Optimizing the Evaluation of Patterns in Pointcuts

Remko Bijker, Christoph Bockisch
Software Engineering Group
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands
r.bijker@student.utwente.nl,
c.m.bockisch@cs.utwente.nl

Andreas Sewe
Software Technology Group
Technische Universität Darmstadt
Hochschulstr. 10
64289 Darmstadt
Germany
sewe@st.informatik.tu-darmstadt.de

ABSTRACT

Pointcuts in aspect-oriented programming languages specify runtime events which cause execution of additional functionality. Hereby, pointcuts typically have a pattern-based static component selecting instructions whose execution triggers an event, e.g., a pattern that selects method-call instructions based on the target method's name. Current implementations realize identification of matching instructions by examining all instructions in the executed program and matching them against all patterns found in the program's pointcuts. But such an implementation is slow. An optimized implementation is therefore highly desirable in runtime environments which support the dynamic deployment of aspects; slow pattern evaluation invariably causes a slowdown of the entire application.

The patterns used in pointcuts as well as the signatures against they are matched, i.e., method, constructor, and field signatures, are well structured. We present two case studies that survey patterns and signatures actually occurring in the wild. From the resulting data we derive several heuristics that can drive pattern-evaluation optimizations, both by creating indexes over the relevant instructions and by optimizing the order in which the sub-patterns are evaluated.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Procedures, functions, and subroutines*

General Terms

Languages, Measurement

Keywords

Aspect-oriented programming, patterns, signatures, pattern matching, pointcuts

1. INTRODUCTION

In many flavors of aspect-oriented programming languages behavior is specified by pairs of pointcuts and advice [13]. Advice define some functionality and pointcuts are Boolean predicates which are evaluated against events, called *join points*, which occur during the execution of the program; whenever a pointcut evaluates to **true** at a join point, the associated advice is executed. Join points are associated with *join-point shadows*, i.e., sequences of instructions executing during the join point in question. Join-point shadows are in turn associated with a member. The list of a method body's instructions can, e.g., be a join-point shadow, but so can a single method-invocation instruction or an instruction accessing a field. The join-point shadows most commonly supported refer to a method, field, constructor, or static initializer.

The primitive predicates in pointcut expressions are called pointcut designators. They fall into two categories: those that are evaluated solely based on syntactic, lexical, and type-based properties of a join point's shadow and those that are evaluated based on the dynamic state in which the join point is executed. The prevalent implementation [14, 1, 6, 5, 10] of aspect-oriented languages is to partially evaluate all static pointcut designators against all join-point-shadow instructions in the program. This determines the join-point shadows whose execution may potentially be matched by the whole pointcut. At the corresponding instructions, *residual* code is inserted that tests whether the part of the pointcut not statically evaluable also evaluates to **true**. Only if this is the case, the advice's functionality is executed.

The statically-evaluable pointcut designators specify the kind of join-point shadow they match, e.g., methods calls, together with a pattern for the associated member's signature. Evaluating the patterns of all static pointcut designators against the associated members of all join-point shadows in a program can be costly. In execution environments that support aspect deployment at runtime, pattern evaluations are also performed at runtime and consequently slow down the application as a whole; thus, optimizations are desirable.

In this paper we present a case study that makes it possible to estimate the potential benefit of optimizations applied to pattern evaluation. The targeted optimizations exploit the structured nature of member signatures and patterns: Method, constructor, and field signatures are comprised of multiple sections and for every section of a signature a pattern defines a sub-pattern. Only when all sub-patterns match the corresponding sections of a signature, the pattern matches as a whole.

What exactly comprises the signature, and what can be referred to in a pattern, depends on the concrete pointcut language. For instance, a method signature always has a name, parameter types, and a return type, while some languages also allow to pattern-match against, e.g., the declaring class or the modifiers of a method.

We propose to treat signatures similar to entries in a database table and patterns similar to queries of these entries; for each kind of signature one table exists with the different sections of the signature as columns. Two mechanisms known from query optimization for databases are then explored to assess their value in optimizing pattern evaluation: the use of indexes on the signature tables and the ordered evaluation of sub-patterns according to their individual selectivity.

After presenting the relevant optimization mechanisms, we present our survey of signatures and patterns found in real-world applications. This shows how the selectivity of sub-patterns can be heuristically estimated and which sub-patterns are actually common. This allows to judge whether the additional effort of maintaining an index will eventually pay off.

2. BACKGROUND

In this section we will first describe the structure of signatures and patterns relevant to our survey (cf. Section 3). We will then discuss database query optimization techniques that are applicable to the domain of pointcut matching.

2.1 Signatures and Patterns

The survey presented in this paper is based on the semantics of signatures and patterns in the ALIA4J[1] approach [4, 6]. As we have shown in the past, the meta-model of aspects and the associated execution semantics are able to support, e.g., the languages AspectJ, Compose*, and JAsCo [7]. Thus, basing this work on ALIA4J allows us to draw conclusions that are applicable to multiple aspect-oriented languages at once.

