MASTER THESIS

OPTIMIZATION OF ASPECT-INSTANTIATION STRATEGIES IN THE JIT-COMPILER

M.D. Zandberg

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE
CHAIR SOFTWARE ENGINEERING

EXAMINATION COMMITTEE
Dr.-Ing. Christoph Bockisch
Dr. ir. Lodewijk Bergmans

DOCUMENT NUMBER
EWI/SE – 2012-003

UNIVERSITY OF TWENTE.

JAN 2012

Optimization of Aspect-Instantiation Strategies in the JIT-compiler

Thesis of a master project in context of the study Computer Science at the University of Twente

Author:
Martin Zandberg

Study:
Computer Science

Institute:
University of Twente

Supervisors:
Dr.-Ing. Christoph Bockisch
Dr. ir. Lodewijk Bergmans

Date:
January, 31, 2012

Abstract

Aspect-instantiation policies define the rules how aspect instances are stored, looked-up and instantiated. Current aspect-oriented languages implementations are restricted to generating bytecode for realizing the semantics of aspect instantiation. Often however, bytecode is not expressive enough to support the most efficient optimizations. The ALIA4J framework for implementing execution environments with support for aspect-oriented languages allows implementing the semantics of language features in terms of bytecode generation and JIT-compilation. In this thesis a JIT compilation strategy for aspect-instantiation is developed that supports a variety of aspect-instantiation policies, ranging from singleton and per-object policies to per-multiple-object policies. The compilation strategy is benchmarked and its performance is compared to state-of-the-art aspect-oriented language implementations.

Table of Contents

List of Abbreviations

AD	Advanced Dispatching
AOP	Aspect-Oriented Programming
ALIA	Advanced-dispatching Language-Implementation Architecture
ALIA4J	Advanced-dispatching Language-Implementation Architecture For Java
FIAL	Framework for Implementing Advanced-dispatching Languages
JIT	Just in Time
JVM	Java Virtual Machine
LIAM	Language-Independent Advanced-dispatching Meta-model
MMTk	The Memory Management Toolkit MMTk
PA	Pointcut-and-Advice
RVM	Research Virtual Machine

Chapter 1

Introduction

In the recent years several new programming languages have been developed that aim to increase the separation of concerns. The separation of concerns allows a developer to develop software that is logically decomposed and can therefore focus on a single concern instead of focusing on everything at once [**1**]. The new languages are based on the manipulation of dispatch and because they go beyond the traditional object-oriented dispatch, they are called advanced dispatching languages [**2**]. Examples of advanced dispatch concepts in these languages are multiple dispatch, predicate dispatch and pointcut-advice. The last concept plays a central role in this thesis and for convenience reasons we abbreviate it to PA.

A main responsibility of PA-languages [**3**], like for instance AspectJ [**4**] and AspectWerkz [**5**], is how to define an aspect-instantiation policy. The aspect-instantiation policy determines how aspects instances are instantiated, stored and looked-up. Since these actions are not under the direct control of the client[1], mechanisms exists that realize at run-time the instantiation, storage and look-up. An example of an aspect-instantiation policy in AspectJ is the per-target policy. Next chapters will show that there exist more policies and also their exact definition will be described.

The instantiation and look-up of aspect instances is task that must be performed fast. The aspect instance serves as the context to run the execution of one or more advices. If advices are executed often then aspect instances are also instantiated and/or looked-up that often. Therefore it is feasible that these actions must be performed fast since it can take a substantial part of the overall execution time if advice is executed often.

The mechanisms that determine the instantiation, look-up and storage of an aspect-instantiation policy form the semantics of the policy. As just mentioned more policies exist and research has showed that some policies are closely related to each other [**6**] [**4**]. In [**7**] a generalized model is defined that captures the most used aspect-instantiation policies found in mainstream PA-languages. The generalized model discusses the instantiation, look-up and storage mechanisms of each policy in detail. The aspect-instantiation policies that are going to be optimized in this work are all captured by the generalized model and this model therefore plays a fundamental role in this work and is elaborated in chapter two.

As stated in the previous paragraph, the instantiation and look-up mechanism should be fast because it can affect a substantial part of the overall execution time. In PA-languages like AspectJ and AspectWerkz, these mechanisms are compiled to the same intermediate representation as the intermediate representation of the base program. For instance, in AspectJ these mechanisms are

[1] A client can be, for example, the programmer working with aspect-oriented program.

represented as Java bytecode instructions. In the end, the bytecode instructions of the mechanisms plus the bytecode instructions of the base program are woven to a final program.

In order to execute the final program, the bytecode instructions are compiled straightforward to machine code by the JIT-compiler. It is straightforward in the sense that it just compiles the bytecode to machine code without knowing that the final program is actually a woven program that consists of the mechanisms plus base program. Because the JIT-compiler does not have knowledge about these mechanisms it cannot apply optimizations that are based on semantics of these mechanisms. This fact introduces a so-called semantic gap [**8**].

The ALIA4J framework [**9**] provides various solutions for closing the semantic gap. The solutions have all in common that the semantic gap is closed by preserving semantics of aspect-instantiation policies. The difference between the solutions is how the implementation is accomplished. There are implementation styles that are based on a high-level of abstraction whereby semantics are implemented as plain Java method, but there exist also styles that work on a much lower level of abstraction that allows machine code to be generated. In this work the latter style is chosen. We mentioned in the first section that aspect instantiation and/or look-up should be fast because it can take a substantial part of overall execution time. Since the focus of this work is on *optimization*, the latter style looks most promising in order to achieve the best optimizations. The challenge of this work is therefore to come up with a solution whereby semantics of the mechanisms are preserved until the latest moment in time so that the JIT-compiler can produce machine that is optimized according to semantics of the mechanisms. The generalized model plays thereby an important role because it captures the most used aspect-instantiation policies.

We implement the solution in the Advanced-dispatching Language-Implementation Architecture for Java (ALIA4J) framework. ALIA4J provides a foundation for sharing the implementation of overlapping advanced dispatching concepts. ALIA4J consist of two major components, LIAM and FIAL that work in a well coordinated fashion. LIAM maps advanced dispatching concepts to a model that is based on a single meta-model. Once advanced dispatching concepts are mapped to a model, it is, together with bytecode that does not use advanced dispatching, compiled to an intermediate representation. In order to execute the intermediate representation, it is passed to one of ALIA4J's execution environments. When the meta-model is going to be executed, ALIA4J instantiates a FIAL component that processes the model conforming to the single model. Due to the clear separation of advanced dispatch concepts and the execution environment, ALIA4J provides a foundation to implement the solution in modular way. One part of the solution can be implemented in LIAM whereas the other part can be implemented in Steamloom^{ALIA}, one of ALIA4J execution environments. Furthermore, ALIA4J provides services and modules that can be used and/or redefined when implementing the solution. Chapter discusses ALIA4J in more detail.

The approach that we follow in order to realize the solution is as follows:

1. We develop a new LIAM entity that contains semantics of the aspect-instantiation policies described in [**7**].
2. We adapt the JIT-compiler of Steamloom^{ALIA} that allows the new LIAM entity to generate machine code that is based on semantics of aspect-instantiation policies.

The new LIAM entity that is going to be developed inherits from an existing LIAM entity. The existing LIAM entity implements the generalized model that in turn models semantics of aspect-instantiation policies. However, the existing implementation uses the high-level style which means that semantics are implemented as plain Java instructions. In chapter four this existing LIAM entity is elaborated and it is shown that the high-level approach is good for experimentation but not for optimization purposes. The new LIAM entity uses the low-level approach and is able to generate machine code. Basically the new LIAM entity becomes a part of the JIT-compiler and temporary takes over control of code generation. When it generates machine code, it does that according to semantics of the generalized model. The look-up, storage and instantiation mechanisms thus work more efficient leading to a better overall execution time.

In order to assess the effectiveness of the optimizations made we defined a series of experiments. A benchmark platform measures the execution time of several aspect-instantiation policies and compares them against implementations in AspectJ [**4**] and Association Aspects [**6**].

The remainder of this thesis is structured as follows:

- Chapter two presents the generalized model;
- Chapter three discusses the required technical background on ALIA4J;
- Chapter four gives an analysis of the problem;
- Chapter five discusses the implementation;
- Chapter six provides the experiments;
- Chapter seven gives the conclusion and provides suggestions for future work.

Chapter 2

The Generalized Model

This chapter discusses the generalized model [**7**] for aspect-instantiation policies. The purpose of an instantiation policy is to define the rules according to which aspects are instantiated. Because aspects in mainstream PA-languages are instantiated automatically, instantiation policies are inevitable. Instantiation policies manage the rules according to *instance storage and look-up*, *instantiation*, and *restriction*.

- The *instance storage and look-up* mechanisms take care how aspect instances are stored after they have been created and how they can be looked-up next time.

- The *instantiation* mechanism decides when a new aspect instance must be created or if an existing instance must be looked-up.

- The *restriction* mechanism has the responsibility for restricting aspect instance to participate in a selection of aspect instances.

The first half of this chapter discusses the aspect-instantiation policies that are used in this work. We discuss the policies with respect to *instance storage and lookup*, *instantiation*, and *restriction*. The second half of this chapter explains how the generalized model is developed. It turns out one aspect-instantiation policy forms the basis of the generalized model.

2.1 Association aspects

The first instantiation policy is the association aspect policy and is the most general one. The other policies, described in the next sections, are all considered to be special cases of this policy. The association aspects concept is described by Sakurai et al. [**6**] and is used to associate an aspect instance to a group of objects. This concept therefore supports behavioral relations amongst a group of objects. With behavioral relations it is, for instance, possible to keep the state between two objects synchronized. The next example gives an illustration.

Consider a class `Bit` shown in listing 2.1. A bit object has one instance variable `value` that represents a boolean value and three instance methods `set`, `clear` and `get`, that respectively set the value to true, set the value to false and return the value. In a real application it might be the case that the state of two bit objects must stay synchronized. For this example it means that setting or clearing one bit causes the value of the other bit to be set or cleared. In order to associate two bit objects with each other and synchronize their states, Sakurai et al. invented association aspects.

```
1   class Bit {
2       boolean value = false;
3
4       void set()      { value = true;  }
5       void clear()    { value = false; }
6       boolean get()   { return value; }
7   }
```

Listing 2-1 – Class `Bit`

Association aspects are realized as an extension to AspectJ. Thereby three keywords are added: 1) the `perobjects` modifier, the `associate` statement and the `associated` pointcut designator. Listing 2.2 shows an association aspect called `Equality` that implements the example.

The `perobjects` modifier on line 1 defines that the aspect associates two objects of type `Bit`. The `associate` statement in the aspect's constructor on line 6 creates the association. As is visible, two objects of type `Bit` called `left` and `right` are associated. The `associated` pointcut designator on line 13 and 19 is used in a pointcut expression to retrieve the aspect instance that is associated with the objects that are present in a join point. The retrieved aspect instance is then used as context to execute one or more advices. In the example two pointcuts plus their advices are defined. Furthermore, the example shows that a pointcut designator supports the use of wildcards. This means that for one pointcut multiple aspect instances can be retrieved and that thus multiple advices can execute.

For instance, consider that the following three associations are made. The first association consists of `<b1:Bit, b2:Bit>`, the second of `<b1:Bit, b3:Bit>`, and the third of `<b2:Bit, b3:Bit>`. Then next, a method executes whereby the following join point shadow is reached:

```
b1.set();
```

The join point is matched by the first pointcut (line 11-13). Namely: The pointcut designator on line 11 matches the signature of the join point. The target (receiver) object is `b1:Bit` and is bound the formal parameter `left`. The associate pointcut designator where `b1` plays the left-hand side object and a wildcard plays the right-hand side object matches two associations. Therefore the pointcut retrieves in total two aspect instances and the advice is executed twice.

The association aspect-instantiation policy manages the rules according to *instance storage and lookup*, *instantiation*, and *restriction* as follows:

The *instance storage and lookup* mechanism: As the example shows, the aspect instance is associated to two objects of type `Bit` and therefore the aspect instance is *stored* in a field of both objects that participate in an association. This is because the aspect instance can be looked-up via both objects. In a case of a lookup via the left object, the right object is checked to see if it exists in the association. If so, then the look-up retrieves the associated aspect instance. In a case of a look-up via the right object, the left object is checked and so on.

The *instantiation* mechanism: Association aspects must be instantiated explicitly. This includes that a group of objects must be manually associated to an aspect instance. For instance: in order to associate objects `b1:Bit` and `b2:Bit` to an `Equality` aspect instance, one has to call the aspect's constructor like `new Equality (b1, b2)`.

The *restriction* mechanism: As explained, association aspects follow an explicit instantiation strategy. When no association is made between objects, then no aspect instance will be retrieved. Thus, it is restricted in the sense that if no association is made, no aspects instance will be retrieved.

```
1   aspect Equality perobjects(Bit, Bit) {
2       Bit left;
3       Bit right;
4
5       Equality(Bit left, Bit right) {
6           associate(left, right);
7           this.left = left;
8           this.right = right;
9       }
10
11      after(Bit left) : call(void Bit.set()) &&
12          target(left) &&
13          associated(left, *) {
14          // notify the right bit of change
15      }
16
17      after(Bit right) : call(void Bit.set()) &&
18          target(right) &&
19          associated(*, right) {
20          // notify the left bit of change
21      }
22  }
```

Listing 2-2 – `Equality` aspect in Association Aspects

2.2 Per-object instantiation

In contrast to the association aspect instantiation policy, the per-object instantiation policy associates an aspect instance to only one object. The policy is supported by the mainstream PA-languages. In AspectJ this policy is subdivided in the per-this instantiation policy and the per-target instantiation policy. The difference between the policies is that the former one selects the caller object when a join point is matched by a pointcut, whereas the latter on selects the callee object.

The per-object aspect-instantiation policy manages the rules according to *instance storage and look-up*, *instantiation*, and *restriction* as follows:

The *instance storage and look-up* mechanism: The aspect instance is stored in a field of the object that is involved at the join point matched by a pointcut. The instance can be lookup by referring to the field.

The *instantiation* mechanism: The per-object instantiation policy follows an implicit instantiation strategy. This means that the aspect instance is created and associated the object automatically.

The *restriction* mechanism: There is no restriction for the per-object instantiation policy because aspect instances are created automatically on demand.

2.3 Singleton aspects

The singleton instantiation policy is useful when an aspect instance does not have to be associated to any object. Therefore only one aspect instance exist throughout the execution of a program.

The singleton aspect-instantiation policy manages the rules according to *instance storage and look-up*, *instantiation*, and *restriction* as follows:

The *instance storage and look-up* mechanism: Because only one aspect instance can exist it does not have to be *stored* in a field of an object. The same applies for *look-up*. The look-up function returns the aspect instance without depending on an object.

The *instantiation* mechanism: The per-object instantiation policy follows an implicit instantiation strategy. This means that the aspect instance is created automatically.

The *restriction* mechanism: There is no restriction for the per-object instantiation policy because aspect instances are created automatically on demand.

2.4 Generalized Model

Based on the three discussed instantiation policies, a generalized model is defined in [7] that models the three discussed instantiation policies. Hereby the association aspect policy is taken as the base instantiation policy. The association aspect policy, however, has support for explicit instantiation only. Since the model must also cover the per-object and singleton instantiation policies, support for implicit instantiation is added. Thus, the generalized model is in fact: association aspects and optional support for implicit instantiation.

Next, we discuss how the three aspect-instantiation policies are mapped to the generalized model. We characterize each policy in the generalized model through the following four properties:

1. *The arity of the key tuple.*
   In general an aspect-instantiation policy establishes a relation between an n-tuple of objects to an aspect instance. The n-tuple of objects is defined as the key-tuple. The arity of the key-tuple is specified by the length of the key-tuple.

2. *The context values exposed.*
   The aspect-instantiation policy determines which context-values are exposed and are bound to the key-tuple. The number of context-values exposed equals the arity of the key-tuple.

3. *Explicit or implicit association strategy.*
   The aspect-instantiation policy follows either an explicit or implicit strategy. If an implicit strategy is used then aspect instances are created automatically according to the rules of the policy. In case of an explicit instantiation strategy, the aspect instance is manually created together with a definition when the aspect instance may be used.

4. *Wildcards.*
   The aspect-instantiation strategy can be configured to use wildcards for the key-tuple. If wildcards are used then the key-tuple can result into more than one aspect instantiation.

Association Aspect Instantiation Policy:
The association aspect-instantiation policy directly maps to the model without any adoptions or constraints because the model is based on this policy. The following properties apply for the association aspects instantiation policy:

Arity of key-tuple:	Any size
Context exposition:	Any context
Instantiation strategy:	Explicit
Wildcards:	Any wildcards

Per-Object Instantiation Policy:
The per-object instantiation policy is mapped to the model with the support for implicit instantiation and is characterized as follows:

Arity of key-tuple:	1
Context exposition:	Caller or Callee
Instantiation strategy:	Implicit
Wildcards:	Not available

Singleton Instantiation Policy:
The singleton instantiation policy is mapped to the model with the support for implicit instantiation and is characterized as follows:

Arity of key-tuple:	0
Context exposition:	Not available
Instantiation strategy:	Implicit
Wildcards:	Not available

2.5 Support for Multi Values & the `StoreHandle` entity

The generalized model assumes that at most one aspect instance is associated to key-tuple. The association relation is considered to be a function. The function is called the instance storage function F_S of a given aspect:

$$F_S : K \rightarrow I_A$$

where K is the key-tuple and I_A an aspect instance.

The storage function of the generalized model does not allow that K results into multiple I_A values. It is thus a mathematical function whereby the input value is associated with exactly one output value.

We would like that the storage function can support so-called multi values. This means that K results in multiple I_A values. The next small example shows why the need for multi values is feasible: If we would like to associate one key-tuple with two aspect instances (for instance, a security aspect and log aspect), then we are forced two deploy two units of dispatch[2], with each unit having its own storage function. The fact that two units of dispatch have to be deployed does not contribute to a faster execution of the overall program. The key-tuple has to be given to the storage function twice which in turn performs a search twice.

We have therefore changed the storage function into a storage relation. The storage relation allows that one K can result into multiple I_A values.

[2] A unit of dispatch is explained in the next chapter.

Each association that is established must be stored at a location that is known to the *instance storage and look-up* mechanisms. The `StoreHandle` entity is defined as the location where associations are stored. One of the first contributions of this work is to have support for multi values and as a consequence we adapted the `StoreHandle` entity accordingly. The `StoreHandle` entity is shown in listing 2.3 and its most two important methods are the `add` and `getMatched` method to which will be referred in the remainder of this thesis. The `add` method establishes an association between a key-tuple and aspect instance. The `getMatched` method is used to retrieve an association.

```
1   public class StoreHandle {
2
3       private  MultivalueMap map = new MultivalueMap();
4
5       public Object[] getMatched(Object[] tuple) {
6               // return associated aspect instances
7       }
8
9       public void add(Tuple key, Object value) {
10              // associate the key (tuple) with value (aspect instance)
11      }
12  }
```

Listing 2-3 – The `StoreHandle` entity

Chapter 3

Background ALIA4J

This chapter discusses the technical background on ALIA4J. We begin with a global overview of ALIA4J in section 3.1. Next, its main components LIAM and FIAL are discussed in section 3.2 and 3.3 respectively. Finally section 3.4 discusses Steamloom^{ALIA} that is one of ALIA4J's execution environments.

3.1 ALIA4J

In the introduction we discussed that the solution is going to be implemented in ALIA4J. The ALIA4J approach provides a foundation for implementing programming languages with advanced dispatching concepts, e.g. aspect-oriented languages, in a way that the implementation of overlapping concepts can be shared [**9**]. With ALIA4J, thus, many advanced dispatching concepts of many languages are implemented in one framework. The interesting dispatch concept for this thesis is pointcut-and-advice, found in aspect-oriented languages. A global overview of the architecture of ALIA4J is given in figure 3.1 whereby two components are depicted with a grey background. These components are the main components of ALIA4J. The left component is the Language-Independent Advanced-dispatching Meta-model (LIAM) that acts as intermediate representation for advanced dispatch declarations, e.g. pointcut-and-advice declarations. Code of the program that is not affected by advanced dispatch is represented as conventional Java bytecode. Then at runtime both the program's model of dispatching declarations and bytecode is passed to one of ALIA4J's execution environments that either interprets or compiles it to final machine code. The execution environment contains a concrete FIAL instantiation that derives an execution model for each dispatch declaration that is executed by the execution environment. The Framework for Implementing Advanced-dispatching Languages (FIAL) is the right component in figure 3.1. A brief overview of the approach can be found in [**2**].

