only at runtime in our example while our compiler evaluates all conditional pointcuts at compile time. This is because currently we cannot use any conditional pointcuts in the inter-type declarations.

5.2 Performance Evaluation

We measured compilation and execution performance of our compiler in order to see how much our double compilation scheme adds to the compilation time, and to see the compiled code generated by our compiler does not have runtime overheads, which was significant with existing compilers.

We used Java Grande benchmark suites as base application, and compiled them with aspects that counts the number of invocations to the methods (1) whose names consist of only lowercase characters, and (2) which violate the Law of Demeter. As comparison, we compiled and executed the same combinations of programs with the AspectJ compiler version 1.2 (ajc) and the AspectBench compiler version 1.0.0 (abc). We also measured the performance of the base programs without aspects, and with aspects that cache the results.

All benchmark tests were executed on top of Sun HotSpot Client Java VM 1.5.0 with the system libraries in J2SE 1.4.2_06, running on dual Xeon 3.06GHz Linux machine with 6GB memory.

5.2.1 Execution Speed

Since our compiler eliminates all runtime checks for the conditional pointcuts, the compiled code runs faster than the code generated by the other compilers. Figure 2 shows the execution times relative to the respective applications compiled by the abc compiler without aspects.

As we can see, the code compiled with our compiler takes longer time than the code compiled without aspects by the factors of 1 to 1.08. We conjecture the overheads are due to the body of aspects. If the compiled code evaluates pointcuts at runtime, the overhead became 32% to 36,929 %. Caching the results of conditions would reduce the overheads, but the resulted code still has overhead up to 36%.

5.2.2 Compilation Speed

Figure 3 shows the compile times of the benchmark programs relative to the compile times by the abc compiler without aspects. Although our compiler employs the double compilation scheme, it merely have 54% to 82% overheads

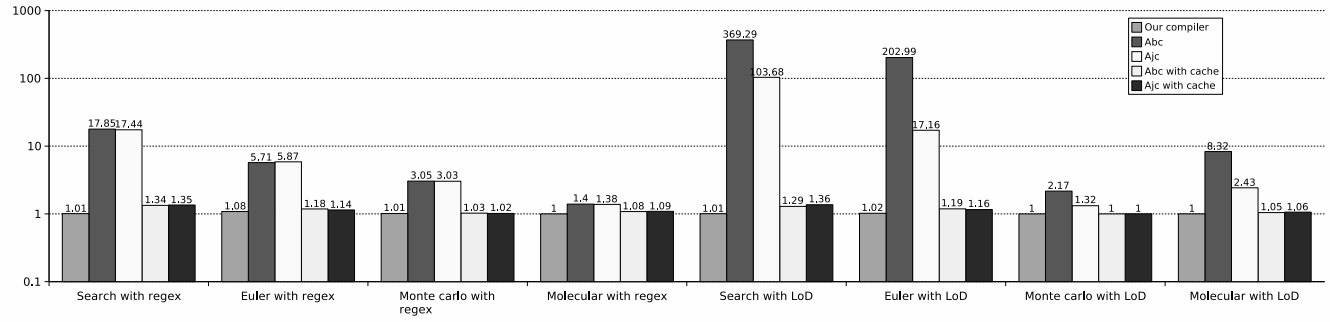

Figure 2: Execution Times Relative to abc Compiler without Aspects

(i.e., less than 100%) compared to the compilation in abc. This would be due to the caching mechanism in the internals of the abc compiler, and also due to our compilation framework that reduces the number of join point shadows being advised.

Since the abc compiler is much slower than the ajc compiler, an implementation that based on ajc would achieve significant speedup in compilation times. We are also planning to use backpatching techniques for the second compilation phase, rather than generating the compiled code from scratch.

6. RELATED WORK

As mentioned, there are incresing number of languages that enable user-defined pointcut primitives [6, 9, 26]. Those languages requires to use a different language or API to define pointcuts. Moreover, the evaluation context of the defined pointcuts may not be runtime.

XAspects is a system that integrates domain-specific aspect languages and component languages [25]. Although the purpose of the language is different from ours, it employs a compilation technique similar to our double compilation scheme.

Our compilation framework can be seen as partial evaluation [13] of conditional pointcuts with respect to join point shadows. However, our framework is much simpler than the standard partial evaluation techniques as our compiler only evaluates expressions that are totally static. If we had a powerful partial evaluator for Java that generaets reasonable code [2], our compilation framework could further optimize the pointcuts in which some of, but not all of the expressions are static with respect to a join point shadow.