ALIA4J supports five kinds of patterns: the **method pattern**, the **constructor pattern**, the **static initializer pattern**, the **field read pattern**, and the **field write pattern**. The sub-patterns used by them are as follows:

Modifiers patterns are used to match the modifiers of a method, constructor, or field. Examples are **private**, **public**, but also **static**, **final**, and many more. It is possible to specify an inverse modifier, e.g., !**final**.

Type patterns are used to match parameter types as well as the return type of methods or the type of fields. The type is matched by either its class or primitive name, package name, or by being a subclass of another class.

Class type patterns are used to match the enclosing type of accessed methods, constructors, static initializers, and fields. They are equivalent to type patterns, except that primitive types are not allowed.

Name patterns are used to match the name of a method or field. Matching can be done with exact matches or by means of regular expressions.

[1]See `http://www.alia4j.org/`.

Parameters patterns are used to match the parameters of a method or constructor. Matching is done by means of matching type patterns for the parameters including a wild card type that greedily matches any number of parameters.

Exceptions patterns are used to match the checked exceptions that can be thrown by a method or constructor. Matching is done by means of matching type patterns for the exceptions.

All sub-patterns can be composed by means of logical connectives. It is thus possible to negate the whole pattern, requiring matching multiple patterns, or requiring one of a set of patterns to match.

ALIA4J treats call sites as join-point shadows, i.e., instructions that either invoke a method or constructor, or read or write a field. At these points, only the name of the accessed method or field as well as the parameter types, the return type (for fields also the field type) are known. To evaluate patterns referring to the declaring class, modifiers or declared exception types, it is necessary to resolve the actual method that is executed at a call, as only the method declaration contains this information. But because method calls are polymorphic in Java, the method actually executed upon a call depends on the receiver object and is not known statically.

As a compromise, ALIA4J organizes methods into sets, called *generic functions*, and matches patterns against the *top method* of a called generic function. A generic function [15] is the logical combination of all non-private methods with the same name and parameter types that are declared in the same type hierarchy. The top method is the method definition contained in the class closest to the root of the type hierarchy; all other methods in the generic function override the top method (and possibly each other). The declaring class, modifiers, and declared exceptions are all matched against the top method. This is a reasonable semantics as all overriding methods must conform to the top method's contract to some degree. For instance, the visibility of a method cannot be reduced by an overriding method and the overriding method cannot declare exceptions beyond the ones declared by the top method.

2.2 Database Query Optimization Techniques

There are several database optimization techniques that might be used to speed up finding the projection of pointcuts. These techniques will be explained and evaluated in this section. This evaluation strongly depends on the kind of queries that are performed. In the context of this paper, queries are coarsely separated into those using an *exact pattern* and those using a *pattern using wildcards*. The reader is assumed to be familiar with basic SQL as documented in the literature [2, 9].

2.2.1 Indices

Rows in a database table are generally unsorted. Thus, finding an entry that matches a query requires visiting each row and evaluating the query. As a faster alternative, an index can be used to find the matching rows in the table when evaluating a pattern. Essentially, an index is a data structure that allows to quickly map from a key value to a row in a table. There are many different ways to implement indices, but the most common ones are based on sorted $B+$ trees or hashing [2].

The chosen implementation has an effect on the performance of maintaining the index as well as of evaluating exact patterns and patterns with wildcards. We thus must discuss the computational complexity of the different operations and also the space overhead required for storing the index. While the theoretical complexities of the necessary operations on the different data structures are known (e.g., [2] of [16, Page 267], another survey is needed to determine their actual impact in practice. We, therefore, leave a discussion about the trade-offs of the different data structures as a subject of future work.

2.2.2 *Search-Plan Optimization*

The generic-function lookup, i.e., the pattern matching, corresponds to a selection on the call-sites table. It is to be expected that the content of this table is not random and that therefore not all entries have the same probability. The different probabilities for certain entries can be derived from statistical data about the different (sub-)parts of generic-function signatures. These statistics lead to the "selectivity factor" (F) for certain sub-patterns which can be exploited in search-plan optimization. This is comparable to the selectivity of predicates of a **WHERE** clause in a SQL statement [17]. The F of each sub-pattern in a concrete pattern can then be used to order the predicates so the ones with the lowest F are evaluated first; the plan is to remove as many rows from the resultset as early as possible. The F for a predicate is determined based on the estimated uniqueness of the values in the table.

3. A SURVEY OF REAL-WORLD SIGNATURES AND PATTERNS

In the previous section we have presented indexes and search-plan optimization as possible optimizations for pattern evaluations. In order to apply these mechanisms some knowledge about the structure of the data on which queries are performed is needed. For instance, it must be known for which sections of signatures the effort of keeping an index may pay off; for search-plan optimization, heuristics are needed to estimate the selectivity of certain sub-patterns. In this section a survey of real-world applications and aspects is presented to discuss common structures of signatures and patterns that can drive these optimizations.

Section 3.1 discusses what are the general characteristics of call sites and generic functions in real-world applications. Section 3.2 discusses what are the general characteristics of patterns in real-world aspects. Finally Section 3.3 formulates optimization strategies based on the findings of both sections.