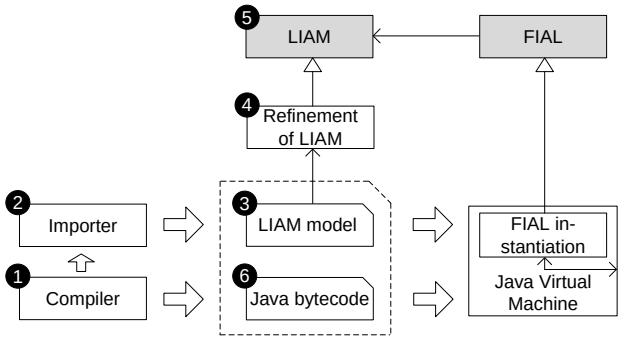


Figure 3-1 - Overview of the application life cycle in ALIA4J-based language implementations

Besides that the components of ALIA4J are shown in figure 3.1, the compilation and execution flow is shown through the use of numbers. The flow starts start by the compiler (**1**) that processes the application's source code. Next, a dedicated importer (**2**) adapts the compiler's output to a model for the advanced dispatch declarations in the program (**3**) based on the refined subclasses (**4**) of the LIAM meta-entities (**5**). Furthermore, the compiler produces an intermediate representation of those parts of the program that are expressible in the base language alone (**6**). Then at runtime: The generated LIAM model containing the advanced dispatch declaration plus Java bytecode not using advanced dispatching is called the program specific intermediate representation and is marked with a dotted square. The program-specific intermediate representation is passed to the FIAL instantiation and subsequently handled by the FIAL framework itself. The FIAL instantiation is embedded into execution environment, i.e., a Java virtual machine. The execution environment can be an extension of a virtual machine or a plug-in of an existing virtual machine. In either case, the FIAL framework generates an execution model for each dispatch site in the application.

3.1.1 LIAM

LIAM defines categories of concepts relevant for dispatching and how these concepts can interact. It therefore uses a model that is shown in figure 3.2. The model consists of ten entities, discussed in detail elsewhere [**8**], that capture the core concepts underlying various dispatch mechanisms. Each entity represents a dispatch concept found in high-level languages but at a finer granularity; one concrete dispatch concept often maps to a combinations of LIAM entities. The ten entities are plain Java classes and can be redefined in case a new dispatch concept is modeled.

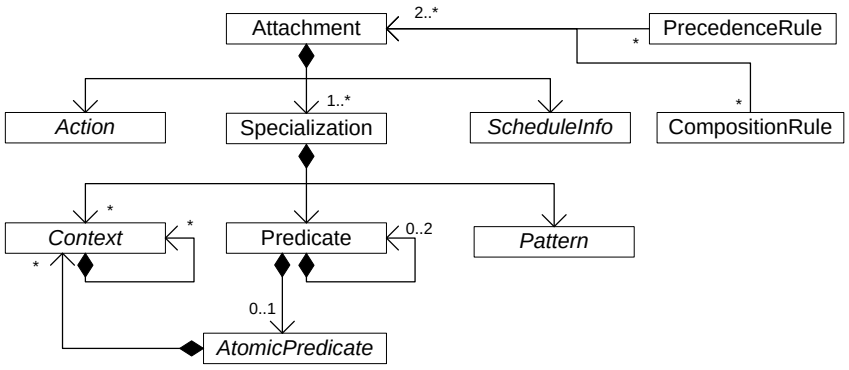


Figure 3-2 – The ten entities of LIAM

An `Attachment` entity corresponds to a unit of dispatch. In terms of a PA-language, it roughly corresponds to a pointcut-and-advice pair. An `Action` entity specifies the functionality that may be executed as the result of dispatch. In terms of a PA-language, it corresponds to the body of an advice. A `Specialization` entity defines static and dynamic properties on which dispatch depends. Static properties are available at compile-time and are defined by `Pattern` entities. A pattern specifies syntactic properties of call sites which are affected by the declared dispatch. Dynamic properties are available at runtime and are defined by `Predicate` entities and `AtomicPredicate` entities. A predicate is a boolean expression of atomic predicate, modeling

dynamic properties where a dispatch depends on. In terms of a PA-language, static properties correspond to static pointcut designators and dynamic properties correspond to dynamic pointcut designators. A `Context` entity models access to values in the context of a dispatch and exposes these as a list of values to an action. Examples of context values are: the calling object (caller), the receiving object (callee) or argument values. The `ScheduleInfo` entity models when functionality defined by the action should execute relatively to the dispatch site. In terms of a PA-language this corresponds to before, after or around. Furthermore there are two entities that model the relation between attachments. The `PrecedenceRule` entity defines the order of execution among actions when jointly applicable at a dispatch site. The `CompositionRule` entity specifies which actions must or must not be executed together.

In the introduction we mentioned that a new LIAM entity is going to be developed that implements semantics of aspect-instantiation strategies at a low-level of abstraction. The LIAM model shown here is therefore going to be extended as follows: we extend an existing `Context` entity that currently implements semantics of aspect-instantiation strategies at a high-level of abstractions. In chapter four and five this `Context` entity is discussed in more detail and is explained why it needs to be extended.

3.1.2 FIAL

FIAL defines several common components required by any execution environment for advanced-dispatching languages using a LIAM-based intermediate representation. As shown in figure 3.1, the *intermediate representation at (3) and (6)*, is passed to the FIAL instantiation to be transformed to an execution model. By transforming advanced dispatching declarations, FIAL generates an execution model for each dispatch site in the program.

Once the execution model is produced it is ready to be processed by either an interpreter of compiler. This depends on whether an interpreter-based execution environment is or a compilation-based execution environment is used. Currently there exist several execution environments; interpreter-based are for instance NOIRIn and compilation-based are SiRIn and Steamloom^{ALIA}.

In this thesis the Steamloom^{ALIA} virtual machine is used as execution environment. The Steamloom^{ALIA} virtual is based on the JikesRVM [**10**] which is an open-source research virtual machine and allows us to research sophisticated optimizations for the new `Context` entity. We mentioned earlier in this thesis that the new `Context` entity is going to generate optimized machine code that is based according to semantics of aspect-instantiation policies. Steamloom^{ALIA} offers support for machine code generation for LIAM entities and can be adapted easily according to our optimizations.

The next section gives a brief introduction on the basic components of Steamloom^{ALIA}. When in future chapters a component reappears and requires more explanation, we give it in the form of an intermezzo.

3.2 STEAMLOOM^{ALIA}

Steamloom^{ALIA} [**11**] is a FIAL-based execution environment built on the Jikes RVM [**10**]. The Jikes RVM is a high performance, open source, research virtual machine that is able to run a Java program. The virtual machine itself is also written in Java, a style of implementation that is termed meta-circular. The RVM can be divided into the following components:

- Core Runtime Services: (object model, thread scheduler, class loader, library support, verifier, dynamic linking, exception management etc.) This component is responsible for managing all the underlying data structures required to execute applications and interfacing with libraries.
- Magic: To support low-level system programming in Java.
- Compilers: (baseline, optimizing). This component is responsible for generating executable code from bytecode.
- Garbage collector - MMTk: This component is responsible for the allocation and de-allocation of objects during the executing of an application.
- Adaptive Optimization System: This component is responsible for profiling and executing application and judiciously using the optimizing compiler to improve its performance.

The listed components show that the Steamloom^{ALIA} virtual machine contains the components that are also present in regular virtual machines. A major exception is, however, that all components are written in Java whereas components used in regular virtual machine are often written in a low-language like C or C++. The fact that components are written in Java allows us to seamlessly integrate the optimization strategies in Java self. The components that play a key role in this work are: the *baseline compiler*, *the memory manager* and the *magic part*. The next sections give a brief introduction on these subsystems respectively.

3.3.1 Baseline compiler

The JikesRVM contains two compilers, the baseline compiler and the optimizing compiler. Both compilers can be used to compile bytecode to final machine code. The baseline compiler is used as an initial compiler to compile bytecode to final machine code. The baseline compiler is a relatively fast compiler and produces relatively simple stack-based machine code.

When the compiled code is executing for a while, Jikes can decide to enable the adaptive optimization system. This system identifies regions of code in which the application spends significant execution time. When the execution time passes a certain threshold it can decide to recompile the “slow” machine code with the optimizing compiler. The optimizing compiler is a relatively slow compiler but produces register-based code. This register-based code executes quicker than stack-based code. The adaptive optimization system only recompiles “slow” code if calculations indicate that the overall execution time can be improved. Because recompilation occurs at runtime it must make these calculations with great care. In case of a miscalculation the recompilation can become the slowest factor and overall execution time degrades.

Despite that the optimizing compiler produces quicker code than the baseline compiler we restrict this thesis to the baseline compiler for the following reasons: The optimizations are built from scratch therefore some time was needed to invest these optimizations. The optimizing compiler is

more complicated than the baseline compiler and again to time limitations it was not possible to map the optimizations to the optimizing compiler.

3.3.2 Garbage collector - MMTk

The Memory Management Toolkit (MMTk) handles all memory-related operations within the JikesRVM. One responsibility of the MMTk is the division of the heap into various spaces. Every space has certain characteristics that determine how objects are allocated and how they are garbage collected. In one of the optimization strategies we use a particular heap space for optimization purposes.

3.3.3 Magic

Most virtual machine components rely on services of the underlying system, such as operating system calls, raw machine memory and registers to implement runtime behavior. An example of runtime behavior is the creation of an object. When an object is created, the virtual machine has to allocate a piece of memory in order to store the object on the heap. Since most virtual machines are written in native code (typical C, C++), they call the `malloc` function to allocate a block of memory on the heap.

For the Jikes RVM the allocation of an object is handled different. Because the JikesRVM is implemented in Java, it not possible to address system services because Java is designed to be a platform-independent language. As such, platform-dependent operations are forbidden. However, we stated that the JikesRVM can be seen as regular virtual machine that can execute any Java program.

In order to justify this statement, the JikesRVM implements the Magic [**12**] component. The Magic component provides an escape hatch to implement system services. All system services and low-level operations that are inexpressible in Java are implemented as empty methods in the Magic class. When a method in the JikesRVM needs a system service, like for example object allocation, it can call the empty method. As soon as the method is compiled to machine code, the compiler filters all calls to the empty Magic methods and substitutes the call with machine code that performs the desired system service.

In chapter five where the implementation is discussed, optimization strategies will make use of the Magic class. The optimizations strategies make use of constant object-references, i.e. constant pointers. In Java this would be impossible to express but the Magic component allows such a concept.

Chapter 4

Problem Analysis and Approach

In the previous chapters we discussed the generalized model, the ALIA4J framework, LIAM and FIAL, and the Steamloom^{ALIA} execution environment. In this chapter we analyze and elaborate on the problem. We illustrate via an example how a LIAM entity representing the semantics of the generalized model currently is implemented within ALIA4J. Then we analyze the presence of the LIAM entity in the execution model and how the JIT-compiler generates machine code instructions that will execute one the of the aspect instantiation strategies defined in the generalized model.

4.1 Problem analysis

In the introduction we made clear that there is a semantic gap between source and intermediate language. The semantic gap that is relevant for this thesis is that the semantics of the generalized model cannot directly be expressed as Java bytecode instructions; bytecode instructions do not contain instructions that specifically describe the workings and constraints of the generalized model.

We said that, with the ALIA4J framework, this semantic gap can be closed by *preserving* the semantics of the generalized model. Hereby an entity of the meta-model (LIAM) acts as the intermediate representation that encodes the generalized model. When the generalized model is encoded in a LIAM entity, its semantics are preserved as we will see in the remainder of this chapter. ALIA4J enables LIAM entities the ability to implement their semantics at different levels of abstractions.

When the advanced dispatching declarations are transformed, FIAL generates an *execution model* for each join point shadow in the program. The execution model contains several LIAM entities that realize advanced dispatching as explained in the previous chapter. One of the LIAM entities present in the execution model is the LIAM entity that contains the semantics of the generalized model. Recall that this entity is responsible for delivering the aspect instances that serve as the context for executing advice. In order to incorporate the semantics of the LIAM entity, Steamloom^{ALIA} can generate the corresponding machine code in two different ways. This depends on whether a LIAM entity has implemented semantics at high-level of abstraction or a low-level of abstraction. The next two sections discuss the two approaches.

In the current version of ALIA4J there is a LIAM entity that implements the semantics of the generalized model at a high-level of abstraction. The term, high-level of abstraction, means that a LIAM entity implements a *plain* Java method realizing semantics in terms of interpretation. In order to execute semantics of the LIAM entity, i.e. execute the aspect instantiation strategy, the JIT-compiler compiles the plain Java method straightforward to machine code. Then when the JIT-compiler compiles the execution model –with the LIAM entity object –it generates machine

code instruction that will call the compiled method of the LIAM entity object. The execution of the method results in the execution of the aspect instantiation strategy that will deliver the proper aspect instances.

The same LIAM entity can also implement semantics at a much lower level. In order to realize this, a so-called *generator* method is implemented. Opposed to the plain Java method, this generator method has the ability to generate machine code instructions that represent semantics. When the execution model is compiled by the JIT-compiler, it calls the generator method of the LIAM entity that generates machine instructions that will execute the aspect instantiation strategy. Because the generator method is a method of the LIAM entity, it can take semantics of the generalized model into consideration during machine code generation. How we take the semantics into consideration and what benefits it brings, becomes clear in chapter five.

The difference between the two styles is that the high-level style is intended for language designers. They can easily and relatively quickly refine and develop LIAM entities. It is considered to be easy because ideas and thoughts can be expresses as a Java solution. As opposed to the high-level style, language implementers work on the low-level style. They work on machine code level where previously designed solutions can be optimized. The optimizations that are performed at this level are the most effective optimizations because machine code is less restricted than Java source code.

4.1.1 The Current `PerTupleContext` in ALIA4J.

The current version of ALIA4J contains a LIAM entity that implements the semantics of the generalized model in a high-level of abstraction. This LIAM entity is known as the `PerTupleContext` class. The next example shows how the `PerTupleContext` class is used within a dispatch site and how the JIT-compiler compiles the site to machine code. The example shows in a detailed way how the high-level approach works. After the example is discussed, we present the problem statement.

4.1.2 An Example of the `PerTupleContext`.

In the example we would like to advise all calls to `Foo.bar()`. For every distinct caller object, we create an aspect instance that serves as the context for executing the advice, i.e., in the example we apply the per-this instantiation strategy. The advice is executed after the `Foo.bar()` join point. We give two implementations of the example, one in AspectJ and one in ALIA4J. The purpose of implementing the example twice is to show how the ALIA4J implementation corresponds to the AspectJ implementation. Hereby it is assumed that sufficient knowledge about AspectJ is known.

```
1   public aspect Example perthis(pointcut1()) {
2       . . .
3       Logger logger = new Logger();
4       . . .
5       . . .
6       pointcut pointcut1():
7           call(public void Foo.bar());
8
9       after(): pointcut1() {
10          logger.log();
11      }
12  }
```

Listing 4-1 – The `Example` aspect in AspectJ

For AspectJ the example is given in listing 4.1. We defined an `Example` aspect with the `perthis` modifier to set the correct instantiation strategy. Furthermore we defined a pointcut named `pointcut1` (line 6-7) that captures method calls to `Foo.bar()`. The pointcut is associated with advice (line 9-10) that runs afterwards. The advice includes a trivial log action, i.e., a `log` method is invoked on a `logger` object which if of type `Logger`. The `logger` object is an instance variable of the `Example` aspect and is created when the aspect is created.

For ALIA4J, we define and deploy an `Attachment` that is configured as shown in the object diagram of figure 4.1 The `Attachment` object is composed of an `Action` object, a `Specialization` object and a `Schedule` object. The action defines that a static method `log(Object[] obj)` of the `Example` class is called as result of dispatch. The `obj` parameter is an array with aspect instances. The `log` method thus receives an object-array of aspect instances. In order to execute the advice within the context of an aspect instance, the `log` method must loop through the array whereby each aspect instance is used as context.

The `schedule` object defines that the action executes after the join point. The `specialization` object defines static and dynamic properties and defines when and how call sites are advised. The static property is defined in the terms of a `MethodPattern` object specifying that calls to `Foo.bar()` are advised. The dynamic properties are the `CallerContext` object and the `PerTupleContext` object that provide access to values in the context of a call to `Foo.bar()`. The `CallerContext` object provides the caller object that is used to form the key-tuple and is given to a `PerTupleContext` object which in turn creates or retrieves the aspect instances.

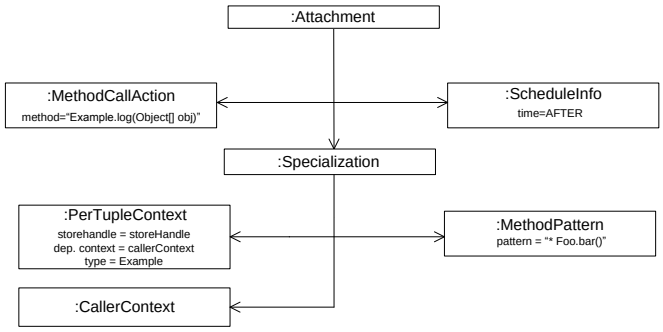


Figure 4-1 – Object diagram of the attachment in ALIA4J

The way of how advices are executed differs between the implementations and is schematically shown in figure 4.2. The left-hand side of the figure shows the AspectJ way and illustrates how the `Example` aspect is compiled to a class (`Example_AJ`) and in particular how the advice declaration is transformed to an instance method. The `afterAdvice` instance method runs directly within the context of the aspect instance and can access instance variables as shown.

The right-hand side shows the ALIA4J way and works slightly different than AspectJ. First of all the class that represents the aspect (`Example_ALIA4J`) is directly declared instead of begin compiled. Second, the `PerTupleContext` does not deliver a single aspect where the `afterAdvice` method can be executed upon, but delivers an array of aspect instances. In order to execute the `afterAdvice` method within the context of an aspect instance the array needs to be processed. This is done by the static `log` method which takes as argument an array. The `log` method iterates through the array and on each aspect instance the `afterAdvice` method is called. As shown in the figure both `afterAdvice` methods are equal and this makes that in the end both implementations have the same advice behavior.

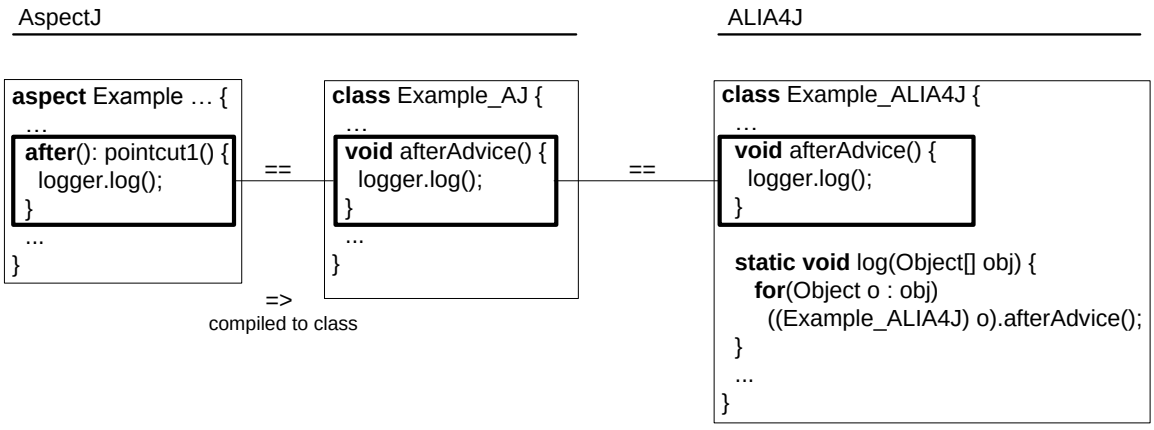


Figure 4-2 – On the left-hand side: The AspectJ approach. On the right-hand side: The ALIA4J approach.

We explained how the example is implemented in AspectJ as well as in ALIA4J. The next part of the example continues with the ALIA4J implementation. We show how the dispatch site is built and how the `PerTupleContext` object is used within the execution model in order to execute the aspect instantiation strategy that retrieves the aspect instances.

Once the attachment of figure 4.1 is deployed, the method `callsite` is called.

```
1   class CallSiteClass {
2
3       private Foo foo = new Foo();
4
5       public void callsite() {
6
7           foo.bar();
8
9       }
10  }
```

Assume that the method is called for the first time. As a consequence of the call, the bytecode of method `callsite` is processed and compiled. During processing of the bytecode, every join point shadow that matches the `MethodPattern` object is rewritten. The next intermezzo explains this:

INTERMEZZO I

In a normal way –without ALIA4J or AspectJ–, the source code of the callsite would have been compiled to following bytecode:

```
1   aload_0
2   getfield < CallSiteClass, foo, Foo >
3   invokevirtual < Foo, bar, ()V >
4   return
```

However, since the *invokevirtual < Foo, bar, ()V >* join point shadow matches the criterion of the `MethodPattern` object, it needs to be advised. Steamloom[ALIA] therefore rewrites the join point shadow as follows:

```
1   aload_0
2   getfield < CallSiteClass, foo, Foo >
3   invokestatic  < JoinPointSites, callsite$2, (Foo)V >
4   Return
```

The original join point shadow is replaced by a call to a static method named `callsite$2` which is defined in a `JoinPointSites` class. For simplicity we call the `callsite$2` method just JoinPointSite. The JoinPointSite is automatically generated by Steamloom[ALIA], hence the strange name, and takes as argument the receiver object of the join point and has `void` as result value. The JoinPointSite is a wrapper method because it wraps the original join point shadow. Within in the JoinPointSite the join point will be executed and Steamloom[ALIA] will add advice instructions after the join point shadow. The join point shadow is wrapped because it can be relatively easy adapted in this way. For the example this means that advice instruction can be easily added after the join point shadow.