7. CONCLUSION

In this paper we proposed a compilation framework for conditional pointcuts in AspectJ-like languages. The framework allows the programmer to describe expressive pointcuts by simply writing conditions in a conditional pointcut without introducing runtime overheads.

We also pointed out that the evaluation context of the user-defined pointcut primitives could change the semantics of the defined pointcuts. Our compilation framework assures to provide runtime context, or the code after weaving, which is consistent with the views to the ordinary conditional pointcuts.

We implemented the framework by modifying the AspectBench Compiler, which successfully compiles several expressive pointcuts defined by means of conditional pointcuts.

The expressiveness of the conditional pointcuts in our approach can be enriched by employing more powerful reflection APIs. For example, when we move on to the system based on Java 5 (or AspectJ 5), we will be able to define pointcut primitives to examine metadata of programs. If we employ Javassist or similar bytecode analysis tools, the pointcut primitives that uses more elaborated properties such as control dependencies could be written.

The future work includes the following topics. A mechanism to bypass the binding-time checking would allow the programmer to exploit global variables in order to accelerate the evaluation of conditional pointcuts. A load-time pointcut evaluation mechanism could alleviate the prerequisite to provide the same classes at compile and execution times. A backpatching mechanism that transforms the unoptimized code into the optimized code would be a faster alternative to the second compilation step.

Acknowledgments

We would like to thank all individuals who gave valuable comments on our work and an early draft of the paper, including Kris de Volder and the members of Software Practices Lab at University of British Columbia, Karl Liberherr and Mich Wand and the members of their group at Northeastern University, Kenichi Asai, Shigeru Chiba and the members of his group at Tokyo Institute of Technology, the members of Kumiki research project, Tetsuo Tamai and his group at University of Tokyo, the members of PPP group, and the SPLAT'05 organizers.

8. REFERENCES

[1] P. Avgustinov, et al. abc: An extensible AspectJ compiler. In *AOSD.05*, 2005. to apper.

[2] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *PEPM'00*, pp.2–11, 2000.

[3] J. Brichau, W. D. Meuter, and K. D. Volder. Jumping aspects. In *ECOOP Workshop on Aspects and Dimensions of Concerns*, 2000.

[4] B. Burke, A. Brok. Aspect-oriented programming and JBoss. The O'Reilly Network, 2003. `http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html`

[5] S. Chiba. Load-time structural reflection in Java. *ECOOP2000*, pp.313–336, 2000.

[6] S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. In [22], pp.102–111.

[7] J. Davies, et al. An aspect oriented performance analysis environment. In *AOSD'03 Practitioner Report*, 2003.

[8] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, eds., *Aspect-Oriented Software Development*, pp.201–217. Addison-Wesley, 2005.

[9] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *APLAS'04*, pp.366–382, 2004.

[10] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD'03*, pp.60–69. 2003.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In [22], pp.26–35.

[12] W. Isberg, within{static}initialization — was Re: withincode(clinit). *AspectJ Users' Mailing List.* `https://dev.eclipse.org/mailman/listinfo/aspectj-users`. December, 2004.

[13] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, 1993.

[14] T. Kamio and H. Masuhara. A value profiler for assisting object-oriented program specialization. In *Proc. of Workshop on New Approaches to Software Construction (WNASC 2004)*, pp.95–102. 2004.

[15] G. Kiczales. The fun has just begun. Keynote talk at AOSD'03, 2003.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, et al. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[17] G. Kiczales, et al. Aspect-oriented programming. In *ECOOP'97*, pp.220–242. 1997.

[18] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: checking the law of Demeter with AspectJ. In *AOSD'03*, pp.40–49. 2003.

[19] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[20] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS'03*, pp.105–121, 2003.

[21] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC2003*, pp.46–60, 2003.

[22] G. C. Murphy and K. J. Lieberherr, editors. *Proceedings of AOSD 2004*, March, 2004.

[23] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *OOPSLA'01*, pp.195–210, 2001.

[24] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD'04*, pp.16–25. 2004.

[25] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *OOPSLA'03 Domain-Driven Development Track*, 2003.

[26] K. D. Volder. Aspect-Oriented Logic Meta Programming. In *Proceedings of Reflection '99*, pp.250–272. 1999.

[27] R. J. Walker and G. C. Murphy. Implicit context: easing software evolution and reuse. In *FSE'00*, pp.69–78, 2000.