3.1 Call site characteristics

For determining what performance optimizations are suitable for pattern matching it is necessary to analyze real-world applications. From these applications the call sites need to be extracted to determine the data that is significant for the optimizations like the selectivity of parts of a signature. For example, when all functions return the same type the selectivity of that return type is low. For any other type, however, the selectivity is very high as it would never select anything.

3.1.1 *Methodology*

To acquire the call-site information a small tool, called Extract, has been written to extract the information from Java class files. Extract uses ASM,[2] a Java bytecode engineering framework, to read the Java class files and ALIA4J to determine what generic function a call site belongs to.

The input to Extract is a list of Java class files or Java archives with class files of a particular application. All Java class files and all classes they reference are analyzed recursively. Examples of referenced classes are the super class, classes used as return or parameter type, classes of instance variables and classes used in the implementation of a method. The final result of Extract is a list with all generic functions and the number of times they were referenced.

The recursive behavior of the analysis means that a large part of the Java API's implementation will be part of the output. However, the implementation of the Java API should be seen as a black box and one would generally not want to add aspects that modify the API's implementation. AspectJ, e.g., thus applies aspects only to the files that it compiles itself; this excludes the class files of the Java API. These internal calls are therefore ignored by this survey. Nevertheless, calls to the Java API from the application are of importance. After the extraction the data is analyzed using common data mining techniques to find correlations and significant information. The applications that have been analyzed in this survey are the following four:

ANTLRv3 A tool to construct parsers, compilers and translators.

FreeCol A turn-based strategy game.

LIAM A major part of the implementation of ALIA4J.

TightVNC A remote control software package.

These applications are varied in nature in an attempt to get a general view of the call-site characteristics instead of only getting a view of non-graphical applications that do not use a network connection, as is commonly the case for benchmark suites used in performance evaluation [3].

Larger applications such as Eclipse, an integrated development environment, and OpenOffice, an office suite, have been considered. But these applications have a complex system of plug-ins and dependencies which makes it hard for Extract to cover all referenced classes. (Tools like TamiFlex [8] can alleviate this problem, but were not yet available when the study was conducted.)

3.1.2 *Acquired information*

Statistics.

The analyzed applications have 2 432 classes containing a total of 28 065 generic functions and 150 432 call sites. The following sections will discuss the most salient features separately for each generic function kind; only the breakdown on class names is presented in a single section as it is almost the same in all cases.

Storage.

Per generic function kind there is also a discussion on the best storage technique for quickly retrieving call sites per

[2]See `http://asm.ow2.org/`.

sub-pattern kind on the basis of the information gathered. The techniques considered here are "bucket arrays," like hash tables, and "sorted collections," like B+-trees.

3.1.3 Class names

Statistics.
The combined test input consists of 2 432 different classes spread over 147 packages, on average placing roughly 16.5 classes in each package.

Depth	Amount	Percentage
0	20	0.8%
1	135	5.6%
2	414	17.0%
3	479	19.7%
4	333	13.7%
5	525	21.6%
6	526	21.6%

Table 1: Package depth for class names

Table 1 shows the classes' package depth, i.e., the number of super packages a class has before reaching the "unnamed" default package. The 20 classes with depth 0, i.e., the ones placed in the unnamed package, and 135 classes with depth 1 do not comply with the standard practice of using a reverse DNS name. The classes at depth 2 are primarily from the Java API, whereas classes at depths of 4 and more are almost exclusively used by applications.

Due to the practice of using a reverse DNS name the first few levels of package naming do not help in quickly reducing the search space. The way package names are generally constructed, however, makes it reasonably easy to determine a start and end point in a sorted set and thus reduce the amount of checks: Assume the classes of the Java API are lexically sorted by their fully qualified name and are put in a sorted set. Then all classes that match the java.lang..$*$ pattern can be found by calculating a subset: The starting point of the range is constructed by removing .$*$ from the pattern, the end point is constructed by replacing ..$*$ with / (U+002F), whose codepoint the Unicode character encoding places immediately after the . (U+002E). Calculating the subset only requires two $O(log\,n)$ comparisons.

Storage.
When looking at the storage techniques the first option is using a sorted collection. This has the benefit of ordering all classes lexically by name; thus, the aforementioned technique can be used. However, storing all classes into buckets by package can be used to efficiently look up all classes in a given package, but due to the use of sub-packages one has to determine how to find all classes that are in a particular package or sub-package. Either the buckets also contain references to sub-packages which are then recursively searched or each class gets inserted into its package and all ancestor packages. A major drawback of the latter technique is the fact that the number of times the class is referenced is equivalent to the package depth plus one. The former technique resembles the behavior of a sorted collection.

3.1.4 Static initializers

Statistics.
A total of 571 static initializer were found. The static initializers do not have a call site, i.e., they are called implicitly, and the name, modifiers, return type, parameters, and exceptions are the same for every static initializer. As such the only way to distinguish between static initializers is their containing class.

Storage.
Static initializers only have one sub-pattern: the declaring-class pattern. Thus, the same considerations as in Section 3.1.3 apply.

3.1.5 Constructors

Statistics.
A total of 3 034 constructors were found; on average about 1.25 constructors have been found per class.