In the last section we explain the `JoinPointSite` and show how the `PerTupleContext` object is used to regulate aspect instantiation and/or aspect selection. Listing 4.2 shows the `JoinPointSite` that is automatically created by Steamloom[ALIA].

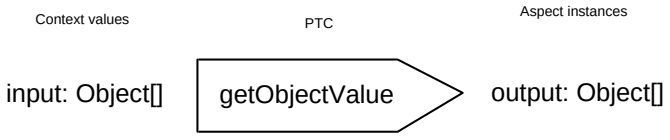
The first two instructions represent the original join point shadow. The rest of the instructions are responsible for executing the aspect-instantiation policy and advice.

Instruction 3 retrieves the `PerTupleContext` object and pushes it onto the stack. It corresponds to the `PerTupleContext` of the object diagram in figure 4.1. The `PerTupleContext` object is the receiver object for the method that is called on line 11.

Instructions 4 – 10 create the key-tuple that is pushed onto the stack. The key-tuple for this example contains the caller object because a per-this instantiation strategy was defined.

In the beginning of this chapter we discussed that a LIAM entity can implement semantics of dispatching mechanisms at different levels of abstractions. The existing `PerTupleContext` entity uses a high-level of abstraction style; semantics are thus implemented in a plain Java method. This plain Java method is called the `getObjectValue` method. The eleventh instruction invokes the `getObjectValue` method. Hereby the receiver object is the `PerTupleContext` object and the key-tuple is passed as argument.

The signature of the `getObjectValue` shows that it requires an object-array as input and outputs an object-array as well. The signature corresponds to the workings of the generalized model. Hereby a key-tuple (the input object-array) is provided to the model's most general instantiation strategy, i.e. the association aspects strategy, which then returns the associated aspects (the output object-array). Schematically:



Instruction 12 passes the object-array that contains the associated aspect instances to a method that then executes the advice.

1 2	aload 0 invokevirtual <Foo, bar, ()V >	}	*wrapped join point*
3	retrieve ${perTuple[, resultType=<Logger>, boundElements size=1, wildcardPositions size=0]}	}	*Retrieves the* `PerTupleContext`$^{NO\text{-}OP}$
4 5	sipush 1 anewarray new < [Object >	}	*array creation to hold the key-tuple*
6	dup		
7	sipush 0		
8 9	retrieve caller invokevirtual < CallerContext;, getObjectValue, ()Object >	}	*retrieves the caller.*
10	aastore		*stores caller in array*
11	invokevirtual <PerTupleContext;, getObjectValue, (Object) Object>		*retrieve aspect instance*
12	invokestatic <Logger, log, ([Object)V >		*execute advice*
13	return		

Listing 4-2 – The `JoinPointSite` method.

4.1.3 Problem statement

The bytecode of the `JoinPointSite` is straightforwardly compiled to machine code by the JIT-compiler. Because Steamloom^{ALIA} knows that the `PerTupleContext` implements semantics at a high-level of abstraction, it generates instructions that call the `getObjectValue` method of the `PerTupleContext` object.

The problem that causes this thesis is as follows: When the JIT-compiler compiles the `getObjectValue` method to machine code, it does not take semantics of the generalized model into consideration. This is because the JIT-compiler simply does not understand semantics; it only understands bytecode instructions.

```
1   public final Object getObjectValue(final Object input) {
2       final Object[] tuple = (Object[]) input;
3       Object[] result = this.storeHandle.getMatched(tuple, this.wildcardPositions);
4       if (result.length == 0 && this.resultType != null) {
5           assert this.wildcardPositions.length == 0;
6           Class<?> clazz;
7           try {   // reflection – instantiate aspect instance
8               clazz = this.resultType.asClass();
9               result = new Object[] { clazz.newInstance() };
10          }
11          catch (Exception) {…}
12          this.storeHandle.add(new Tuple<Object>(tuple), result[0]);
13      }
14      return result;
15  }
```

Listing 4-3 – The `getObjectValue` method

In order to illustrate what we exactly mean with "not understanding semantics", we show an example of how the `getObjectValue` method is implemented and how it is compiled to machine code.

Listing 4.3 shows a part of the implementation of the `getObjectValue` method. We explained that the current `PerTupleContext` class implements semantics of the generalized model at a high level of abstraction via the `getObjectValue` method. When this method is compiled by the JIT-compiler, it generates corresponding machine code. Once compiled, it can be called from every `JoinPointSite` method. The implementation of the method takes care that it can handle the all the aspect-instantiation policies that are defined by the generalized model. Likewise, the generated machine code can handle these aspect-instantiation policies. When a `JoinPointSite` is compiled, it retrieves the `PerTupleContext` (instruction 3 of listing 4.2) object. This object is configured with a certain aspect instantiation policy. Because the JIT-compiler does not know about this configuration, it cannot create the direct machine code instructions that will execute the proper aspect instantiation strategy. Instead, it generates an instruction (instruction 11 of listing 4.2), that will execute the `getObjectValue` method. This method will determine at run-time, which aspect instantiation strategy must be executed.

The central problem is as follows: It can already be determined at compile-time which strategy is used, but the JIT-compiler cannot. Therefore it has to generate instructions that will call the `getObjectValue` method. The drawback of this approach is that everytime the `getObjectValue` method executes, it must determine the correct aspect instantiation strategy. This determination causes a lot of overhead, and therefore we consider this approach as slow. This slow approach is not in line with the statement that was made in the beginning of this chapter. We stated that aspect instance retrieval must execute fast because advice executes within the context of an aspect instance. As such, we do not want to have overhead.

The `getObjectValue` method is thus implemented in a general way in order to support all the possible configurations of the `PerTupleContext` object. For instance, the conditional expression on line 4 is evaluated everytime the method executes. This evaluation is unnecessary and causes overhead. We illustrate this with an example:

Figure 4.3 shows in the middle a program with two `JoinPointSites`. When the JIT-compiler compiles each `JoinPointSite`, it generates instructions that at run-time will call the `getObjectValue` method. The receiver object, i.e. `PerTupleContext` object, that is used for the call can differ per `JoinPointSite`. In the figure we have two different receiver objects. The first receiver object, `o1`, is configured with an implicit instantiation strategy[3]. Therefore, the conditional expression of the compiled[4] `getObjectValue` method at the right-hand side of figure 4.3 will at run-time evaluate to true. The second receiver object, `o2`, is configured with an explicit instantiation strategy. Therefore, the conditional expression will evaluate to false.

[3] If the resultType variable is set to a type, then an implicit instantiation strategy is used. If the variable is not set, then an explicit instantiation strategy is used.
[4] Consider the Java source code as machine code.

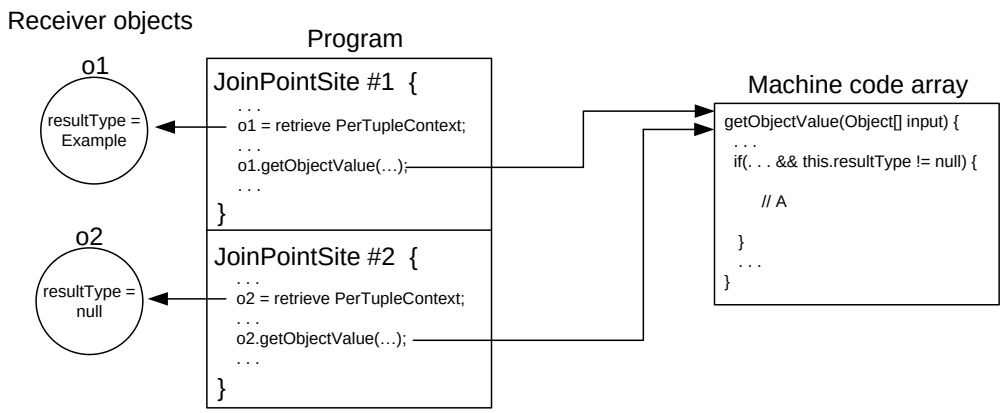


Figure 4-3 – The current way of executing a JoinPointSite when the PerTupleContext implements semantics at a high-level of abstraction

The problem in this example is that conditional expression of the compiled `getObjectValue` method is unnecessary. When the JIT-compiler compiles the first `JoinPointSite`, then it knows that the resultType is not null and that the conditional expression in the `getObjectValue` will evaluate to true.

In this section we have shown by one example why the current way of compiling a `JoinPointSite` and `getObjectValue` method is not the most efficient way. Chapter five discusses in detail more situations that cause unnecessary overhead. However, the main purpose of chapter five is the optimization of the `PerTupleContext` class. But before optimizations are discussed, we propose in the next section an approach of how we can apply such optimizations.

4.2 Approach

In order to take semantics of the generalized model into consideration we create a new `PerTupleContext` class. This new class is able to interface with the JIT-compiler, basically, the new `PerTupleContext` class becomes a part of the JIT-compiler. Because the `PerTupleContext` contains semantics of the generalized model, it can generate specific machine code. With specific we mean that we can tailor the machine code according to the optimization strategies that are described in the next chapter and thereby we have full access to the machine code generator; we are thus completely free of Java constraints.

To become a part of the JIT-compiler, we let the `PerTupleContext` class implement the `CompilerSupport` interface which is shown in listing 4.4. As such, the JIT-compiler can approach a `PerTupleContext` object as a `CompilerSupport` object and is then able to call the `generateASM` method. When this method is called, then the JIT-compiler hands over control of code generation to the `PerTupleContext` object.

```
public interface CompilerSupport {

    public void generateASM(Assembler asm, BaselineCompilerState compilerState);

}
```

Listing 4-4 – The `CompilerSupport` Interface

The fact that the `PerTupleContext` becomes a part of the JIT-compiler has a consequence on how the `JoinPointSite` is built. Namely as soon as the `PerTupleContext` generates its own specific machine code, then the call to `getObjectValue` is not needed anymore, likewise the receiver object can be discarded.

Listing 4.5 shows how the `JoinPointSite` is generated when the `PerTupleContext` implements the `CompilerSupport` interface. The call to `getObjectValue` and its receiver object have been omitted.

1 2	aload 0 invokevirtual <Foo, bar, ()V >	}	wrapped join point
4 5	sipush 1 anewarray new < [Object >	}	array creation to hold the key-tuple
6	dup		
7	sipush 0		
8 9	retrieve caller invokevirtual < CallerContext;, getObjectValue, ()Object >	}	retrieves the caller.
10	aastore		stores caller in array
12	invokestatic <Logger, log, ([Object)V >		execute advice
13	return		

Listing 4-5 – The JoinPointSite when the `PerTupleContext` implements the `CompilerSupport` interface.

The JIT-compiles now knows how to hand over control of code generation. The next part of the approach discusses where in the JoinPointSite this should be done.

The JIT-compiler of Steamloom^{ALIA} contains an extension point which allows `CompilerSupport` objects temporary take over control of code generation. When such a point is reached, the compiler calls the `generateASM` method to hand over control. Because the `JoinPointSite` does not contain a `getObjectValue` method call anymore, it seems at first glance that it is not possible to know where this point is. However, Steamloom^{ALIA} solves this by generating a marker in the `JoinPointSite`. The marker is invokestatic instruction that points to a dummy method. As soon as it encounters such a marker it calls the `generateASM` method. The dummy instruction looks as follows:

```
invokestatic <SupportedGetValue;, 2, (Ljava/lang/Object;)Ljava/lang/Object;>
```

It is called a dummy instruction because the method name "2" cannot and does not exist. However the number 2 is not totally meaningless. As said, in listing 4.4 the receiver object for the `getObjectValue` is omitted because it is not needed anymore within the `JoinPointSite`. However, the JIT-compiler needs this receiver object in order to call the `generateASM` method. Via the method name "2" it can retrieve the correct receiver object, by consulting a global registry that maps unique numbers to `PerTupleContext` objects.

In figure 4.4 we give an overview of the approach. As explained, we hand over control of code generation when the `invokestatic` dummy instruction is reached. Then the JIT-compiler gets the correct `PerTupleContext` object which is the receiver object for method call to `generateASM`. The `generateASM` produces tailored machine code according to the configuration of the `PerTupleContext` object.

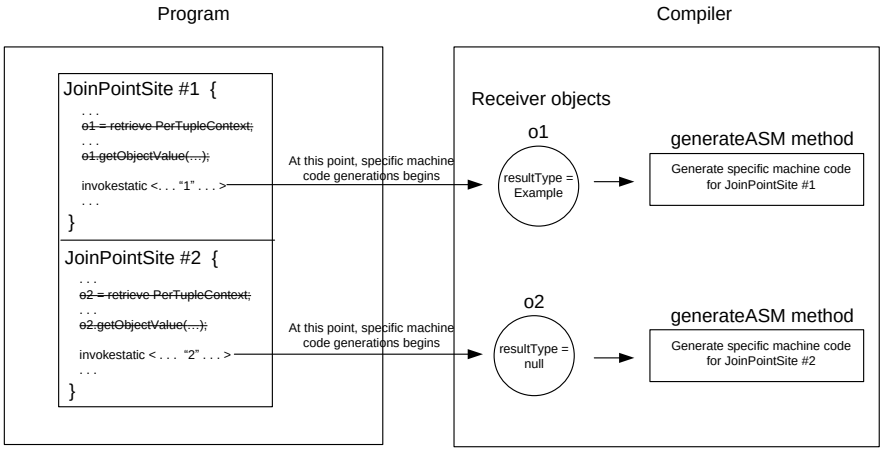


Figure 4-4 – The new approach of executing a JoinPointSite. The semantics of the aspect-instantiation policy correspond directly to the generated machine code.

Chapter 5

Implementation

In the second chapter we described the generalized model that covers the three most used aspect-instantiation policies. The fourth chapter showed how this generalized model is implemented in the `PerTupleContext`. Hereby semantics of the generalized model are implemented at a high level of abstraction. Next, the problem statement described that semantics cannot directly be understood by the JIT-compiler when an aspect-instantiation policy is compiled to final machine code. In this chapter we analyze the semantics of the `PerTupleContext` in order to derive an optimization strategy that is used by the `generateASM` method when generating machine code that represents the aspect-instantiation policy.

This chapter is organized as follows. Section 5.1 discusses how the semantics of the `PerTupleContext` are defined in order to derive eight so-called *optimizations strategies*. Then we discuss in section 5.2 each optimization strategy in detail and explain how the `generateASM` method generates for each optimization strategy the most efficient machine code.

5.1 Defining semantics of the PerTupleContext

This section analyzes semantics of the `PerTupleContext` in order to derive eight so-called *optimization strategies*. An optimization strategy is used by the `generateASM` method when the JIT-compiler hands over control of code generation. According to the optimization strategy, the most efficient machine code is generated. In order to determine which optimization strategy to use, the `PerTupleContext` object is first initialized. Prior to deployment of an attachment, the `PerTupleContext` is initialized by providing a number of arguments to the constructor which in turn configures a number of properties. For each configurable part a corresponding property is defined and based on the values these properties, an optimization strategy can be determined.

There exist two types of properties. The first type of properties is based on semantics of the generalized model and is called “derived properties”. The second type of properties is based on the added behavior of the `StoreHandle` entity and is named “additional properties”. After the properties have been discussed, the optimization strategies are explained.

5.1.1 Derived Properties

Property #1 – Explicit or Implicit Instantiation Strategy
The first property defines whether the `PerTupleContext` operates in an **implicit** instantiation mode or **explicit** instantiation mode. This property corresponds to the third characterization of the generalized model (see section 2.4) that defines whether the instantiation strategy is explicit or implicit.

Property #2 – Context Sensitive / Insensitive
The second property defines whether the `PerTupleContext` object is **sensitive** on context values or is **insensitive** on context values. This property corresponds to the first and second characterization of the generalized model. This first characterization defined the arity of the key-tuple. If the arity is zero then no context values are present in the key-tuple and the `PerTupleContext` is insensitive on context values. If the arity is greater than zero then the `PerTupleContext` is sensitive on context values. The third characterization defined which context values are exposed to form the key-tuple. However, for optimization purposes it does not matter which context values are going to be exposed because every context value is considered as an object.

Property #3 – Exact / Range Query
The third property defines whether a `PerTupleContext` object uses a **range query** or an **exact query** in order to get the associated aspect instances from the `StoreHandle` entity. This property corresponds to fourth characterization of the generalized model that defines if the key-tuple contains wildcards. If the key-tuple does contain wildcards then a range query is used to retrieve the associated aspect instance, otherwise an exact query is used.

5.1.2 Additional Properties

Property #4 – Fixed Associations
The fourth property defines whether the `StoreHandle` entity that is used by the `PerTupleContext` object allows associations to be added or not. Therefore this property is either set to **fixed associations** or **not-fixed associations**.

5.1.3 Overview of the combinations of properties

The legal combinations of values of the properties are shown in table 5.1. In total eight legal combinations can be derived whereby each combination is numbered.

Property 1 Implicit/Explicit	Property 2 Sensitive/ Insensitive	Property 3 Exact/Range	Property 4 Fixed Association		Number
Implicit Instantiation Strategy	Context Insensitive	Exact	False		1
Implicit Instantiation Strategy	Context Sensitive	Exact	False		2
Explicit Instantiation Strategy	Context Insensitive	Exact	True		3
Explicit Instantiation Strategy	Context Insensitive	Exact	False		4
Explicit Instantiation Strategy	Context Sensitive	Exact	True		5
Explicit Instantiation Strategy	Context Sensitive	Exact	False		6
Explicit Instantiation Strategy	Context Sensitive	Range	True		7
Explicit Instantiation Strategy	Context Sensitive	Range	False		8

Table 5-1 – The eight combinations of properties

Note that not all the combination are shown in table 5.1. This is because certain combinations are not valid. For example the combination: [*implicit instantiation strategy, context sensitive, exact query and fixed association*] is not a legal combination. The fixed property is only applicable when an explicit instantiation strategy is used because the client can decide that no more associations may be added to the `StoreHandle` entity. In case an implicit instantiation is used, association can always be added and thus the fixed property cannot be set to true. Another illegal combination is [*explicit instantiation strategy, context insensitive and range query and fixed associations*]. It is not possible to use wildcards when the arity of the key-tuple is zero.

Table 5.1 gave an overview of the optimization strategies whereby each strategy consists of four properties with each property set to a specific value. The next section discusses what each value of a property means to the `PerTupleContext` and what optimization decisions can be made.

5.2 The optimization strategies

This section discusses what each value of a property exactly means to the `PerTupleContext`. In the previous section we already discussed this, but at a global level. In section 5.2.1 each possible value of a property is discussed in more detail and provides what kind of optimizations can be made. From section 5.2.2 we define for each combination an optimization strategy, whereby the accent is thus on the conjunction of the four properties, rather than each individual property. Each optimization strategy can generate efficient machine code that only is valid for the combination corresponding combination. But before we begin, we explain what goes on before the generation of machine code starts.

The `PerTupleContext` can generate machine code because it implements the `CompilerSupport` interface (listing 5.1). As explained in the previous chapter, the `generateASM` method is called by the JIT-compiler when it encounters the dummy instruction.

```
1    public interface CompilerSupport {
2
3        public void generateASM(Assembler asm, BaselineCompilerState compilerState);
4
5    }
```

Listing 5-1 – The `CompilerSupport` interface

The `generateASM` method contains two arguments, `asm` and `compilerState`. The `asm` variable is of type `Assembler` and allows assembler instructions like `MOV`, `PUSH`, `POP`, `ADD`, `JMP`, etc to be generated. When such an assembler instruction is generated, the `generateASM` method directly manipulates the array that holds the binary machine code; we operate thus on the lowest abstraction level. The variable `compilerState` holds the state of the compiler. The state variable represents values like indexes and offsets of the machine code array. In the remaining sections it becomes clear how the `compilerState` is used for the generation of machine code.

As a last thing we mention some general facts that are applicable to all eight optimization strategies:

- The `generateASM` method is called by the JIT-compiler when it encounters the dummy instruction. From that point machine code is generated as was explained in section 4.2. When we refer to the `generateASM` method, we refer thus to the generation of machine code instruction for the `JoinPointSite`.

- Before the `generateASM` method is called, the machine code for the `JoinPointSite` is already partially generated by the JIT-compiler. The partially generated machine code is responsible for the creation of the key-tuple. When it executes, it has as result that a key-tuple is pushed onto the stack. Every optimization strategy must thus handle the key-tuple in a correct way.