Modifier	Amount	Percentage
package visible	759	25.0%
public	1 971	65.0%
private	151	5.0%
protected	153	5.0%
transient	14	0.5%
annotation	53	1.7%
deprecated	7	0.2%

Table 2: Modifier usage for constructors

Table 2 shows how often a particular modifier is used for a constructor. The first set of four modifiers, the ones that govern access to the constructor, are mutually exclusive and cover all constructors. As a result the total of the first four modifiers is always exactly 100%. The other modifiers are optional and multiple of them can be used per constructor.

# Parameters	Amount	Percentage
0	669	22.1%
1	1 171	38.6%
2	720	23.7%
3	241	7.9%
4	144	4.7%
5+	89	2.9%

Table 3: Parameter usage for constructors

Table 3 relates the number of parameters to the amount of constructors having such a parameter count. The 3 034 constructors declare a total of 4 420 parameters, giving about 1.5 parameters per constructor. Of all these parameters 588 are of type java.lang.String, 564 are of type **int** and 158 are of type **boolean**, from a total of 518 parameters types.

Of the 3 034 constructors 2 855 do not throw an exception, leaving 179 that do throw at least one. A total of 188 exceptions is declared; a few constructors declare more than one exception. The exceptions most frequently thrown by the programs surveyed are javax.xml.stream.XMLSteamException

(84 times) and java.io.IOException (40 times), which are both part of the Java API.

A total of 14 526 call sites of constructors were found. Of these 64% call a constructor from the Java API; only 36% of the calls are for creating application-specific classes. Classes from the java.lang package are constructed 39% of the time, with the StringBuffer/Builder being responsible for over 50% of the calls.

Storage.

For the modifiers a bucket array is best as there are only a few valid different buckets to consider. However, there are sometimes multiple buckets a constructor would match against. In that case they have to be put in all, but this is not a big problem as a relatively small amount would be placed in multiple buckets. It has to be considered whether physically storing the **public** modifiers bucket is needed at all, as it matches the vast majority of constructors.

We consider three techniques to store parameters: First, the number of parameters can be used to determine the bucket. Second, the first parameter type can be used. (If a function has no parameters, **void** is used as first parameter.) Third, the concatenation of all parameter-type names can be stored lexically sorted.

The benefit of the first technique is that searching for a particular amount of parameters is extremely efficient, whereas the second is efficient in finding constructors that have a particular type as first parameter. The third technique is well suited for finding constructors that start with particular parameters, but finding constructors with a given length requires looking through the whole collection.

Finally the exceptions can best be stored in a bucket as well. Here, each declared exception is put into a bucket. Given the low amount of actually declared exceptions and the low amount of constructors with more than one declared exception this does not impact storage much. The constructors that do not throw an exception are not stored specially.

3.1.6 Field reads and writes

Statistics.

In total, 3 884 fields have been found, yielding 1.6 fields per class. We furthermore found 27 979 field reads and 9 752 field writes, respectively 7.2 and 2.5 per field. Roughly 16.4% are reads from and 12.4% are writes to classes from the Java API. (All **static final** reads are ignored because the majority of call sites can optimized them away by the compiler.)

In total, there are 2 697 different field names. The most frequently used field name logger is used 38 times; thus, the field names themselves are all quite selective. The majority, 95.9%, of the field names start with a lower case letter. Of the 3.2% of field names that start with an upper case letter 55.6% are static. About 0.9% of all names start with either a $ (U+0024) or a _ (U+005F).

The field names have an average length of 9.7 characters. Around 99% of these characters are letters. Looking at the first three characters does not show any discernible patterns; the most common prefix, can, is used in less than 1.75% of the field names. This means that field names diverge relatively fast.

The length and divergence of the field names can be used to estimate the time required for one comparison of two strings. If, e.g., at most 1.75% of the field names start with the same three letters one knows that in three comparisons there is at most a 1.75% chance that further characters have to be examined.

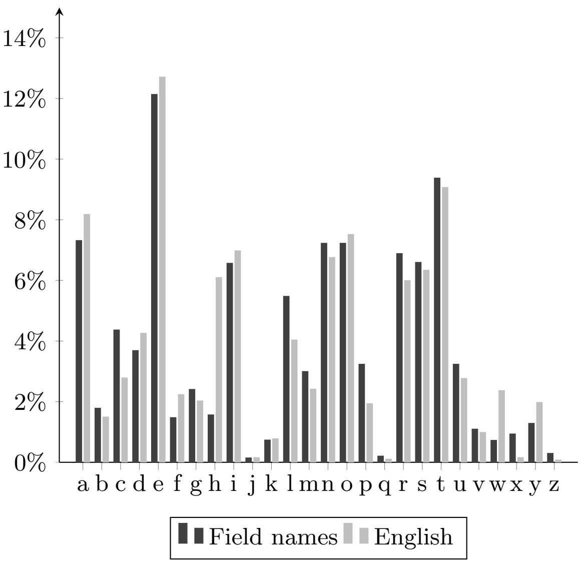

Figure 1: Letter frequencies of field names compared to English

Figure 1 shows the letter frequencies in the field names which are fairly similar to the letter frequencies of English, although there are a few letters whose frequency differs significantly.

Modifier	Amount	Percentage
package visible	862	22.1%
public	236	6.1%
private	2 189	56.4%
protected	597	15.4%
static	395	10.2%
volatile	8	0.2%
transient	50	1.3%
annotation	22	0.6%
deprecated	9	0.2%

Table 4: Modifier usage for fields

Table 4 shows how often a particular modifier is used for a field. This table is almost identical to Table 2 with the exception that the percentages for **public** and **private** modifiers are switched; fields are about ten times more often **private** than constructors are. (The situation is reversed for **public**.)

Of the field types 53% are either a primitive or comes from the java.lang package, with **int**, **boolean** and java.lang.String taking respectively 22%, 12%, and 10% of the total.

Field reads and writes cannot throw checked exceptions.

Storage.

For storing the name a sorted collection makes finding a particular name or a range easy. Using a bucket array is possible, but either the whole name has to be hashed or only the first character is taken into account. In the former case doing a name range lookup becomes expensive, whereas in the latter case one still has to go through a long list of items after the first bucket. If one were to chain the buckets per character one would in effect be building a sorted collection.

The modifiers can only be stored in a bucket array due to the limited amount of options. It can be considered to not create a physical bucket for the **private** fields as they match more than half of the fields and as such are not very selective.

The type of the field can best be stored in a sorted collection. There are quite a number of types, although it is very conceivable that a set of types from one package is considered. In that case having a collection sorted on the type name would make getting those ranges work in the same way as for class names. However, it is conceivable to store the data in a bucket array if there can be, e.g., due to a less powerful pointcut language, no range lookups on the type of a field.

3.1.7 Methods

Statistics.

A total of 17 294 methods were found, which means about 7.1 methods have been found per class. There are 6 812 different names for the methods, yielding 2.5 methods with the same name.

The average length of a method name is 12.3 characters; almost 3 characters more than field names. Around 99% of the characters are a letter.

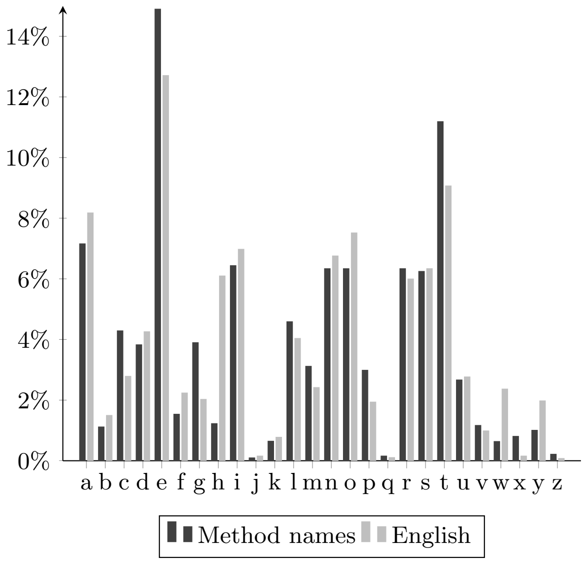

Figure 2: Letter frequencies of method names compared to English

Figure 2 shows the letter frequency in the method names which is less similar to the letter frequency in English. Letters “c”, “e”, and “t” are used significantly more often whereas the usage of letters such as “h”, “w”, and “y” have dropped by up to 75%.

Contrary to the insignificance of the first three letters of field names the first three letters of method names are significant as can be seen in Table 5. What has to be noted is that 417 of the 499 methods that start with “acc” are `access$n` functions created by the compiler for inner classes that try to access outer classes.

Name	Amount	Percentage
get	4 596	25.6%
set	1 346	7.8%
cre	541	3.1%
acc	499	2.9%
add	413	2.4%

Table 5: Frequency of first three letters in method name

Modifier	Amount	Percentage
package visible	1 022	5.9%
public	13 399	77.5%
private	1 450	8.4%
protected	1 423	8.2%
static	2 178	12.6%
final	1 527	8.8%
synchronized	315	1.8%
volatile	127	0.7%
transient	62	0.4%
native	58	0.3%
abstract	1 623	9.4%
annotation	558	3.2%
deprecated	24	0.1%

Table 6: Modifier usage for methods

Table 6 shows how often a particular modifier is used for a method. As multiple modifiers can be used at the same time the total is more than 100%. What immediately catches one’s eye is that almost 80% of the methods are **public** and that furthermore a large portion is **static**.

Modifier	Amount	Percentage
package visible	545	25.1%
public	1 331	61.1%
private	301	13.8%
protected	1	0.0%
final	353	16.2%
synchronized	23	1.1%
volatile	1	0.0%
transient	8	0.4%
native	25	1.1%
annotation	432	19.8%
deprecated	7	0.3%

Table 7: Modifier usage for static methods

Table 7 shows the modifiers, but only for **static** methods. What stands out is the majority of package visible methods are also **static** and that there are almost no **protected static** methods. (Note that **abstract static** methods are impossible as you cannot override static methods.)