- When a program is executed by Steamloom^{ALIA}, methods are compiled to machine code at run-time by the JIT-compiler. This means that at run-time, before a method can

execute, it has to be compiled to machine code. Therefore we sometimes explicitly mention if something is performed at compile-time or at run-time. With *compile-time* is meant that the JIT-compiler is currently generating machine code instructions. The generated machine code instructions are subsequently executed at *run-time*. With JIT-compilation the two times are thus intertwined, this is opposed to a not-JIT-compiler, where compile-time is before run-time and the two times are strictly separated.

During the discussion of the implementation of an optimization strategy, sometimes an excerpt of generated machine code or bytecode is shown. In order to understand the printed machine code or bytecode, we show for both cases an example. Every byte code instruction is shown as in figure 5.1. The first column, `[0]`, indicates the index of the bytecode instruction in the bytecode array of the method being compiled. The second column, `sipush 0`, gives the name of the instruction. In this example an instruction is used that needs an argument, i.e. the `0`, but there exist also instructions that need no arguments.

Meaning of the Java bytecode instructions:

```
[0] sipush 0
```

Figure 5-1 – Example of a Java bytecode instruction**.**

Figure 5.2 shows how machine code is printed. The first column, `000043`, indicates the index in the machine code array. The second column, `MOV`, is the name of the assembler instruction. The third column, `EDX`, shows the destination register and the fourth column `4[ESP]` shows the source register.

Meaning of the machine code instructions:

```
000043   |        MOV          EDX          4[ESP]    |
```

Figure 5-2 – Example of a machine code instruction.

5.2.1 Properties of the Optimization Strategies

Property #1 – Instantiation strategy

- *Implicit instantiation strategy*

  The `PerTupleContext` follows an implicit instantiation strategy and has does have control over aspect instantiation. The machine code that is generated will at run-time pose the following behavior: Either a new aspect instance is created and returned, if for the key-tuple no association exists (situation 1). Or an existing aspect instance is returned if for the key-tuple an association exists (situation 2).

  As soon as the key-tuple is associated with an aspect instance, the association is stored in the `StoreHandle` entity. After storage the key-tuple is thus associated with an aspect instance. This means that situation 1 occurs exactly once and from then on situation 2 always occurs.

- *Explicit instantiation strategy*

  The `PerTupleContext` follows an explicit instantiation strategy and this means that the creation of aspect instances, establishing and storing associations to `StoreHandle` entity is not done by the `PerTupleContext`. Instead an external entity, called the client, is responsible for performing these actions. The generated machine code has the following behavior when executed: Either no aspect instance is returned if for the key-tuple no association was established by the client. Or one or more aspect instances are returned if for the key-tuple one or more association were established by the client.

Property #2 – Context dependence

- *Context insensitive*

  Because the key-tuple is empty, the `PerTupleContext` can determine at compile-time if a key-tuple is associated with an aspect instance. Furthermore the `PerTupleContext` can query the `StoreHandle` entity without providing the key-tuple. The `StoreHandle` entity then directly knows that the key-tuple is empty which result into a faster search. The machine code instructions that are generated can therefore ignore and discard the empty key-tuple. Before the `generateASM` method is called, the machine code of the `JoinPointSite` method is already partially generated by the JIT-compiler. Listing 5.2 shows the machine code that is generated so far:

```
[0] sipush 0
000036 |        PUSH          0                    |
[3] anewarray new < [Ljava/lang/Object; >
000038 |        PUSH          44                   |
00003A |        PUSH          8255                 |
00003F |        MOV           EAX          8[ESP]  |
000043 |        MOV           EDX          4[ESP]  |
000047 |        CALL          [57081464]           |
00004D |        PUSH          EAX                  |
```

Listing 5-2 – The generated machine code for the first configuration before `generateASM` is called.

When the `[0] sipush 0` and the `[3] anewarray new` instructions execute they result into an empty array that is pushed onto the stack. Because the empty array is not used it can directly be discarded. Via the `asm` variable the `emitPOP` method is called that generates machine code instructions that at run-time will pop-off the empty array.

- *Context sensitive*

In this case, the key-tuple is not empty and the `PerTupleContext` cannot determine at compile-time if a key-tuple is associated with an aspect instance because the context values are not known. The `PerTupleContext` generates machine code that determines at run-time if for a key-tuple an association exists. Because this determination must happen fast, a caching mechanism will be used that caches previous fetched aspect instances.

Before the `generateASM` method is called, machine code instructions of the `JoinPointSite` method are already partially generated by the JIT-compiler. Because the aspect instantiation policy is context sensitive, the generated machine code is slightly different than as in the context insensitive case. Listing 5.3 shows the machine code that is generated so far:

```
[0] sipush 1
00003A|        PUSH          1                    |
[3] anewarray new < [Ljava/lang/Object; >
00003C|        PUSH          44                   |
00003E|        PUSH          8246                 |
000043|        MOV           EAX          8[ESP]  |
000047|        MOV           EDX          4[ESP]  |
00004B|        CALL          [57081464]           |
000051|        PUSH          EAX                  |
[6] dup
000052|        PUSH          [ESP]                |
[7] sipush 0
000055|        PUSH          0                    |
[10] aload 1
000057|        PUSH          12[ESP]              |
[12] aastore
…              …             …            …
…              …             …            …
…              …             …            …
```

Listing 5-3 – The generated machine code for the second configuration before `generateASM` is called.

When the `[0] sipush 1` and `[3] anewarray` instructions execute, they have as result that an object-array of size *n* is created and pushed onto the stack. The *n* in the code listing corresponds to one. This means thus that the aspect instantiation policy depends on one context value. If the `PerTupleContext` object would have been configured to depend upon two context values, the *n* would correspond to two. The next instructions are used to form the key-tuple by filling the object-array with the context value. The `[7] sipush 0` has a result that the array index is pushed onto the stack, the `[10] aload 1` has as result that the context value is pushed onto the stack. The exact context value is not important for this illustration although we mention it for understanding purposes. The `aload 1` instruction refers to the first local variable of the `JoinPointSite` method which in turn thus corresponds to the first argument. As explained in chapter four, the argument for the `JoinPointSite` method is the receiver object of the wrapped join point. The `[12] aastore` instruction stores the context value at the first position in the object-array. The `[6] dup` instruction is needed to duplicate the reference of the object-array since the `aastore` instruction consumes one reference when executed. Otherwise the object-array would become unreachable.

Property #3 – Query type

- *Exact query*

  The `PerTupleContext` generates machine code instructions that will query the `StoreHandle` entity with the complete key-tuple. The `StoreHandle` therefore knows that it can perform an exact query.

- *Range query*

  The `PerTupleContext` generates machine code instructions that will query the `StoreHandle` entity with the incomplete key-tuple and wildcards. The `StoreHandle` therefore knows that it can perform a range query.

Property #4 – Associations

- *Fixed associations*

  The `PerTupleContext` generates machine code instructions that will exactly once consult the `StoreHandle` entity and cache the result once. Because the `StoreHandle` does not change, the cached result is always valid and does not need to be invalidated.

- *Not-fixed associations*

  The `PerTupleContext` knows that the associations in the `StoreHandle` entity are not fixed. Therefore it generates machine code instructions will consult the `StoreHandle` entity only if the cached result is invalid.

5.2.2 Optimization Strategy 1

The machine code instructions that need to be generated realize the aspect-instantiation policy for this optimization strategy. Before the generation of machine code starts, the `PerTupleContext` determines at compile-time if for an empty key-tuple an association exists by querying the `StoreHandle` entity. If no association exists then the first situation is applicable and the following strategy is generated that at run-time will have the following behavior:

- Test if for the empty key-tuple an association exists (1);

  If no association exists then:
  - Create the aspect instance (2);
  - Establish and store the association (3);

  If an association exists then:
  - Retrieve the association (4);

- Return the associated aspect instances (5).

As soon as the aspect instance is created and the association is established and stored, the test instruction (1) prevents that instructions (2) and (3) will execute again.

On the other hand, if the `generateASM` method determines at compile-time that an association exists then it knows that the first situation has occurred and that only the second situation may occur. In that case machine code is generated that at run-time behaves as follows:

- Retrieve the association (4);
- Return the associated aspect instances (5).

The instructions for the second situation do not have to be protected by a test instruction because an association exist and can be retrieved from the `StoreHandle` entity. When compiling a `JoinPointSite` and the second situation is determined, then the test instruction does not have to be generated. This saves at run-time some overhead that otherwise would have been occurred when a test instruction was needed.

The remainder of this section discusses how the machine code for both situations is generated.

Instruction (2) cannot be executed at compile-time since the creation of aspect instances is deferred until the latest moment in time, i.e. lazy instantiation. Likewise instruction (3) cannot be executed at compile-time since there is no aspect instance to associate. The `generateASM` method generates thus machine code for the two instructions. At run-time an aspect will be instantiated and an association is established and stored. Instruction (1) will therefore evaluate at run-time exactly once to true and afterwards always evaluate to false.

For instruction (4) the `generateASM` method generates machine code that at run-time consults the `StoreHandle` entity that will fetch the association.

The `generateASM` method generates for instruction (5) machine code that will at run-time create an object-array and store the associated aspect instance. The object-array is then returned as result.

For first situation the `generateASM` method must thus generate machine code that determines at run-time if an association exists. If the second situation is applicable then no determination needs to be done at run-time. However, with respect to optimization we can make one major improvement. The remainder of this section is devoted on this improvement:

Once the aspect instance is created and the association is established and stored, the key-tuple always results into the same association even though the associations in the `StoreHandle` entity are not fixed. This means that instruction (4) will always retrieves the same association and that instruction (5) always returns an object-array of size one with the same aspect instance.

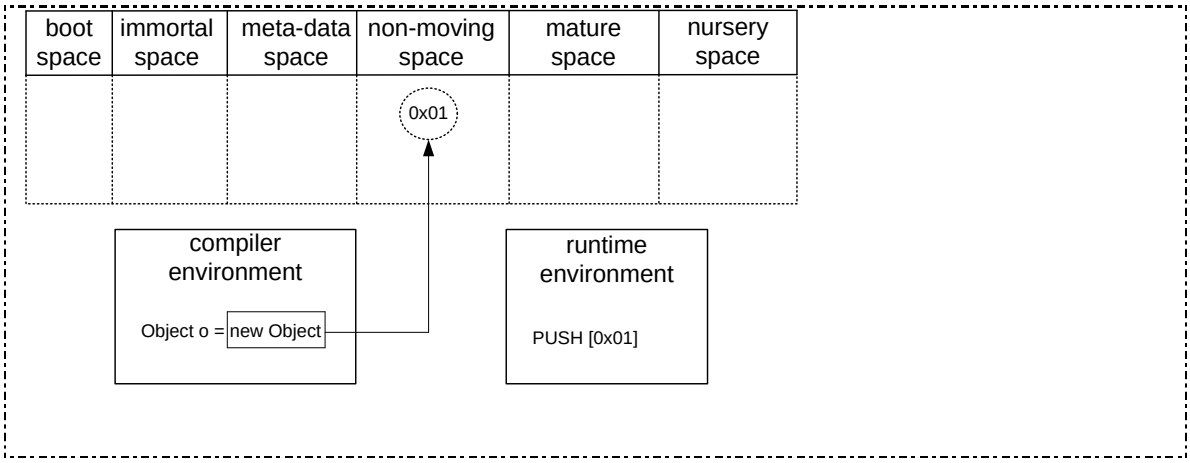
Therefore instruction (5) is executed at compile-time. The object-array is created at compile-time and has a size of one in order to hold exactly one aspect instance. This raises the following question: What is the benefit of creating an object-array at compile-time over run-time? At first glance it looks rather complicated because a compile-time object has to be transformed to a run-time object. The next intermezzo explains that this object-array is created in the so-called *non-moving* space and that there is no form of complication.

INTERMEZZO II

The JikesRVM has a module called the Memory Management Toolkit (MMTk). The MMTk handles all memory-related operations within the JikesRVM. One responsibility of the MMTk is the division of the heap into various spaces. Every space has certain characteristics that determine how objects are allocated and how they are garbage collected [**13**].

Each space is used for a different purpose. The *boot* space stores precompiled classes and data structures when the JikesRVM is built. The *immortal* space stores objects that live forever, and is not garbage collected after all. The *meta-data* space holds temporary memory management-related objects. The *non-moving* space stores objects that may never move, but can be garage collected when they are dead though. The *nursery* space is the space where all the application objects are allocated. If this space becomes exhausted then the garbage collector is signaled to run. The objects that survive garbage collection are moved to the mature space. When the mature space becomes exhausted then the garbage collector is informed to garbage collect this space too.

Objects that are created in the non-moving space will never move and thus always have the same memory address. An object that is created at compile-time will thus have the same address when that object is used at run-time. This allows the use of constant object references since the reference to object will always be the same. During code generation we can make use of constant object references and thus emit a constant memory address. The following picture show this:



Once the object-array is created in the non-moving space, the `generateASM` method is able to generate machine code that contains a constant reference to the object-array. This also eliminates the need to generate instruction (4); the aspect instance can directly be stored in the object-array and the `StoreHandle` entity does not need to be consulted anymore. As we will see in a moment, this optimization is particular interesting for the second situation because then machine code instructions are generated that only refer to the result.

The generated machine code for the first situation is shown in figure 5.3. The machine code is represented as pseudo-code because otherwise it would result into an abundance of machine code instructions which makes it hard to understand.

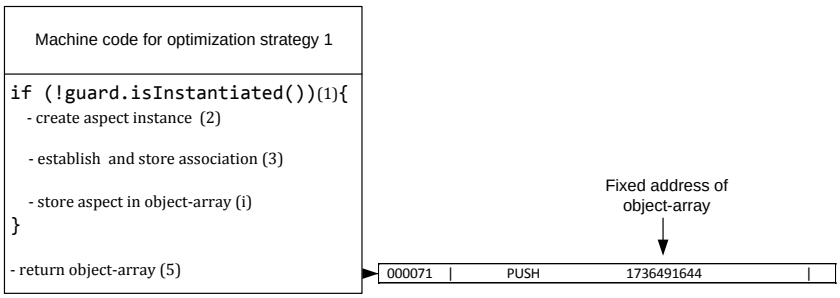


Figure 5-3 – Pseudo machine code instructions for the first optimization strategy

According to the properties of this configuration, exactly one aspect is instantiated and one association is established and stored. The machine code at location (2) and (3) correspond to instructions (2) and (3) respectively. For instruction (4) no machine code is generated because the object-array is used as explained. Instead at location (i) machine code is generated that stores the aspect instance in the non-moving array. Machine code at location (5) corresponds to instruction (5) and returns an object-array with the aspect instance. The object-array is returned by pushing its fixed address on the stack.

The generated machine code for the second situation is shown in figure 5.4. The object-array is returned by pushing its fixed address on the stack.

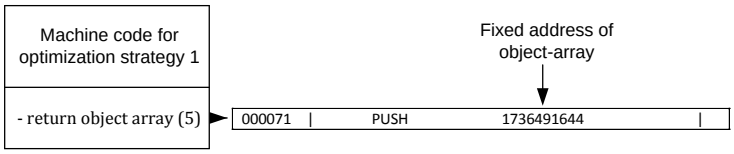


Figure 5-4 – Pseudo machine code instructions for the first optimization strategy

The generated machine code of either figure 5.3 or 5.4 executes according to the properties of this optimization strategy. This continues until Steamloom^{ALIA} *invalidates* a `JoinPointSite`. Invalidation of a `JoinPointSite` occurs when another attachment is deployed or undeployed. The next intermezzo exemplifies this:

INTERMEZZO III

Invalidation of JoinPointSites.

Consider the following method x and y.

```
public void method x() {
  deploySecurityAttachment();
  y();
  deployLogAttachment();
  y();
}
```

```
public void method y() {
  joinpoint();
}
```

Both the `SecurityAttachment` and `LogAttachment` advice the `joinpoint` with before advice, but they are deployed at different times. When method y is called for the first time, the `SecurityAttachment` was deployed. The machine code is generated as described and the advice executes accordingly. Because ALIA4J supports the dynamic deployment of attachments we can, for example, deploy the `LogAttachment` after method y has returned. The second time method y is called, two attachments have been deployed. The dynamic deployment of attachments results in an invalidation of the security JoinPointSite.

When the `JoinPointSite` gets invalidated it has to be recompiled by the JIT-compiler and likewise the `generateASM` method is called again. If such invalidation occurs, we can benefit from this mechanism because if a `JoinPointSite` contains machine code according to the first situation then it will be recompiled according to the second situation.

5.2.2 Optimization Strategy 2

The second optimization strategy is the same as the first strategy, except that it is sensitive on context values. The machine code that is generated must thus not discard the key-tuple but use it in order to determine if an association exists.

The machine code that is generated for the optimization strategy realizes the aspect-instantiation policy. In the first optimization strategy the `generateASM` method could determine at compile-time if for the empty key-tuple an association exists. Unfortunately for this optimization strategy the `generateASM` method cannot do this. Every time the `JoinPointSite` executes, the key-tuple can be different and no assumptions about the context value can be made at compile-time. Therefore the determination must be done at run-time. The `generateASM` method generates the flowing instructions that at run-time will execute as follows:

- Test if for the key-tuple an association exists (1);

  If this is not the case then:
  - Create the aspect instance (2);
  - Establish and store the association (3);

  If this is the case then:
  - Retrieve the association (4);

- Return the associated aspect instances (5).

These instructions are equivalent with the instructions that were described for the previous optimization strategy, but the machine code that is generated by the `generateASM` method is different.

In the previous optimization strategy an object-array could be created in the non-moving space and the generated machine code contained a constant reference to address the object-array. Because this strategy depends upon a context value that is not known until run-time, such a non-moving object-array cannot be used. This forces the generated machine code to the consult the `StoreHandle` entity in order to retrieve the associated aspects.

We defined that querying the `StoreHandle` entity is slow because it has to search for associated aspect instances. This means that every time the `JoinPointSite` executes the `StoreHandle` entity is queried. Consulting the `StoreHandle` entity is in fact unnecessary if it is known that for one key-tuple the same aspect instance will be associated and stays associated. We have therefore defined a caching mechanism that operates at machine code level. The caching mechanism caches results (associated aspect instances) to belong to an evaluated query (key-tuple). Although the `StoreHandle` might change over time, it can still be assumed that a key-tuple is always associated to one and the same aspect instance. Figure 5.5 shows the generated machine code for this optimization strategy:

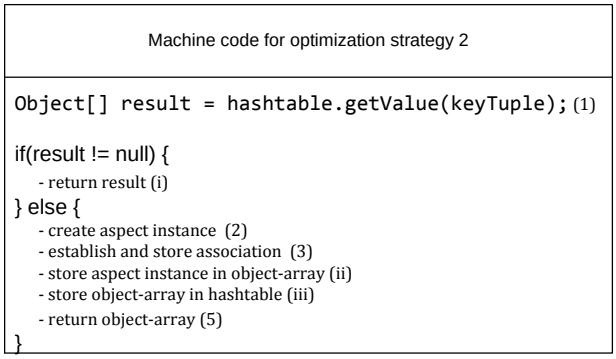


Figure 5-5 – Pseudo machine code instructions for the second optimization strategy

The generated machine code at location (1) determines at run-time if for a key-tuple a cached value (and thus an association) exists and corresponds to instruction point (1). The key-tuple is passed to a hash table which returns either `null` or an object-array. If `null` is returned then no cached result is available and likewise the key-tuple is not associated to an aspect instance. This causes that the machine code in the else-section will be executed. On the other hand, if an instance of object-array is returned then a cached result is available and likewise the key-tuple is associated to an aspect instance. This causes that the if-section will be executed. The following paragraphs elaborate on the machine code the two possibilities:

The machine code that is present in the else-section executes at run-time as follows: at location (2) an aspect instance is created and at (3) an association is established and stored. At location (5) an object-array is returned as result. The generated machine code corresponds to the instructions that were defined in the beginning of this section. Instruction (4) is not generated because the caching mechanism retrieves the association. The rest of the machine code is used to implement the caching mechanism. For every key-tuple that gets associated with an aspect instance an object-array is created that stores the aspect instance (ii). Because this optimization strategy follows an implicit instantiation strategy and thus the `PerTupleContext` has control over aspect instantiation, we can assume that for a given query the result will always be the same. The object-array will therefore always be valid as result. The object-array is stored in a hashtable (iii).

The hash table is global data structures and can be reached from every `JoinPointSite`. As already may be obvious, the *key* corresponds to the query (the key-tuple) and the *value* to the object-value.

The machine code that is present in the if-section executes when a cached result for a key-tuple is available. At location (i) the cached object-array will returned as result.

5.2.3 Optimization Strategy 3

Before the `generateASM` method is called, the machine code of the `JoinPointSite` method is already partially generated by the JIT-compiler. The second property, i.e. sensitive/insensitive on context values, of this optimizations strategy is set to context insensitive which means that the machine code that need to be generated does not depend upon the empty key-tuple. Therefore the empty key-tuple can be discarded as was done in the first optimization strategy.

The machine code that needs to be generated realizes the aspect instantiation policy for this optimization strategy. Before the generation of code starts, the `PerTupleContext` determines at compile-time what aspect instances are associated to the empty key-tuple. In order to retrieve the associated aspect instances, an exact query that contains no key-tuple is passed to the `StoreHandle` entity. The associations in the `StoreHandle` entity are fixed which means that for a given query the same set of aspect instances is always returned. The following strategy is generated that at run-time will have the following behavior:

- Return the associated aspect instances (1).

The machine code is generated as follows. First the associated aspect instances are retrieved at compile-time. Next an object-array is created in the non-moving space and filled with the retrieved aspect instances. The size of the object-array equals the number of associated aspect instances. Finally, the `generateASM` method generates the following machine code, shown in figure 5.6:

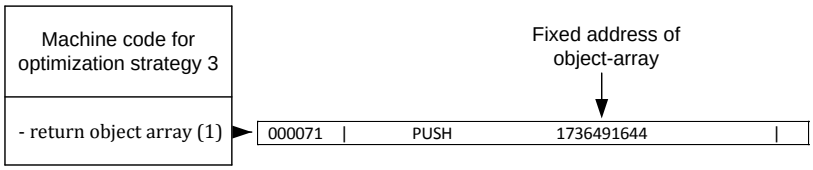


Figure 5-6 – Pseudo machine code instructions for the third optimization strategy

According to the properties of this configuration, only the result – containing the associated aspect instances – is returned. The machine code at location (1) contains a constant object reference and corresponds to instruction (1).

5.2.4 Optimization Strategy 4

This optimization strategy is the same as the third optimization strategy with the exception that the associations in the `StoreHandle` entity are not fixed and can thus change over time. This means that the key-tuple can be associated to a different set of aspect instances at any moment. The `generateASM` method therefore generates machine code that passes the key-tuple to the `StoreHandle` entity. The following strategy is generated that at run-time will have the following behavior:

- Retrieve the associations (1);
- Return the associated aspect instances (2);

When these instructions execute the `StoreHandle` entity is consulted (1) and the correct associated aspect instances are returned (2).

It might be the case that the associations in the `StoreHandle` entity do not change very often and furthermore we defined that consulting the `StoreHandle` entity is slow because it has to search for the associated aspect instances. We therefore define for this optimization strategy a caching mechanism that caches the retrieved aspect instances that belong to a key-tuple. The caching mechanism is the same mechanism used for the second optimization strategy, but in order to use it for this strategy a modification must be made. In the second optimization strategy the `PerTupleContext` has control over aspect instantiation and it knows that a key-tuple is always associated to the same aspect instance. In other words, the cached result is always valid. That does not apply for this optimization strategy because the associations in the `StoreHandle` entity can change and thus it cannot be guaranteed that a cached value is valid.

In order to determine if a cached result is valid, we have extended the caching mechanism with an invalidation mechanism. Everytime the cached result is used, the mechanism has to check if a result is valid. Because the cached value itself, i.e. the object-array containing the aspect instance, cannot be equipped with this mechanism a wrapper class is designed. This wrapper class, or more precisely the `ResultRef` class, wraps the cached results and implements the invalidation mechanism that works basically as follows: The `ResultRef` class is equipped with a state variable that represent whether a cached result is valid or invalid. To make a cached result invalid, the `ResultRef` is also equipped with a `invalidate` method that is called when the associations in the `StoreHandle` change.

The generated machine code for this optimization strategy is shown in figure 5.7.

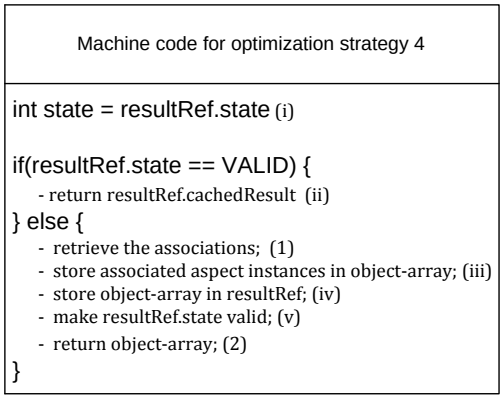


Figure 5-7 – Pseudo machine code instructions for the fourth optimization strategy

The generated machine code at location (1) corresponds to instruction (1) and retrieves the associated aspect instances for the empty key-tuple by consulting the `StoreHandle` entity. After the aspect instances are retrieved, they are returned as object-array at location (2) that corresponds to instruction (2). The rest of machine code is responsible for the caching mechanism. When the generated machine code at (i) executes, the state of the cached result is retrieved. If the state turns out to be valid then the cached result can be returned at (ii). If the state is invalid then the cached result needs to be updated. After the associations are retrieved at (1), an object-array is created where the associated aspect instances are stored (iii) and this object-array is then stored as cached result (iv). As a last action, the machine code at (v) sets the state to valid so that next time the cached result can be returned.

5.2.5 Optimization Strategy 5

This optimization strategy equals the second strategy, except that an explicit instantiation is used. The key-tuple is used in order to retrieve the associated aspect instances. The `generateASM` method generates the instructions that at run-time will have the following behavior:

- Retrieve the associations (1);
- Return the associated aspect instances (2);

When these instructions execute the `StoreHandle` entity is consulted (1) and the associated aspect instances are returned (2). In contrast to the second optimization strategy the number of associated aspects depends on the associations in the `StoreHandle` entity. The associations in the `StoreHandle` entity are fixed which means that for the key-tuple the same number of aspect instances is always returned.

Because consulting the `StoreHandle` entity is defined as slow, we also implement for this optimization strategy a caching mechanism. Figure 5.8 shows the generated machine code for this optimization strategy:

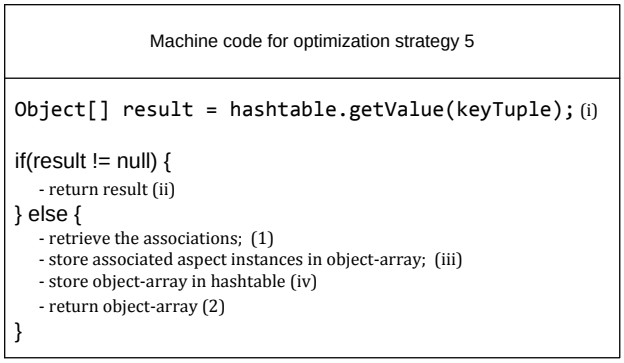


Figure 5-8 – Pseudo machine code instructions for the fifth optimization strategy

The generated machine code at location (1) corresponds to instruction (1) and retrieves the associated aspect instances by performing an exact query to the `StoreHandle` entity. After the aspect instances are retrieved, they are returned as object-array at location (2) that corresponds to instruction (2). The rest of machine code is used for the caching mechanism and executes at run-time as follows:

The machine code at location (i) passes the key-tuple to a hashtable that returns either an object-array containing the cached value, or it returns `null` and in that case no result is cached. If no result is cached then the machine code at (1) executes and retrieves the associated aspect instances. Next, machine code at location (iii) stores the associated aspect instances in an object-array and this object-array is stored into a hashtable at location (iv). In case a result is not null then a result is cached and machine code at (ii) executes that returns the cached result.

5.2.6 Optimization Strategy 6

This optimization strategy equals the fifth strategy, except that the associations in the `StoreHandle` are not fixed. The `generateASM` method generates the instructions that at run-time will have the following behavior:

- Retrieve the associations (1);
- Return the associated aspect instances (2);

When these instructions execute the `StoreHandle` entity is consulted (1) and the associated aspect instances are returned (2). The associations in the `StoreHandle` entity are not fixed and can thus change over time.

It might be the case that the associations in the `StoreHandle` entity do not change very often and furthermore we defined that consulting the `StoreHandle` entity is slow because it has to search for the associated aspect instances. We therefore define a caching mechanism with an invalidation mechanism that caches the retrieved aspect instances that belong to a key-tuple.

The generated machine code for this optimization strategy is shown in figure 5.9.

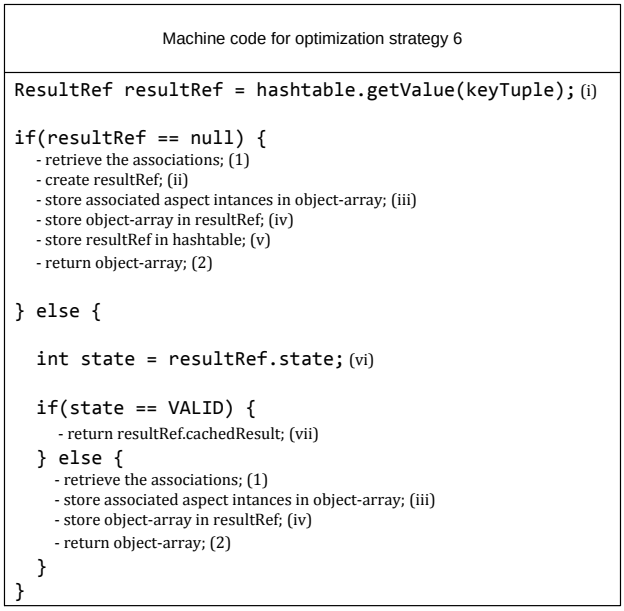


Figure 5-9 – Pseudo machine code instructions for the sixth optimization strategy

The generated machine code at location (1) corresponds to instruction (1) and retrieves the associated aspect instances by performing an exact query to the `StoreHandle` entity. After the aspect instances are retrieved, they are returned as object-array at location (2) that corresponds to instruction (2). The rest of machine code is used for the caching mechanism and executes at run-time as follows:

When the generated machine code at (i) executes, a key-tuple is passed to a hashtable that either returns `null` or a `ResultRef` object. If `null` is returned then no result is cached and the machine code in the outer if-section executes. The machine code at (1) retrieves the associated aspect instances by consulting the `StoreHandle` entity. Next, a `ResultRef` object is created at location (ii). The retrieved associated aspect instances are stored in an object-array at (iii) and the object-array is stored in the `ResultRef` object at (iv). The `ResultRef` object is stored in the hashtable at (v) so that next time a cached result is available. Finally the object-array is returned as result at (2).

If a `ResultRef` object is returned at (i) then a cached result is available. Because the associations in the StoreHandle entity are not fixed it might possible that the cached result is invalid. Therefore the state is retrieved at (vi) and checked. If the state is valid then the inner-if section executes and the cached result is returned at location (vii). If the state is invalid then the inner-else section executes that updates the old cached value. At (1) new aspect instances are retrieved which then are stored in an object-array at (iii). The object-array is stored in the `ResultRef` object at (iv) whereby thus the old cached value is overwritten. Finally the object-array is returned as result at (2).

5.2.7 Optimization Strategy 7

This optimization strategy equals the fifth optimization strategy except that the `StoreHandle` entity is queried with a range query.

5.2.8 Optimization Strategy 8

This optimization strategy equals the sixth optimization strategy except that the `StoreHandle` entity is queried with a range query.

5.3 Additional work – Dynamic Linking

Dynamic linking in Java means that classes and methods that form the application are loaded and linked on demand during execution [**14**]. The purpose of dynamic linking is that only the necessary parts of the program are loaded and linked. This results into a faster program startup and no compilation and linking time is wasted on parts that are not used.

Before a class or method can be linked into the running program they are first need to be loaded. With loading we mean that the machine code of a class or method of a given name is found. The compiled machine code cannot be executed yet, because it must first be linked. The linking process includes that the machine code is combined into the runtime state of the JikesRVM so that it can be executed. Dynamic linking involves three activities: verification, preparation and resolution of symbolic references. Here we focus only on the last activity.

For the eight optimizations strategies discussed in previous sections, specific machine code is generated that represents the aspect-instantiation policy. The first two optimization strategies make use of an implicit instantiation strategy and this means that the generated machine code may refer to a class or method that at run-time is not yet loaded or linked. There is thus a chance that dynamic liking occurs when the generated machine code executes. Because we let the `PerTupleContext` played a role of JIT-compiler, we also must take care that the generated code supports dynamic linking.

The remainder of this section discusses how the generated code support dynamic linking and what kind of adaptations were made. We first explain in the next intermezzo how it is supported in the JikesRVM:

INTERMEZZO IV

The following code excerpt shows an example when dynamic linking occurs. In the example the JIT-compiler is compiling the bytecode of method `m` to final machine code. When the JIT-compiler encounters the call to `invokespecial <init>()V` on line three, it tries to find where the machine code of the `<init>()V` method is located. The location of the machine code is used by the JIT-compiler in order to emit the call instruction.

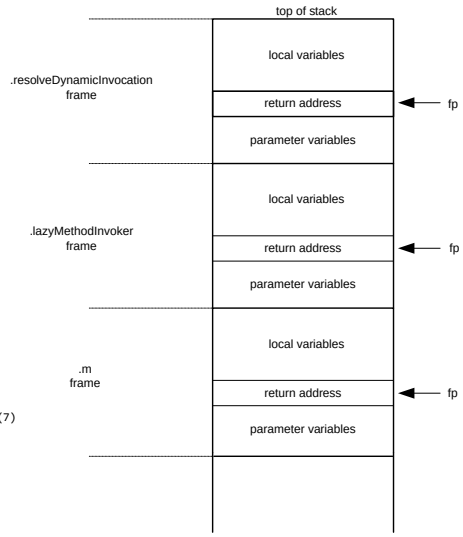
```
     bytecode of method m

1    new Apple
2    dup
3    invokespecial <init>()V     <---- method reference
```

When the `invokespecial` instruction on line 3 is compiled to final machine code, the location of the machine code of the `<init>()V` method is searched. If the `<init>()V` method was used earlier then the corresponding machine code is prepared to be executed and thus no loading or dynamic linking is needed. If, on the other hand, the `<init>()V` method was never used earlier then it has to be loaded and linked before it can be executed.

When the Apple class was loaded, its Type Information Block (TIB) is initialized. A TIB contains pointers to the compiled method bodies (executable code) for the virtual methods. The pointer to the compiled method body of `<init>()V` points to a so-called method stub. The call that is generated by the JIT-compiler will therefore try to execute the method stub. Until now, no dynamic linking occurred.

As soon as the method-stub is reached dynamic liking will occur and the *lazy method invoker* is executed. The purpose of this *lazy method invoker* is to update the TIB with the correct executable code for the `<init>()V` method. The pseudo-code for the lazy method invoker is as follows:

```
public class DynamicLinker {

  static void lazyMethodInvoker() {

    DynamicLink dl = resolveDynamicInvocation(); (1)
    ...
    // compile dl.targetMethod and update TIB; (2)
    ...
  }


  static DynamicLink resolveDynamicInvocation() {

    Address callingFrame = getCallerFP(getFP()); (3)
    Address returnAddress = getReturnAddress(callingFrame); (4)
    callingFrame = getCallerFP(callingFrame); (5)

    CompiledMethod cm = getCompiledMethod(callingFrame); (6)
    Offset callingInstructionOffset = cm.getInstructionOffset(returnAddress); (7)

    DynamicLink dl = new DynamicLink(); (8)
    cm.getDynamicLink(dl, callingInstructionOffset); (9)

    return dl; (10)
  }

}
```



The compilation of the `<init>()V` method and updating of the TIB occurs at location (2). But before this can be done, the `resolveDynamicInvocation` method is called at (1). In this method a `DynamicLink` object is created and the target method, i.e. the method reference to `<init>()V`, is set.

In order to find the target method the stack is inspected. At the right-hand side the stack is shown according to the situation when the `resolveDynamicInvocation` method is called. The topmost frame of the stack is thus the `resolveDynamicInvocation` method frame. Then at (3) the calling frame, i.e. `lazyMethodInvoker` method, is retrieved and at (4) its return address is stored. This return address points to an instruction of the calling frame of the `lazyMethodInvoker` method frame. One step further at (5) the `lazyMethodInvoker` method's calling frame is retrieved which is the frame of method `m`. Via an id in the method frame (not shown), the compiled method is retrieved at (6). The compiled method is an internal entity where Jikes stores the method's machine code and other compilation state data.

At location (7) an offset is calculated between the stored return address and the location of the machine code of `m`. The `DynamicLink` object is created at (8) and as final action the `getDynamicLink` method is called at (9) whereby two arguments are passed: 1) the `DynamicLink` object and 2) the offset.

The purpose of the `getDynamicLink` method is to locate the target method, i.e. the `<init>()V` method and store it in the `DynamicLink` object. The target method is located via the offset variable. Namely the offset variable points to an instruction in `m` of where to continue. If it is known where to continue, it can also be known what instruction caused the call. Once that instruction is known, the corresponding bytecode index is calculated. For this example the bytecode would be index 3 and thus the reference to the`<init>()V` method is retrieved and stored in the `DynamicLink` object.

The dynamic linking process as described in the intermezzo works perfectly for machine code that is generated by the JIT-compiler.

Once the calling instruction is known the corresponding bytecode instruction is calculated. This mechanism works thus perfectly for machine code that was compiled by Jikes' baseline compiler

because it knows how the machine code is related to the bytecode and a mapping can be easily made. This mapping is shown in figure 5.10.

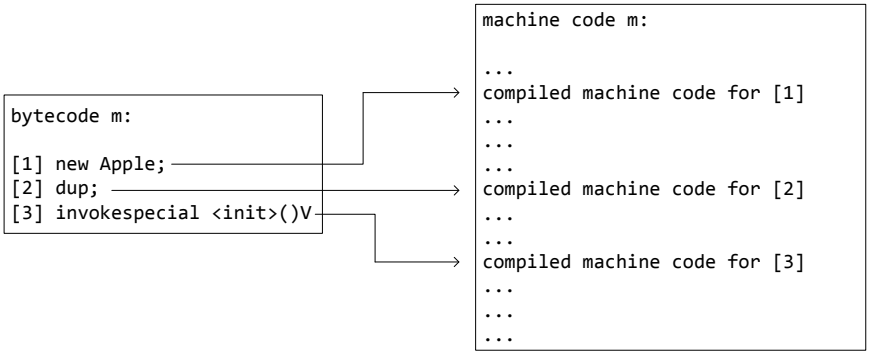


Figure 5-10 – Correct mapping of bytecode instructions to machine code instructions

If, however, the `generateASM` method generates machine code then the relation between machine code and bytecode is different. Figure 5.11 shows this:

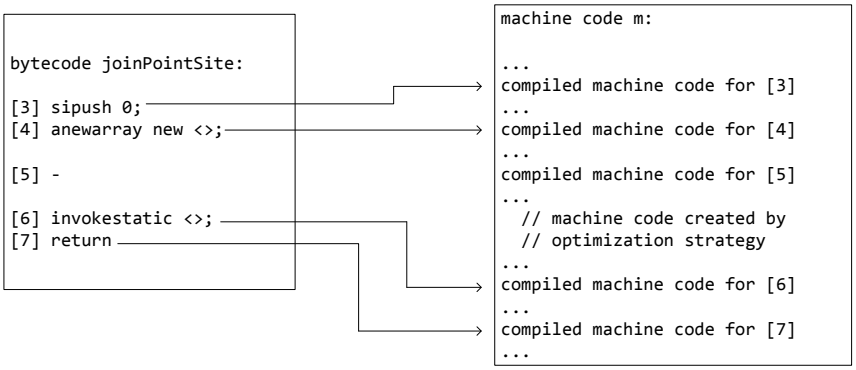


Figure 5-11 – Incorrect mapping of bytecode instructions to machine code instructions

The machine code instructions that are generated by the `generateASM` method do not map to a corresponding bytecode instruction. Therefore an additional mapping system is designed that sets the correct data in the dynamic link object.

Chapter 6

Experiments

This chapter presents the setup and results of the benchmarks that are performed on aspect-instantiation policies. We use the three aspect-instantiation policies of the generalized model and measure their performance in AspectJ [**4**], Association Aspects [**3**] and ALIA4J [**8**].

This chapter is organized as follows. Section 6.1 discusses the micro benchmark platform that is designed as a separate project within this work. The benchmarking methodology is discussed in section 6.2 and describes how the experiments are setup and how the data is analyzed and, finally, section 6.3 shows the results of the experiments.

6.1 Micro Benchmark Platform

The micro benchmark platform is designed as a separate project within the Steamloom^{ALIA} virtual machine that is able to measure the execution time of a part of a method and produces one data point per benchmark invocation. Prior to the invocation of the benchmark platform it must be notified which class and method are going to be measured. Once the benchmark platform is running, an object of the given class is instantiated, the object's method is called a number of times and finally one data point is produced that contains the time that was needed to execute the method. In next sections it becomes clear how this data point is retrieved and how these data points are analyzed in order get a result.

At first sight it seems rather trivial to measure the performance of a method, but since non-determinism at run-time causes the execution of a Java application to differ from run to run, it turns out to be complicated [**15**]. Typical sources of non-determinism are garbage collection, thread scheduling, and various other hidden system effects. There exist different methodologies on benchmarking applications that deal with non-determinism but as described in [**15**], these methodologies differ from each other in a number of ways. It is shown that certain methodologies can be misleading and even can produce incorrect conclusions because the data analysis is not statistically rigorous. In this work a methodology is adopted that is statistically rigorous for the benchmarks that are going to be performed by the benchmark platform. This methodology is described in section 6.2.