Of 37.7% of the methods the return type is **void**. About 36.4% are a primitive type or come from the `java.lang` package, with **int**, **boolean** and java.lang.String respectively taking 10.4%, 10.1%, and 7.7%. When ignoring the **void** return type, these numbers are similar to the type of fields.

When considering the methods starting with "get" and "set" more closely, one finds that there are 6 (0.01%) of the former methods have a return type of **void** and 49 (3.6%) of the latter have a return type different from **void**. This means that when encountering a pattern matching methods that start with "get" or "set" in an aspect one can be fairly sure the method respectively returns or does not return something. Consequently, the return-type check should be done last.

# Parameters	Amount	Percentage
0	6568	38.0%
1	7115	41.1%
2	2247	13.0%
3	728	4.2%
4	342	2.0%
5+	293	1.7%

Table 8: Parameter usage for methods

Table 8 relates the number of parameters to the amount of methods having such a parameter count. With 16 838 parameters there are roughly 0.94 parameters per method. That is about 0.5 less parameters per method than parameters per constructor.

Of all passed parameters 11 595 (68.9%) have a type which is part of the Java API, so 5 243 (31.1%) have a custom type. Of the parameters from the Java API 8 335 (49.5%) are either a primitive or from the java.lang package. Looking at the specific types of parameters the following can be gathered: 3 218 (19.1%) are **int**, 1 859 (11.0%) are java.lang.String, 970 (5.7%) are **boolean** and 734 (4.4%) are java.lang.Object.

About 7.6% of the methods declare an exception. Of these 1 317 methods 992 declare one exception, 129 declare two, 191 declare three, and 5 declare four. Of the declared exceptions 44% are application-specific and the remaining 56% are exceptions taken for the Java API. The majority, about 70%, of the methods that declare an exception are application methods. About 7.1% of the application-specific methods declare an exception.

A total of 98 175 call sites of methods have been found. Of these, 46.7% call a method from the Java API; 53.3% of the calls are directed at application-specific classes. The java.lang package is called in 21.8% of the cases. Hereby, the classes StringBuffer/Builder are responsible for over 50% of those calls. Most of these string-building calls are generated automatically by the compiler when it encounters the concatenation of strings using the + operator. Other Java API packages that are commonly called are javax.swing and java.awt at 13.9%, and java.util at 9.9%.

Storage.

The storage techniques suitable for the sub-patterns of methods are mostly described above for other generic function types. The name pattern is described in the section about fields, whereas the modifier, parameter and exception pattern are described in the constructor section. The return type, however, should be handled in a unique fashion: While it follows the same technique as the type pattern of fields, it also has a **void** type which the type pattern does not have. As this matches a large part of the methods it should be considered whether actually storing the bucket for **void** types is of use.

3.2 Aspect characteristics

To be able to tell how often a particular part of a pointcut pattern is used one needs to look at real-world aspects and the patterns they use in their pointcuts. For example, if none of the aspects specify a class-name pattern it would be of little use to build up a sorted list of class names.

3.2.1 Methodology

The following AspectJ applications and libraries and the patterns used in them are analyzed:

ajlib-incubator A library with reusable aspects.

Contract4J 5 aspects to support "design by contract" in Java.

Glassbox An application for monitoring other applications by using aspects.

NVersion, RecoveryCache Generic aspects for fault tolerance.

Sable Benchmarks Set of generic aspects used for benchmark purposes.

3.2.2 Acquired information

In total, 170 different patterns have been found in 242 different aspects. What is noteworthy about the found patterns is that 35.6% of them were found within code that limits the call sites for a particular pattern to a specific class or package. In these cases doing a full search of all call sites would be a waste of time; only looking at an unsorted list of call sites within a particular class or package would suffice.

3.2.3 Static initializers

Two patterns have been found that look at static initializers. One matches for all static initializers, whereas the other matches exactly one class.

3.2.4 Constructors

In total, ten patterns matching a constructor have been found. Of these, four have an "any class" pattern, one has a "any class in package" pattern and the remaining five match a particular class. Two patterns match the **native** modifier and one matches the **public** modifier. It has to be noted, however, that the former two patterns do not make sense, as the **native** modifier is prohibited for constructors by the Java language specification [11, Chapter 8.8.3]; thus these patterns can never match. Furthermore there is one pattern that requires an exception to be declared.

The ten patterns are used by sixteen different pointcuts; six pointcuts use the "match any constructor" pattern, two times a "match any public constructor" pattern. All other patterns have been found only once.

3.2.5 Field reads and writes

We found five field read and four field write patterns. Of these, four matched "any class," three "any class in package," and two patterns matched a specific class. The latter two patterns furthermore matched a specific "name" pattern. Two modifiers patterns matched **public** methods, and two others matched on method respectively being **static** or not being **static**.

Only the "match any field read/write" patterns were found multiple times; three and two times for field reads and field writes, respectively.

3.2.6 Methods

The majority of patterns are method patterns; 149 of the found patterns match a method call. 36 of these match "any class," 2 match "any class in package," and the remaining 111 match a specific class. Of these, 75 classes match a class from the Java API.

18 patterns match a method with "any name," 23 match the "first few characters of a name," one matches the "last characters of a name," and the remaining 107 match a specific name. A small number of the same name patterns were found in different patterns, however, none in more than three patterns. In total there are 103 unique patterns.