The benchmark platform consists of a central class called `MicroBenchmark` and implements the benchmark harness. Prior to benchmarking, a `MicroBenchmark` object is created whereby the configuration values, i.e. what class and method are going to be measured, are passed as parameters to the constructor. During benchmarking the harness performs the measurements by creating an object of the given class and executes the object's method a number of times until a stable result is reached. Finally the result is returned as a data point.

In order for a class to be acceptable by the benchmark platform, it must implement the `OperationHarness` interface that is described in the next section. Thereafter, the `MicroBenchmark` platform is described in detail and shows how a stable result is retrieved.

6.1.1 The `OperationHarness` Interface

The `OperationHarness` interface is shown in listing 6.1. The implementing class implements the three methods which then can be called by the platform. The `setup` method is called once and is used to initialize the benchmark, for instance, the deployment of an attachment or configuration of an aspect.

```
1  public interface OperationHarness {
2
3      Object setup();
4      void operation();
5      void nullOperation();
6  }
```

Listing 6-1 – The `OperationHarness` interface

The purpose of the `operation` and `nullOperation` methods is to counterbalance each other in such a way that only the instructions of interest in the `operation` method are measured. In Steamloom[ALIA] the unit of compilation is a method and the same applies for this benchmark platform; the unit of measurement is a method. Often a method contains more than just the instructions that need to be measured and removing them is not always possible. For example, in FIAL the `operation` method would be executing the machine code that was generated by the `PerTupleContext`, but the machine code can never be executed alone, only during a generic function dispatch. As explained in previous chapters, the generated machine code is generated within a `JoinPointSite` method. Before the generated machine code executes, a key-tuple is created and afterwards the advice is executed within the `JoinPointSite` method. When the `JoinPointSite` method is measured, the benchmark platform thus also measures the key-tuple creation and advice execution.

In order to measure the instructions of interest, i.e. the machine code of the aspect-instantiation policy, the `operation` and `nullOperation` methods must be implemented as follows. The `operation` method is the method that contains the generic function call and the instructions that realize the aspect-instantiation policy and advice execution. The `nullOperation` also contains a generic function call, but in contrast to the `operation` method, no aspect-instantiation policy is present.

The net result of the measurement is then the execution time of the aspect-instantiation policy and is calculated by subtracting the execution time of `nullOperation` method from the `operation` method. The instructions in the `nullOperation` thus neutralize the instructions in the `operation` method that must not be measured. For every benchmark case we must define what instructions have been placed in the `operation` method and `nullOperation` method. If the `nullOperation` method is not carefully defined then the benchmarking can lead to false results.

6.1.2 The Benchmark harness

The class `MicroBenchmark` implements the benchmark harness and can be configured with an operation to benchmark. Starting the `MicroBenchmark` class will be referred to as a virtual machine invocation and produces one data point. During benchmarking it repeatedly does the following actions until a stable result is reached: 1) record the start time 2) execute the method that needs to be measured 3) record the stop time 4) calculate the difference between start en stop time and 5) return the result as data point.

Once the `MicroBenchmark` class has started to run, its benchmark harness begins with the benchmarking. The benchmarking process performs three nested loops in order to get one stable data point. The inner loop is the so-called *measure-iterations-loop* and invokes the `operation` and `nullOperation` method alternately.

The inner loop and the `operation` method are shown in listing 6.2. The `operation` method is invoked fifteen times on line 9 – 11. The fifteen invocations reduce the effect of executing the loop and the conditional logic and avoid performance problems of tight loops. Just above the `operation` method is the `nullOperation` method and is also invoked 15 times.

The constant `MEAURE_ITERATIONS` is set to `8000` which means that the `operation` and `nullOperation` methods are invoked `120,000` times everytime the loop runs. The value of the `MEAURE_ITERATIONS` constant and the number of fifteen copies have been determined by trial and error. The variable `means` is used to switch between the `operation` and `nullOperation` method.

```
1   for (long i = 0; i < MicroBenchmark.MEASURE_ITERATIONS; i++) {
2       if (means == Means.BASELINE) {
3           // baseline
4           operationHarness.nullOperation();
5           operationHarness.nullOperation();
6           // ... 13 more times
7       } else {
8           // real measurement
9           operationHarness.operation();
10          operationHarness.operation();
11          // .. 13 more times
12      }
13  }
```

Listing 6-2 – The measure-iterations-loop

The next loop is the *means-loop* and is shown in listing 6.3. In this loop the `means` variable is altered (line 1) every time the loop is entered. The statement on line 2 disables the garbage collector for the following reasons. The first reason is that garbage collector thread does not run. In a normal situation this thread runs in parallel with the program's main thread and it becomes active when a certain threshold of free memory is reached. Because we do not know when this thread becomes active it is a source of non-determinism and can pollute the measurement. The second reason is that thread-switching is disabled whereby another potential source of non-determinism is disabled.

However, there is still a chance that the operating system decides to change between processes. We have no control of this and thus it cannot be guaranteed that a measurement is totally free of pollutions. The next loop explains how it can be detected if a measurement is possible polluted by a hidden source of non-determinism. After garbage collection has been disabled the current system time is stored in variable `start` on line 4. The *measure-iteration-loop* runs on line 6 and executes the `operation` or `nullOperation` method as explained. Then on line 8 the current system time is stored again in variable `end`. The difference between the `start` and `end` gives the times spent in the measure-iteration-loop. Because the execution of the `nanoTime` method is also included in the measurement, we measure its execution time and subtract it from the result (line 10 – 12). Line 14 enables the garbage collector so that the garbage collector gets the opportunity to remove dead object that were created during the measurement. The last statements (line 17 – 20) store the `exectionTime` of either the `operation` or `nullOperation` method in a list. The list is used by the next loop which is discussed next.

```
1   for (Means means : Means.values()) {
2       VM.disableGC(); // also disables thread switching, must be enabled after measurement
3
4       long start = System.nanoTime();
5
6       // execution of the inner loop
7
8       long end = System.nanoTime();
9
10      long creation = System.nanoTime();
11      long creationTime = creation - end;
12      long executionTime = end - start - creationTime;
13
14      VM.enableGC();
15      System.gc();
16
17      if (means == Means.BASELINE)
18          // store execution time in operationTimesList
19      else
20          // store execution time in nullOperationTimesList
21  }
```

Listing 6-3 – The means-loop

The last loop is the called the *outer-loop* and is shown in listing 6.4. The outer-loop runs until either the `iteration` variable becomes greater or equal to `MAX_ITERTATIONS_PANIC` which is set to `100`, or if the measurement results are converging. The measurement results are converging if for both the `operationTimes` list and the `nullOperationTimes` list, the list's maximum value divided by the list's minimum value does not exceed a `CONVERGENCE_THRESHOLD`. Initially the `CONVERGENCE_THRESHOLD` variable is set to `0.05`.

If after `MAX_ITERATIONS` the measurement results are not converging then the `CONVERGENCE_THRESHOLD` is increased by 5% at the end of the loop iteration (that means that it is increased by 5% of its own value not by an absolute value of 5%). Thus if the strict convergence rule cannot be met, then it is loosened with iterations.

When the outer-loop exits we have measured at least 5 times (`MicroBenchmark.MIN_ITERATION` is set to 5), the execution time of the `operation` and `nullOperation` method which only differ by `CONVERGENCE_THRESHOLD` * 100%. It can therefore be assumed that the last measurement was no peak, since it behaved similar to the last five ones. Therefore, we take the last measurement as the data point for this VM invocation.

```
1   while ((iteration < MicroBenchmark.MIN_ITERATION ||
2       MicroBenchmark.notConverging(nullOperationTimes, operationTimes, false))
3       && iteration < MicroBenchmark.MAX_ITERATIONS_PANIC) {
4
5           // means-loop
6
7           iteration++;
8
9           if (MicroBenchmark.notConverging(nullOperationTimes, operationTimes, false)
10              && iteration > MicroBenchmark.MAX_ITERATIONS) {
11
12              MicroBenchmark.CONVERGENCE_THRESHOLD *= 1.05D;
13          }
14  }
```

Listing 6-4 – The outer-loop

6.2 Methodology

This section discusses the methodology and consists of an *experimental design* part and *data analysis* part.

The experimental design refers to setting up the experiments: the benchmarks were run using a machine with an Intel Pentium 4 CPU, running at 2.8 GHz. The operation system was Ubuntu Linux using kernel version 2.6.35-24-generic. The version of the JikesRVM was 2.9.3 and the heap size was configured to be 2048 Mb.

Data analysis refers to analyzing the data obtained from the experiments. During the experimental design, two types of errors can occur; systematic errors and random errors. Systematic errors, or biases, occur because of an experimental mistake or incorrect procedure (for instance, too many instruction in the `nullOperation`). Random errors are unpredictable and non-deterministic and can be hard to detect, when a random error occurs it influences the result of a measurement. An example of a random error is an external system event and it is impossible to predict when such a system event occurs. Because they are invisible and have an influence on the result, a statistical model in [**15**] describes the overall effect of random error on experimental results. The statistical model is discussed next.

Starting the `MicroBenchmark` class will be referred to as a virtual machine invocation and produces one data point. After *x* virtual machine invocations, a population of *x* data points is available. A confidence interval for the mean derived from samples then quantifies the range of values that have a given probability of including the actual population mean. The way of how confidence intervals are built is essentially always the same, but in [**16**] a distinction is made on the number of samples gathered from the underlying population: The number of samples *n* is large (n >= 30) or the number of samples is small (n < 30).

Since we have control of performing the benchmarks, we choose to have large *n*; in order to build a confidence interval for the mean, we use an approach that is applicable for a large *n*:

Building the confidence interval requires that we have a number of measurements x_i, $1 <= i <= n$, from a population with mean μ and variance σ^2. The mean of the measurements $\bar{x}$ is computed as

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

The actual true value of μ is approximated by the mean of measurement $\bar{x}$ and a range of values [c1, c2] around $\bar{x}$ is computed that defines the confidence interval at given probability. The confidence interval [c1, c2] is defined such that the probability of μ being between c1 and c2 equals 1 – α; α is called the *significance level* and (1 – α) is called the *confidence level*.

The approach we use to compare two measurements is by looking whether the confidence intervals overlap. We therefore use a 95% confidence interval that means that there is a 95% probability that the actual distribution of the mean of the underlying population, μ, is within the confidence interval.

If the two confidence intervals do overlap, then it cannot be concluded that the differences seen in the mean are not due to random fluctuations in the measurements. Or in other words, the difference seen in the mean values is possibly due to random effects. If there is no overlap then it can be concluded that there is no evidence to suggest that there is not a statistically significant difference.

6.3 Benchmarks

This section describes the benchmarks that have been performed on several aspect-instantiation policies. The benchmarks are divided into several sections. Section 6.3.1 benchmarks singleton policies that are defined as aspect in AspectJ and Association Aspects against singleton policies defined as an attachment in ALIA4J. The attachment is configured with the optimized version of the `PerTupleContext`. Section 6.3.2 benchmarks per-object policies that are defined as per-target aspect in AspectJ and Association Aspects against per-target policies defined as attachment in ALIA4J. Section 6.3.3 benchmarks per-group policies that are defined as aspect in Association Aspect against per-group policies defined as attachment in ALIA4J. Finally section 6.3.4 benchmarks policies defined as attachment configured with the unoptimized version of the `PerTupleContext` against attachments configured with the optimized version of the `PerTupleContext`.

6.3.1 Benchmarking singleton policies

This section benchmarks the singleton policy. AspectJ and Association Aspects only support singleton policies that follow an implicit instantiation strategy and therefore we define exactly one aspect for them. With ALIA4J, however, we can either define whether an implicit or explicit instantiation strategy is used and likewise two attachments are defined. The following table shows how the benchmark is defined and how the four properties are configured to represent the singleton policy.

	benchmark 1 (AJ)	**benchmark 2 (AA)**	**benchmark 3 (ALIA4J)**	**benchmark 4 (ALIA4J)**
implicit / explicit	implicit	implicit	implicit	explicit
sensitive / insensitive	insensitive	insensitive	insensitive	insensitive
exact / range	exact	exact	exact	exact
fixed / not-fixed	not-fixed	not-fixed	not-fixed	fixed

Benchmark 1 and 2:
In benchmark 1 and 2 the singleton instantiation policy is measured in AspectJ and Association Aspects. The policy is implemented in AspectJ by defining an aspect called `AJ_1_1`, and is shown on the left-hand side of listing 6.5. Because in Association Aspects the singleton policy is implemented in the same way, an equivalent aspect called `AA_1_1` is shown on the right-hand side. The aspect contains one pointcut-and-advice pair. The pointcut matches a call in the `operation` method of the benchmark case. The advice runs afterwards and performs a call to a static `advice` method that increases a trivial counter.

```
1   public aspect AJ_1_1 {
2
3       pointcut p1():
4           call(void *.*.method_AJ_1_1());
5
6       after(): p1() {
7           Advice.advice(this);
8       }
9   }
```

```
1   public aspect AA_1_1 {
2
3       pointcut p1():
4           call(void *.*.method_AA_1_1());
5
6       after(): p1() {
7           Advice.advice(this);
8       }
9   }
```

Listing 6-5 – Aspect `AJ_1_1` and `AA_1_1`

Listing 6.6 shows on the left-hand side how the `operation` and `nullOperation` method are defined for the `AJ_1_1` aspect. Before the two methods are discussed, we first mention that the instructions in both methods are also applicable for the `AA_1_1` aspect because it is compiled in the same way as the `AJ_1_1` aspect.

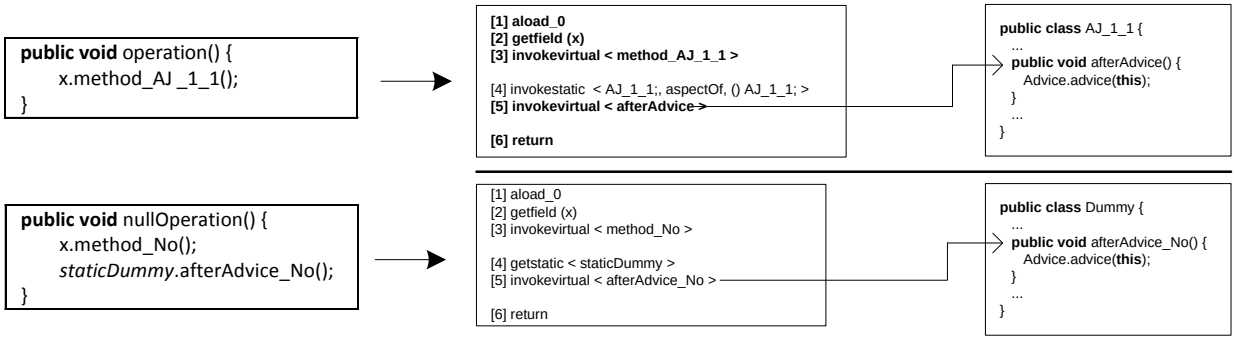
The `operation` method contains the join point shadow `x.method_AJ_1_1()` and is advised by the aspect's advice. The `nullOperation` method contains the instructions that neutralize the instructions that must not be measured. In order to define which instructions are placed in the `nullOperation` method, we first discuss the corresponding bytecode of the `operation` method.

The center of listing 6.6 shows the corresponding bytecode. The bytecode of the `operation` method has six instructions and the instructions that must be neutralized are shown in bold. Instructions 1 – 3 correspond to the join point shadow. Instruction 4 retrieves the aspect instance and is the receiver object for instruction 5 that executes the advice by a calling the aspect's `afterAdvice` method. Instruction 6 returns. The instruction that must be measured is the retrieval of the aspect instance.

The instructions that are placed in the `nullOperation` must thus neutralize all instructions, except instruction 4. The following instructions are therefore placed in the `nullOperation`:

- Instructions 1 – 3 neutralize the join point shadow.
- Instruction 5 is call to the `afterAdvice_No` method and neutralizes the call to the `afterAdvice` method. The implementation of the `afterAdvice_No` method equals the `afterAdvice` method as shown on the right-hand side. In order for the method to execute, a receiver object is needed. This requires that one additional instruction is needed to retrieve the receiver. Because this additional instruction must have a minimal impact on the measurement we retrieve the receiver object by referring to a global static variable.

Because of the additional instruction, the `nullOperation` method does not make a perfect neutralization. The results of this benchmark are therefore a little bit too low.



Listing 6-6 – `operation` and `nullOperation` methods for the AspectJ/Association Aspects benchmark

Benchmark 3:

In order to implement the benchmark in ALIA4J an attachment is deployed. An object diagram of the attachment is shown in figure 6.1. The `PerTupleContext` follows an implicit instantiation strategy whereby an object of `ALIA_1_1` will be created that represents the aspect instance. The pointcut matches a call in the `operation` method of the benchmark case. The advice runs afterwards and performs a call to a static `advice` method that increases a trivial counter.

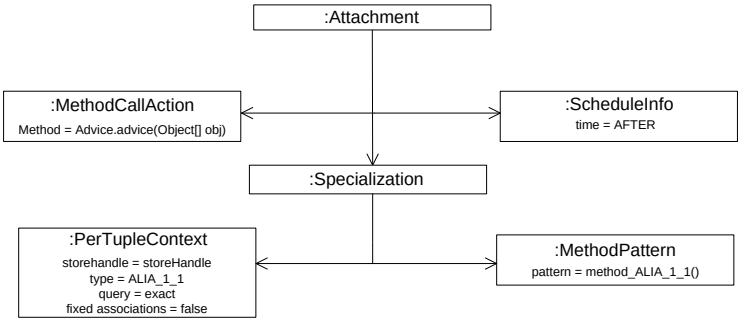


Figure 6-1 – Object diagram of the attachment following an implicit instantiation strategy

Listing 6.7 shows on the left-hand side how the `operation` and `nullOperation` method are defined. The `operation` method contains the join point shadow `x.method_ALIA_1_1()` and when it executes it is advised by the advice of the attachment. The `nullOperation` method contains the instructions that neutralize the instructions that must not be measured. In order to define which instructions are placed in the `nullOperation` method, we first discuss the corresponding bytecode of the `operation` method.

The right-hand side of listing 6.7 shows the corresponding bytecode. The bytecode of the `operation` method contains of two parts. In the first part the join point shadow is replaced with a static call to the `JoinPointSite` method as was explained in intermezzo 1. The bytecode of the first part contains six instructions and the instructions in bold must be neutralized.

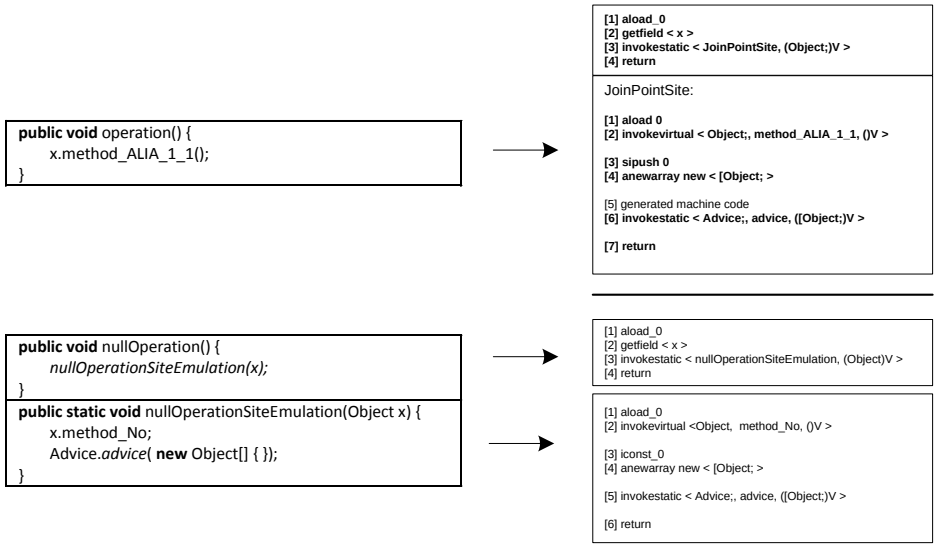
The second part contains the `JoinPointSite` instructions. Instructions 1 – 2 correspond to the join point shadow. Instructions 3 – 4 create an array that holds the key-tuple. Instruction 5 contains the machine code that was generated by the `generateASM` method and is the instruction that must be measured. Instruction 6 is the call to the `advice` method whereby an object-array with the retrieved aspect instances is passed as parameter. The last instruction returns.

In order to measure the instruction of interest, i.e. instruction 5, we define the `nullOperation` method as follows:

- Instructions 1 – 4 neutralize the first part of the `operation` method. Hereby a static call to the `nullOperationSiteEmulation` is made.

In order to neutralize the second part of the `operation` method, the following instructions are placed in the `nullOperationSiteEmulation` method.