10 patterns match on the **public**, 2 on the **native**, and 1 on the **static** modifier. Also, 3 patterns match methods without the **static** modifier. One of the methods matches both the **public** and the **static** modifier.

13 return type patterns match **void**, 13 match a specific type and one matches all non-**void** return types. The remaining patterns match any return type.

41 parameter patterns specify that there may be no parameters, 4 specify that there must be exactly one parameter regardless of type, 12 parameters have a specific pattern for the parameters, 3 describe the first parameter but allow more parameters, and the remaining 89 patterns match any parameters.

In total, 4 patterns match declared exceptions; in 2 of these cases any exception suffices whereas in the others a specific exception must be declared to be thrown.

Of the 212 found aspects 67 match any class. Of these, 23 match any name, resulting in 44 cases that describe at least a partial name. The former all match on either **public**, non-**static** or "any" modifiers. Furthermore the matched return types of these 23 are **void**, non-**void** and "any." This means that in the case no class and no name is given the pattern can be considered to be a "match any" pattern in view of using sorted data sets for lookup.

For patterns that match "any class in a package" the other sub-patterns are generally match "any" sub-patterns with the exception of an occasional match **public** or match "throws java.lang.Throwable". This means that in that case the package could be used as initial data set.

Of the 212 found patterns 143 match a specific class and two match a class within a specific package. Of these, 143 only six have a match "any method name" pattern. 44 of the remaining 67 patterns match on the method name. This leaves 23 patterns that do not match on the class or method name. Of these, 23 patterns, four are fairly selective as they specify selective parameter types, modifiers or declared exceptions.

3.3 Optimization strategies

This section describes the default optimization strategies that can be extracted from the Sections 3.1 and 3.2. As in those sections the optimization strategy will be discussed per generic-function kind.

3.3.1 Static initializers

Static-initializer patterns can only be evaluated in one way: looking at the class name. To optimize this, the generic functions could be sorted by their full class name to aid the lookup speed. Due to the small amount of patterns matching on static initializers found in the case study it is unclear whether an index data structure can pay off. If such patterns occur sufficiently often, a sorted collection can offer quick access.

3.3.2 Constructors

For constructors there are basically two sub-patterns that can be used for optimization: A sorted collection with the full class names when a class name is known and a bucket array with the call sites per modifier otherwise. Hereby, the sorted list of class names would be the primary way of searching; only if there is no class name to match, one can consider using the bucket array.

In cases where modifiers other than **public** are used in method patterns, matching that modifier using the bucket array should be considered due to the high selectivity of those modifiers. However, when the full class name is very selective, i.e., the class name has no wildcards in it, that would still take precedence.

There are no aspects referring to declared parameters and exceptions of a constructor. Therefore, we cannot deduce an optimal lookup and storage strategy. Until sufficient data is available, we assume that carrying over the results for methods is a reasonable starting point.

3.3.3 Field reads and writes

For field reads and writes the main optimization point is the class name as well. Nevertheless, the modifiers are also useful, as the ones found in the survey are also very selective, with the exception of the **private** modifier.

In contrast, the field name and the type are not worth considering, even though the name is actually very selective in this case. This is due to the fact that it is rarely used and thus the overhead of building the index outweighs the small extra selectivity over the class name which can be used much more often. Alas, a majority of the patterns found in the survey match everything; thus, in these cases there is little to optimize.

3.3.4 Methods

Most methods can be matched by their name. They are matching a specific pattern making them not that selective, e.g., get∗ and set∗. So in reality they match more cases, first checking the class name will yield better initial selectivity unless the class name can be anything. Parameter and return types are the next most selective.

The modifiers of a method are in most cases not selective at all; most of them match more than 75% of the methods. This actually makes the return type more selective than the modifiers for the case where there are no class or method names to match.

Exceptions are rarely used in patterns. Effectively, patterns that do refer to declared exceptions just require that at least one exception is thrown but they do not further specify patterns for the exception types. In the two patterns we found in the case study that do specify a required exception type, the sub-pattern for the declaring class is more

Sub-pattern	Storage	Static init.	Constructor	Field	Method
class name	sorted	1	1	1	1
name	sorted	-	-	3	2
modifiers	bucket	-	2	2	4
type	bucket	-	-	4	-
return type	sorted	-	-	-	3
parameters	sorted	-	3	-	5
exceptions	bucket	-	4	-	6

Table 9: Best storage techniques per sub-pattern and order of use by type

selective than the exceptions sub-pattern. As such, the exceptions sub-pattern should not be evaluated early.

3.3.5 Conclusion

Taking the above analysis into account, we now give a summary of the best storage techniques for improving the performance of the pattern matching in Table 9. Hereby, the numbers represent the order in which to evaluate the sub-patterns.

As can be seen, the class name is the best sub-pattern to start with. This is usually followed by the modifiers, except for the case of methods; here, the name of the method is a more selective secondary search parameter. Methods also have a third search parameter, the return type.

Note, however, that this data depends on the generic functions used by the actual applications and on the aspects used to search for a subset of these. If an application, for example, declares only a few checked exceptions and all aspects look for a particular exception, then a simple bucket array could be the most selective and thus best way to start the search. Another example would be a pattern matching all **public static** methods starting with get. In that case the name sub-pattern would match around 25% of the methods whereas the modifiers sub-pattern matches only 8% and thus the modifiers pattern would be the most selective.

4. RELATED WORK

Masuhara et al. have discussed the implementation of the weaver component in the AspectJ compiler [14]. In particular, they also discuss the algorithm used for finding join-point shadows for pointcuts. As mentioned in the introduction, this entails an evaluation at all join-point shadows in the program. The authors also suggest a mechanism they call *fast match* to rule out the matching on a per-class-basis. This mechanism builds on the fact that each Java class is represented in one bytecode file which contains the so-called *constant pool*, i.e., a table of symbols used in the class. The symbols include all signatures of methods and fields accessed by instructions in this class.

In the fast match approach, patterns are first evaluated against the signatures in the constant pool. If no match was found, the class cannot contain matching join-point shadows. Thus, no expensive parsing of the method bodies' instructions is required in order to find the location of join-point shadows.

This approach does not consider the structure of signatures and patterns themselves, but groups join-point shadows according to the locations in which they occur. The constant pool is a summary of the occurring join-point shadows and is used to exclude some locations from the search as shortcut. Our approach is orthogonal to that and exploits common structures in signatures and patterns themselves.

The STEAMLOOM VIRTUAL MACHINE [12] supports dynamic aspect deployment and therefore also seeks to improve performance of partial pointcut evaluation. STEAMLOOM implements an indexing mechanism that allows to quickly map from matched signatures to the locations of the corresponding join-point shadows. This is similar to the fast match in the AspectJ compiler, but also does not consider heuristics of common signature and pattern structures.

5. CONCLUSIONS AND FUTURE WORK

In this work, we have presented indexes and search-plan optimization as potential mechanisms to optimize partial evaluation of patterns as found in the pointcuts of aspect-oriented languages. Both kinds of optimizations require knowledge of the structure of data against which patterns are evaluated; commonalities in method, constructor, or field signatures and the selectivity of sub-patterns must be known *a priori*. To develop heuristics about both, we have performed a survey on four applications written in Java and five written in AspectJ.

The results of this survey show that depending on the kind of join-point shadow matched by a pattern (method call, field read, etc.) different sub-patterns are reasonably selective. For instance, the sub-pattern on modifiers is very selective for constructors, but in general not very selective for methods. These observations can be used for a context-insensitive search-plan optimization that always orders the evaluation of sub-patterns in the same way for a given kind of pattern.

When the actual pattern is considered in the search-plan optimization, better optimizations are possible. For instance, a modifiers sub-pattern that only matches for **transient** or **volatile** methods is very selective, i.e., it selects less than 1% or the method signatures. In such cases, the modifiers sub-pattern should be evaluated early, although this is not the best strategy in general.

The survey has also shown which kinds of indexes are suitable. For example, a sorted tree allows to efficiently find a range of entries with a common prefix. This means that such an index is beneficial for exact patterns and patterns *ending* with a wildcard. Another beneficial index structure are buckets which map one key to a collection of rows, e.g., buckets can be created that store a list of all method signatures that contain a certain modifier.

In future work, we will extend the presented survey and consider additional, larger programs, both object-oriented and aspect-oriented ones. It is especially desirable to add aspect-oriented programs that use constructor patterns more extensively. We will also investigate using automatic data mining approaches to identify correlations that can be exploited in optimizating the pattern evaluation.

Furthermore, we will implement the optimization strategies and required index data structures in the ALIA4J approach. This will enable us to benchmark the actual impact of these optimizations at runtime. In this future case study, we will compare different implementation strategies for index data structures and evaluate their impact, in both memory usage and execution time, on dynamic operations like insertion of new signatures when classes are loaded.

6. REFERENCES

[1] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *TAOSD*. 2006.

[2] A. J. Bernstein and M. Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2001.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of OOPSLA*, 2006.

[4] C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, 2009.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM.

[6] C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of VMIL*, 2007.

[7] C. Bockisch, A. Sewe, M. Mezini, A. de Roo, W. Havinga, L. Bergmans, and K. de Schutter. Modeling of representative AO languages on top of the reference model. Technical Report AOSD-Europe-TUD-9, Technische Universität Darmstadt, 2008.

[8] E. Bodden, A. Sewe, J. Sinschek, and M. Mezini. Taming Reflection (Extended version). Technical Report TUD-CS-2010-0066, CASED, 2010.

[9] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.

[10] A. de Roo, M. Hendriks, W. Havinga, P. Dürr, and L. Bergmans. Compose*: A language- and platform-independent aspect compiler for composition filters. In *Proceedings of WASDeTT*, 2008.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification*. Addison-Wesley, 3rd edition, 2005.

[12] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Technische Universität Darmstadt, 2006.

[13] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP*, 2003.

[14] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of CC*, 2003.

[15] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *Proceedings of OOPSLA*, 2008.

[16] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 1999.

[17] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of SIGMOD*, 1979.