- Instructions 1 – 2 neutralize the join point shadow.
- Instructions 3 – 4 neutralize the empty array creation in the `JoinPointSite` method. The creation of the empty array is not a part of the aspect-instantiation policy.
- Instruction 5 neutralizes the call to the `advice` method by creating a call to the `advice_No` method. Because the advice method requires an object-array we use the empty array that was created by instructions 3 – 4



Listing 6-7 – `operation` and `nullOperation` methods for the ALIA4J benchmark

Benchmark 4:

In benchmark 4 the singleton instantiation-policy is measured in ALIA4J. In this benchmark an explicit instantiation strategy is used and the associations in the `StoreHandle` entity are fixed. An object diagram of the attachment is shown in figure 6.2.

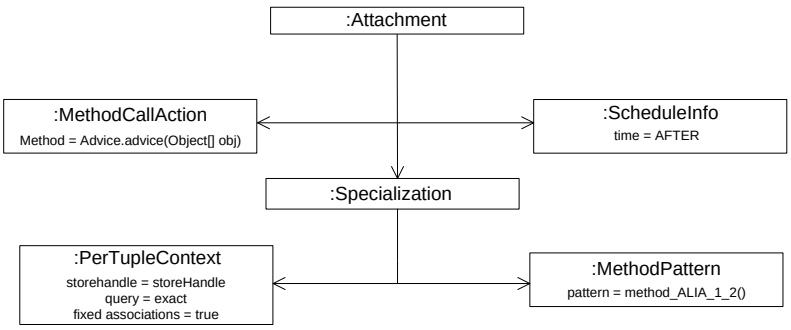
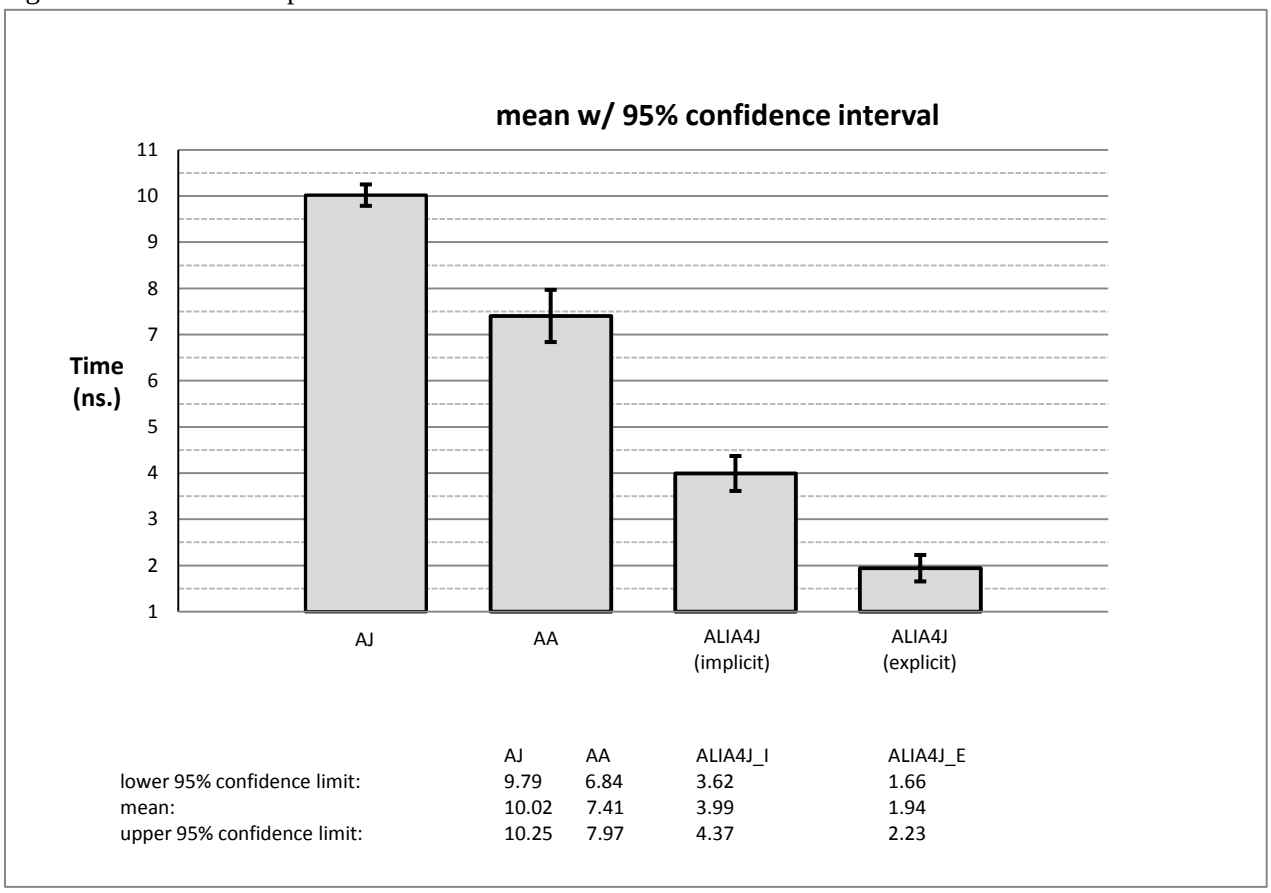


Figure 6-2 – Object diagram of the attachment that follows an explicit instantiation

The `operation` and `nullOperation` methods are not discussed for this benchmark because they equal the methods of the first benchmark. The only difference is seen in the `operation` method where generated machine code differs, but this is just what must be measured.

Figure 6.3 shows the elapsed time for the four benchmark cases.



	AJ	AA	ALIA4J_I	ALIA4J_E
lower 95% confidence limit:	9.79	6.84	3.62	1.66
mean:	10.02	7.41	3.99	1.94
upper 95% confidence limit:	10.25	7.97	4.37	2.23

Figure 6-3 – Mean and confidence interval of the singleton policy benchmark

Conclusion: ALIA4J_I is on average 2.5 times as fast as AspectJ and on average 1.9 times as fast as Association Aspects. The confidence intervals of the results do not overlap and this means that there is no evidence to suggest that there is no statistical significant difference[5]. ALIA4J_E is on average 5.2 times as fast as AspectJ and on average 3.8 times as fast as Association Aspects. Also here the confidence intervals do not overlap which means that there is no evidence to suggest that there is no statistical significant difference. The fact that ALIA4J_E is faster than ALIA4J_I is because in ALIA4J_E the result is directly returned without the need to check if an association exists. In ALIA4J_I it must be determined whether an association exists and it is therefore bit slower than ALIA4J_E.

[5] There is in fact still a probability of 5% that the differences are simply due to random effects. We can thus not assure with 100% certainty that there is an actual difference between the measurements.

6.3.2 Benchmarking per-target policies

This section benchmarks the par-target policy. AspectJ and Association Aspects only support per-target policies that follow an implicit instantiation strategy and therefore we define exactly one aspect for them. With ALIA4J, however, we can either define whether an implicit or explicit instantiation strategy is used and likewise two attachments are defined. The following table shows how the benchmark is defined and how the four properties are configured to represent the per-target policy.

	benchmark 1 (AJ)	**benchmark 2 (AA)**	**benchmark 3 (ALIA4J)**	**benchmark 4 (ALIA4J)**
implicit / explicit	implicit	implicit	implicit	explicit
sensitive / insensitive	sensitive	sensitive	sensitive	sensitive
exact / range	exact	exact	exact	exact
fixed / not-fixed	not-fixed	not-fixed	not-fixed	fixed

Benchmark 1 and 2:
In benchmark 1 and 2 the per-target instantiation policy is measured in AspectJ and Association Aspects. In order to implement the per-target policy in AspectJ we have defined an aspect called `AJ_2_1`, shown on the left-hand side of listing 6.8. Because in Association Aspects the per-target policy is implemented in a same way, an equivalent aspect called `AA_2_1` is shown on the right-hand side. The aspect contains one pointcut-and-advice pair. The pointcut matches a call in the `operation` method of the benchmark case. The advice runs afterwards and performs a call to a static `advice` method that increases a trivial counter.

```
1  public aspect AJ_2_1 pertarget(p1()) {
2
3      pointcut p1():
4          call(void *.*.method_AJ_2_1());
5
6      after(): p1() {
7          Advice.advice(this);
8      }
9  }
```

```
1  public aspect AA_2_1 pertarget(p1()) {
2
3      pointcut p1():
4          call(void *.*.method_AA_2_1());
5
6      after(): p1() {
7          Advice.advice(this);
8      }
9  }
```

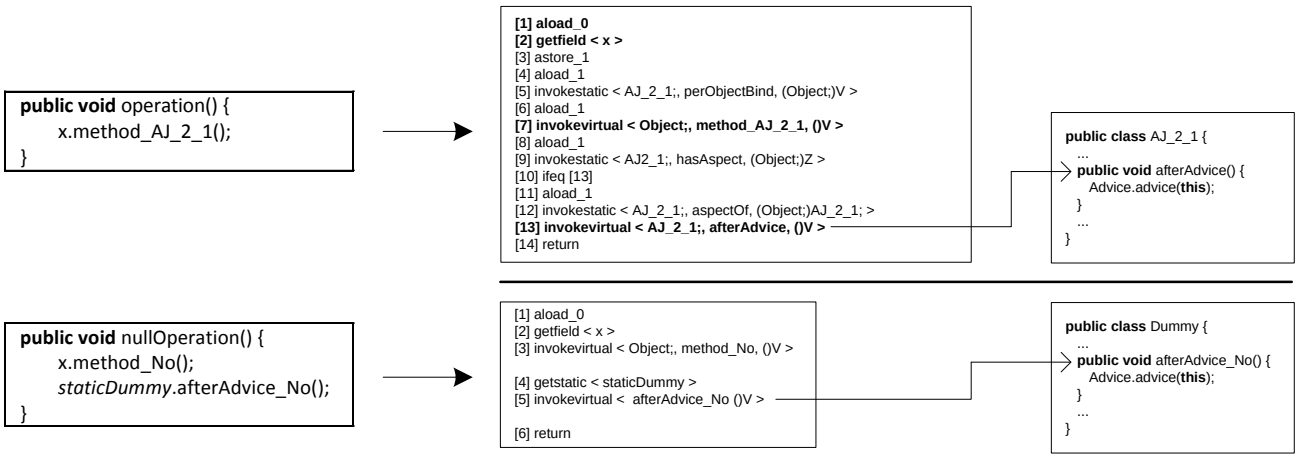
Listing 6-8 – Aspect `AJ_2_1` and `AA_2_1`

Listing 6.9 shows on the left-hand side how the `operation` and `nullOperation` method are defined. The corresponding bytecode instructions are shown on the right-hand side, the instructions in bold must be neutralized. Just as in the previous section, aspect `AA_2_1` is compiled in same way as `AJ_2_1` and therefore the `operation` and `nullOperation` method also apply for this aspect.

The instructions in the `nullOperation` method are as follows:

- Instructions 1 – 3 neutralize the original join point shadow.
- Instruction 5 is a call to the `afterAdviceNo` method and neutralizes the call to the `afterAdvice` method. Again, we need a receiver object in order to execute the `afterAdviceNo` method.

Because of the additional instruction, the `nullOperation` method does not make a perfect neutralization. The results of this benchmark are therefore a little bit too low.



Listing 6-9 – `operation` and `nullOperation` methods for the AspectJ/Association Aspects benchmark

Benchmark 3:

In order to implement the benchmark case in ALIA4J an attachment is deployed. The attachment is shown in figure 6.4. The `PerTupleContext` follows an implicit instantiation strategy whereby an object of `ALIA_2_1` will created to serve as aspect instance. The pointcut matches a call to join point shadow `method_ALIA_2_1` that is defined in the `operation` method of the benchmark harness. The advice runs afterwards and performs a call to a static `advice` method that increases a trivial counter. The `PerTupleContext` depends on the `CallerContext`.

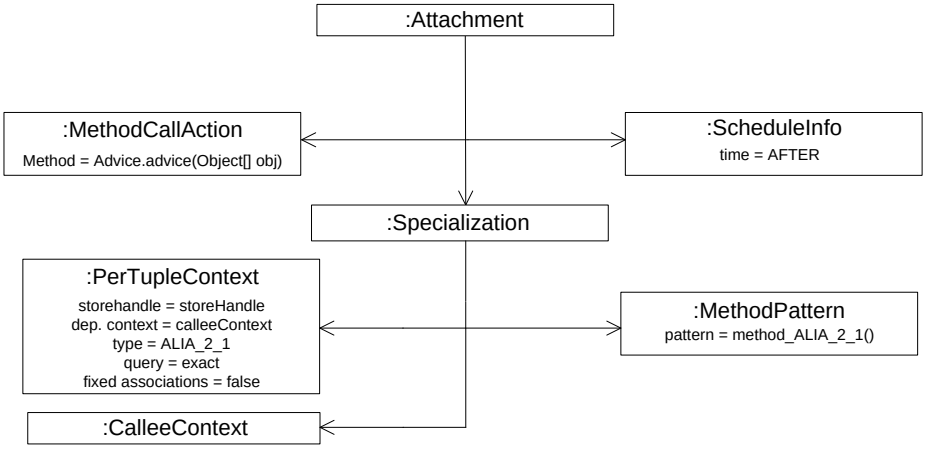
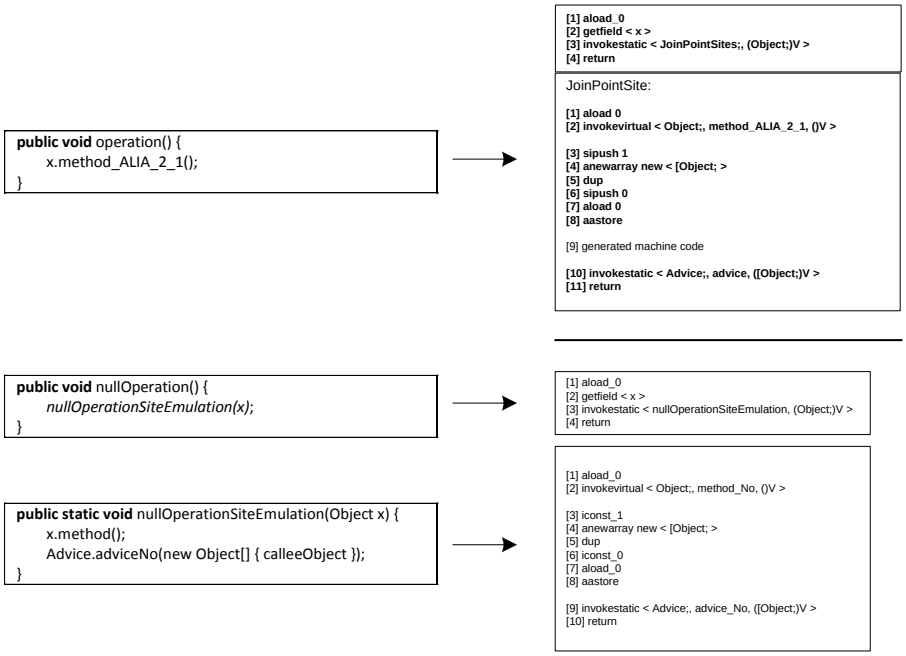


Figure 6-4 – Object diagram of the attachment following an implicit instantiation strategy

Listing 6.10 shows on the left-hand side how the `operation` and `nullOperation` method are defined for this benchmark case. The corresponding bytecode instructions are shown on the right-hand side, the instructions in bold must be neutralized.



Listing 6-10 – `operation` and `nullOperation` methods for the ALIA4J benchmark

Benchmark 4:

In benchmark 4 the policy is measured in ALIA4J whereby the `PerTupleContext` follows an explicit instantiation strategy and the association in the StoreHandle entity are fixed. An object diagram of the attachment is shown in figure 6.5.

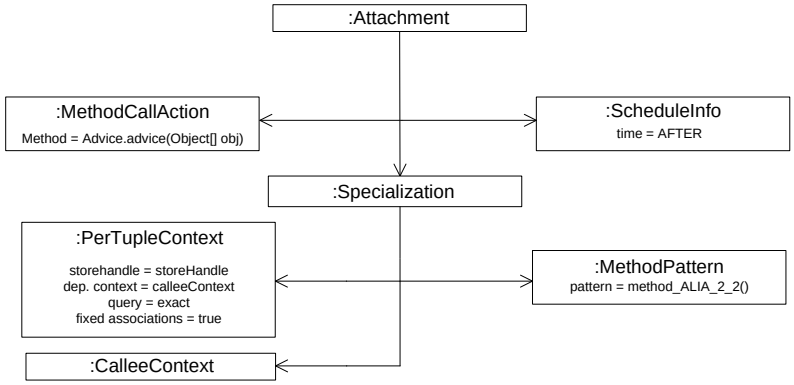
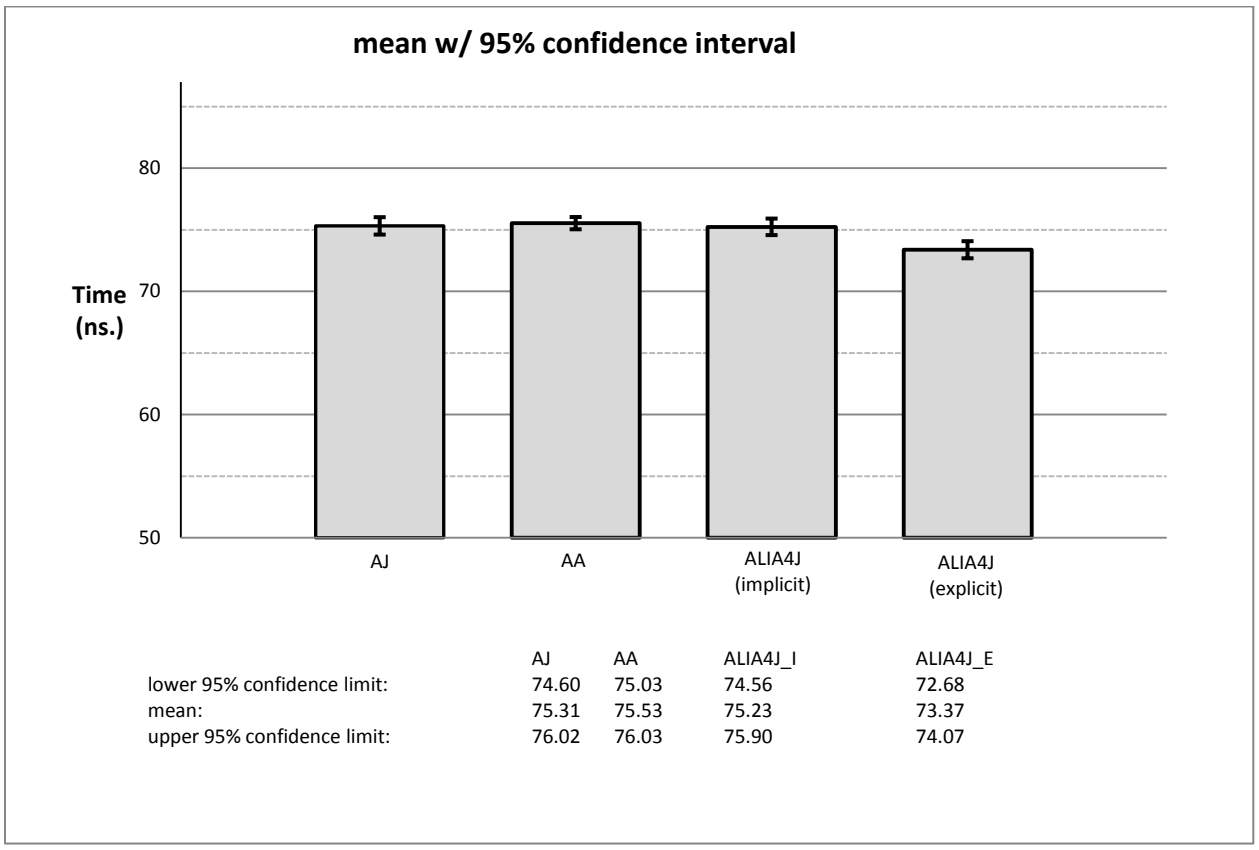


Figure 6-5 – Object diagram of the attachment following an explicit instantiation strategy

Figure 6.6 shows the elapsed time for the four benchmark cases.

Conclusion: ALIA4J_I is on average almost even fast as AspectJ and Association Aspects. ALIA4J_E is on average a little bit faster than ALIA4J_I. The confidence intervals do overlap thus it cannot be concluded that the differences seen in the mean values are not due to random fluctuation in the measurements. For this benchmark ALIA4J cannot outperform AspectJ and Association Aspects. The reason for this is that both implementations depend on context values that are used by aspect-instantiation policy to retrieve the aspect instance. Both implementations use a different strategy on how those context values are stored. In AspectJ and Association Aspects an aspect can only be deployed at compile-time. This allows AspectJ and Association Aspects to determine at compile-time which join point shadows are going to be advised. The class that contains a join point shadow that is potential being advised is therefore equipped with an additional hidden field that can hold an aspect instance. When at run-time a join point shadow is reached and advice needs to be invoked, the aspect instance can directly be retrieved by referring to the additional field. In ALIA4J attachments are deployed at run-time and thus ALIA4J cannot determine at compile-time which classes are affected. No additional field is therefore available but instead a hash table is used to determine the aspect instance.



	AJ	AA	ALIA4J_I	ALIA4J_E
lower 95% confidence limit:	74.60	75.03	74.56	72.68
mean:	75.31	75.53	75.23	73.37
upper 95% confidence limit:	76.02	76.03	75.90	74.07

Figure 6-6 – Mean and confidence interval of the per-target policy benchmark

6.3.3 Benchmarking association aspect policies

This section benchmarks the association aspect policy. Listing 6.11 shows aspect `AA_3_1` that implements this policy in Association Aspects. The following table shows how the benchmark is defined and how the four properties are configured to represent the per-group policy.

	benchmark 1 (AA)	**benchmark 2 (ALIA4J)**	**benchmark 3 (ALIA4J)**
implicit / explicit	explicit	explicit	explicit
sensitive / insensitive	sensitive	sensitive	sensitive
exact / range	range	range	range
fixed / not-fixed	not-fixed	not-fixed	fixed

The aspect contains a constructor whereby two objects must be provided by the client. The aspect creates on line 4 an association between the two objects. Furthermore one pointcut-and-advice pair is present. The pointcut is defined on line 7 – 10 and matches a call in the `operation` method of the benchmark case. In addition to earlier discussed aspects, this pointcut binds the target object, i.e. the callee object, and requires that an association must exist. The advice runs afterwards and performs a call to a static `advice` method that increases a trivial counter.

```
1   public aspect AA_3_1 perobjects(X, X) {
2
3       public AA_3_1(X left, X right) {
4           associate(left, right);
5       }
6
7       pointcut p1():
8           call(void *.*.method_AA_3_1()) &&
9           target(target) &&
10          associated(target, *);
11
12      after(X target): p1(target) {
13          Advice.advice(this);
14      }
15  }
```

Listing 6-11 – Aspect AA_ 3_1

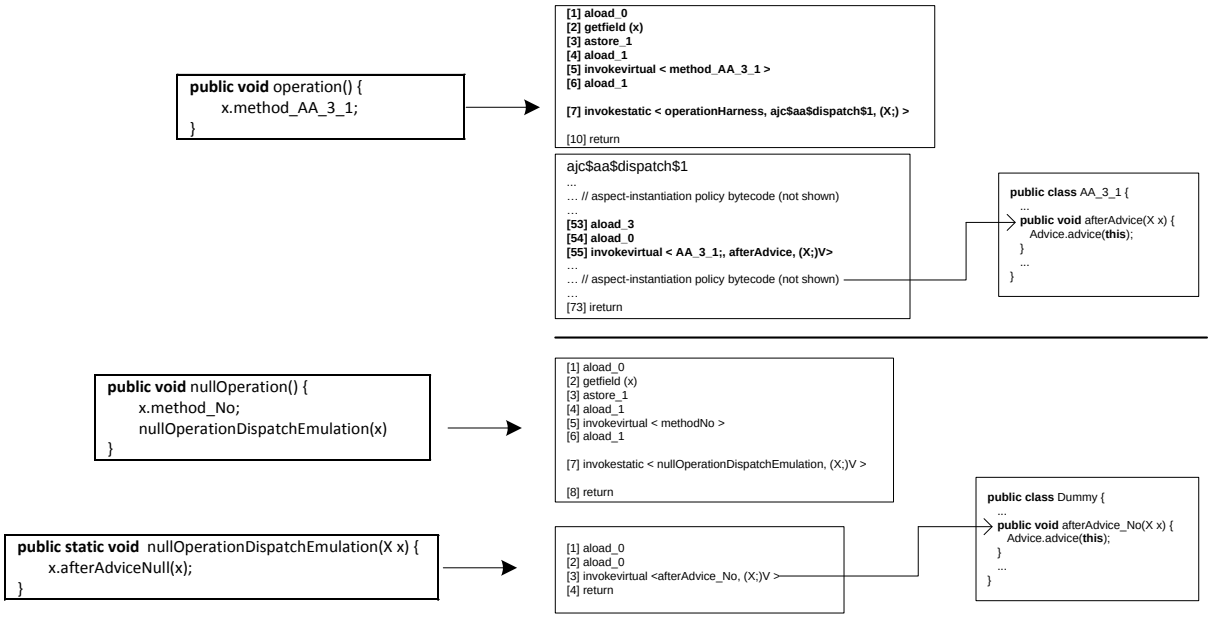
Listing 6.12 shows on the left-hand side how the `operation` and `nullOperation` method are defined for this benchmark case. The corresponding bytecode instructions are shown on the right-hand side, the instructions in bold must be neutralized.

The instructions in the `nullOperation` method are as follows:

- Instructions 1 – 6 neutralize the original join point shadow and store the caller object in a local variable.
- Instruction 7 is a static call to the `nullOperationDispatchEmulation` method and neutralizes the static call to the `ajc$aa$dispatch$1` method.

The instructions in the `nullOperationDispatchEmulation` method are as follows:

- Instructions 1 – 3 call the `afterAdvice_No` method in order to neutralize the call to the `afterAdvice` method.



Listing 6-12 – `operation` and `nullOperation` methods for the Association Aspect benchmark

In order to implement the benchmark in ALIA4J two attachments are deployed. In the first attachment the `StoreHandle` entity is fixed whereas in the second attachment the `StoreHandle` entity is not fixed. An object diagram of both attachments is shown in figure 6.7; the first attachment is shown at the left-hand side, the second attachment at the right-hand side.

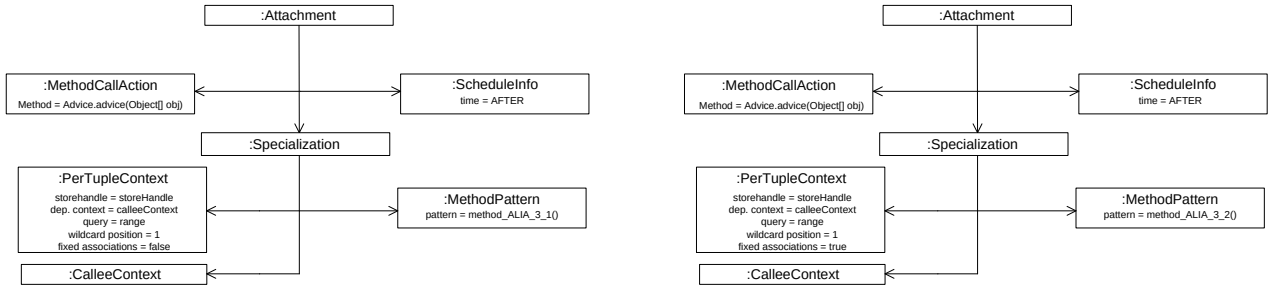


Figure 6-7 – On the left: attachment whereby associations in the `StoreHandle` entity are not fixed. On the right: attachment whereby associations are fixed.

Figure 6.8 shows the elapsed time for the three benchmark cases.

Conclusion: ALIA4J_F is on average 12.6 times as fast as Association Aspects. The confidence intervals of the results do not overlap and this means that there is no evidence to suggest that there is no statistical difference. ALIA4J_NF is on average 12.3 times as fast as Association Aspects.

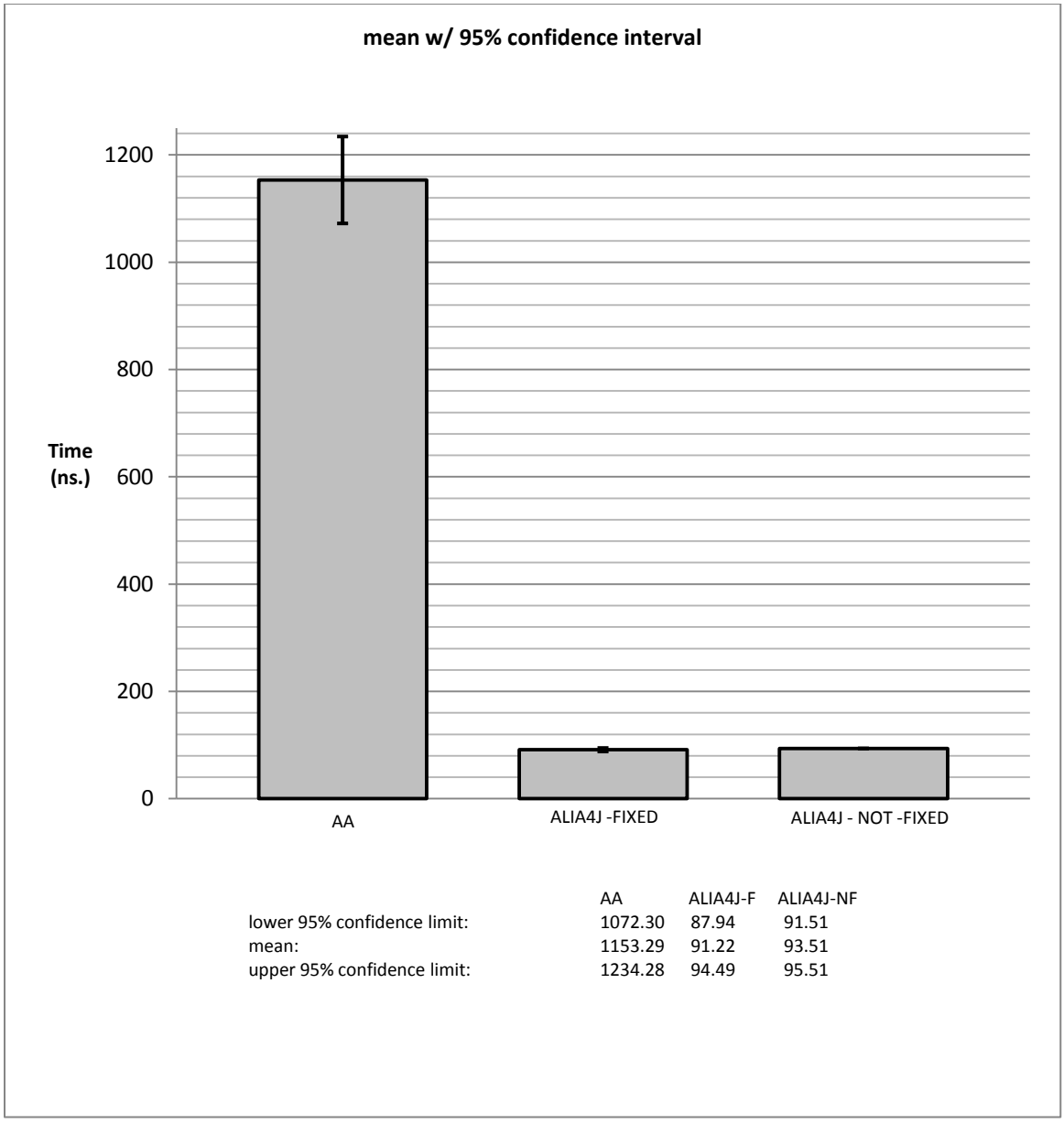
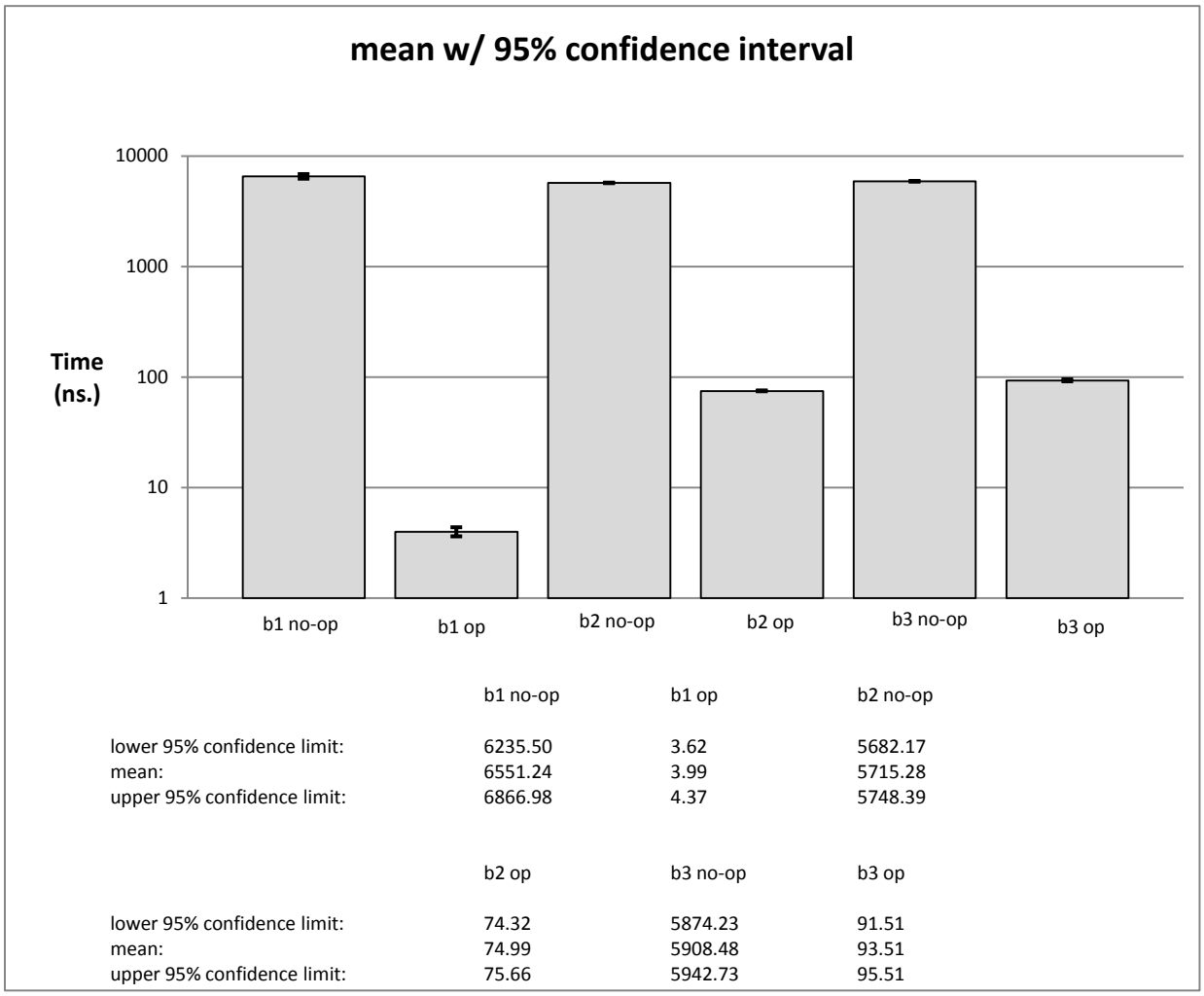


Figure 6-8 – Mean and confidence interval of the association aspect benchmark

6.3.4 Benchmarking policies in the unoptimized PerTupleContext against policies in the optimized PerTupleContext.

In this section we benchmark all the discussed aspect-instant policies in unoptimized version of the `PerTupleContext` and compare it with the results of the optimized version of the `PerTupleContext`. Figure 6.9 shows the elapsed time for the six benchmark cases.

	benchmark 1	benchmark 2	benchmark 3
implicit / explicit	implicit	implicit	explicit
sensitive / insensitive	insensitive	sensitive	sensitive
exact / range	exact	exact	range
fixed / not-fixed	not-fixed	not-fixed	not-fixed



	b1 no-op	b1 op	b2 no-op
lower 95% confidence limit:	6235.50	3.62	5682.17
mean:	6551.24	3.99	5715.28
upper 95% confidence limit:	6866.98	4.37	5748.39

	b2 op	b3 no-op	b3 op
lower 95% confidence limit:	74.32	5874.23	91.51
mean:	74.99	5908.48	93.51
upper 95% confidence limit:	75.66	5942.73	95.51

Figure 6-9 – Mean and confidence interval of the `PerTupleContext` benchmark

Note that the Y-axis is logarithmic. Conclusion: As can be seen, the performance of aspect-instantiation policies in the unoptimized `PerTupleContext` is poorer than that of policies in the optimized `PerTupleContext`.

Chapter 7

Conclusions & Future Work

In this thesis we have researched the optimization of aspect-instantiation strategies in the JIT-compiler. The aspect-instantiation strategies define the rules how aspect instances are stored, looked-up and instantiated. It is feasible that an aspect-instantiation strategy is fast. This is because an aspect instance serves as the context for executing one or more advices just as an object serves as context for executing a method. If advice is executed often then the aspect-instantiation strategy is also executed often. In languages like AspectJ and Association Aspects the aspect-instantiation strategies are woven at compile-time with the base program to form one final program. When at run-time the bytecode of the program is compiled to machine code by the JIT-compiler, it cannot distinguish what part of the program corresponds to the original program and what part corresponds to the aspect-instantiation policy. Or in other words: semantics of the aspect-instantiation policies are lost and this causes that the JIT-compiler cannot produce machine code that is optimized according to semantics of an aspect-instantiation policy.

In order to preserve semantics we used the ALIA4J approach. The `PerTupleContext` entity is provided as an entity that covers three commonly used instantiation policies. However, this `PerTupleContext` entity implements semantics of the generalized of at a high-level of abstraction, i.e. in the so-called `getObjectValue` method. The high-level of abstraction approach works perfectly for experimentation purposes but not for optimization purposes because no optimized machine code can be generated.

The objective of this thesis is to generate machine code that is optimized according to the aspect-instantiation policy. We have therefore created a new `PerTupleContext` entity that inherits from the existing `PerTupleContext`. The difference between two entities is that the former one uses the low-level of abstraction style whereas the latter uses the high-level of abstraction style. The new `PerTupleContext` is able to interfere with the JIT-compiler. When the JIT-compiler encounters the new `PerTupleContext` entity then hands over control of machine code generating by calling the `generateASM` method. This method then generates machine code according to the semantics of the generalized model. In this way semantics of aspect-instantiation policies are used until the latest moment in time resulting into more efficient machine code.

We have defined eight optimization strategies that are used by the `PerTupleContext` entity to generate machine code that is optimized according to an aspect-instantiation policy. The optimization strategy is determined on the values of four properties of the generalized model. The generation of machine code occurs at run-time and this allows the `PerTupleContext` to make use of run-time information.

In order to assess the effectiveness of optimization strategies we have defined a series of benchmarks. With a benchmark platform we have benchmarked several aspect-instantiation policies and compared them against implementations in AspectJ and Association Aspects. It turns out that ALIA4J outperforms AspectJ and Association Aspect with the singleton and per-group

instantiation policy but does not perform significantly better with the per-target instantiation policy.

In future work the implementation of the `PerTupleContext` can be improved. The current implementation only supports the baseline compiler of the JikesRVM. An implementation for the optimizing compiler could be a next step. Another idea for future work is to incorporate aspect instance tables [**16**] in optimization strategies that are sensitive on context values. Instead of referring to an internal hash table, we can modify an object's TIB or header with a field that can hold an aspect instance.

Bibliography

[1] Edsger W. Dijkstra, "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, pp. 60-66, 1982.

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

[3] H. Masuhara and G Kiczales, "Modeling Crosscutting in Aspect-Oriented Mechanisms," pp. 2-28, 2003.

[4] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," *In AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development.*, pp. 26-35, 2004.

[5] J. Bonér, "Aspectwerkz - dynamic aop for Java," *[Online]. Available: http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf*, 2004.

[6] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya, "Association aspects," *in AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pp. 16–25, 2004.

[7] A. Loker, "A Generalised Model for Instantiation Policies," 2010.

[8] C. Bockisch, A. Sewe, M. Mezini, and M. Akşit, "An Overview of ALIA4J: An Execution Model for Advanced-Dispatching Languages.," *Online-Edition: http://dx.doi.org/10.1007/978-3-642-21952-8_11*, 2011.

[9] C. Bockisch, "An Efficient and Flexible Implementation of Aspect-Oriented Languages," *PhD Thesis*, 2009.

[10] B. Alpern et al., "Implementing jalapeño in Java," *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 314--324, 1999.

[11] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann, "Virtual machine support for dynamic join points," *roceedings of the 3rd international conference on Aspect-oriented software development*, pp. 83--92, 2004.

[12] D. Frampton et al., "Demystifying magic: high-level low-level programming," *In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, vol. ACM, pp. 81--90, 2009.

[13] P. McGachey, "An Improved Generational Copying Garbage Collector," *Master Thesis*, 2005.

[14] S. Drossopoulou and S. Eisenbach, "Manifestations of Java Dynamic Linking - an approximate understanding at source language level," 2002.

[15] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pp. 57-76, 2007.

[16] M. Haupt and M. Mezini, "Virtual Machine Support for Aspects with Advice Instance Tables," *L'Objet*, 2005.

[17] L. David, "Measuring Computer Performance: A Practitioner's Guide," 2000.