Fachgebiet Programmierung eingebetteter Systeme
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

Optimized Aspect Execution Mechanisms inside Virtual Machines for Embedded Systems

vorgelegt von
Diplom-Informatikerin
Christine Hundt

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Hans-Ulrich Heiß Technische Universität Berlin
Berichtende:	Prof. Dr. Sabine Glesner Technische Universität Berlin
Berichtender:	Prof. Dr. Robert Hirschfeld Hasso-Plattner-Institut, Universität Potsdam

Tag der wissenschaftlichen Aussprache: 4. November 2011

Berlin 2012
D 83

Abstract

The rapidly growing mobile market has stimulated the demand for more and more complex custom applications for embedded mobile devices, for example, smart phones. To manage this complexity and, at the same time, to keep the time to market small, advanced software engineering methods have to be applied. *Aspect-oriented programming* (AOP) provides advanced modularization and abstraction mechanisms. The main advantages of this concept are improved maintainability, reusability, and extensibility of applications. Furthermore, *dynamic* AOP can be used to implement the dynamic adaptation of mobile device applications to changing contexts, like the location. However, the overhead introduced by the additional abstraction mechanisms limits the applicability to embedded mobile devices because they have limited resources (CPU, memory) compared to desktop PCs.

To overcome this problem, we present a set of optimizations that significantly reduce the overhead of common AOP mechanisms and, finally, make AOP applicable for embedded mobile devices. The foundation of our work is a thorough analysis of the overhead that is typically generated by the realization of AOP mechanisms. The key idea of our approach is a deep integration of AOP mechanisms into the virtual machine. To this end, we shift mechanisms like the registration of activated aspects to the level of the JVM. Furthermore, we optimize the execution of AOP programs by introducing caching mechanisms and specialized bytecode instructions that are tailored for the execution of AOP mechanisms. Moreover, we analyze AOP-specific semantic code properties in order to develop optimizations that utilize these AOP-specific semantic information and that exploit typical AOP usage schemes. In addition to the AOP optimizations, we realize an efficient dynamic aspect deployment mechanism. We apply our optimizations to the Java-based aspect-oriented programming language ObjectTeams [HHM07] by extending the extremely small and portable JamVM [Lou] Java virtual machine.

To evaluate our approach, we execute micro benchmarks, investigate the effect of our optimizations on a real-world application, and finally discuss the transferability of our optimizations to other approaches. Our evaluation shows a considerable performance gain for the *aspect activation* and the *aspect execution* of ObjectTeams. In particular, we demonstrate that our optimizations improve the performance of commonly used AOP mechanisms by up to 90%. At the same time, we reduce the code size of the adapted classes, which is also important for small devices. Finally, with our case study, namely the OTPong game application, we show that our approach is capable of significantly optimizing the execution time of real-word applications. Our main contribution is a significant reduction of the overhead of high-level AOP constructs, which is also demonstrated by the results of our experiments. The success of the optimizations provides evidence that advanced high-level abstraction techniques like AOP can be efficiently used in embedded mobile devices. Furthermore, our work shows that efficient dynamic aspect deployment can be supported on the level of the JVM. This substantially enhances the dynamic capabilities of ObjectTeams.

Zusammenfassung

Der rasant wachsende Mobilgerätemarkt verlangt nach immer komplexeren Anwendungen für eingebettete Mobilgeräte, wie Smartphones. Um diese Komplexität beherrschbar zu machen und gleichzeitig die Produktzyklen möglichst kurz zu halten, ist es nötig fortgeschrittene Softwareentwicklungsmethoden anzuwenden. Die *aspektorientierte Programmierung* (AOP) stellt fortgeschrittene Modularisierungs- und Abstraktionsmechanismen zur Verfügung. Die wichtigsten Vorteile dieses Konzeptes sind die verbesserte Wartbarkeit, Wiederverwendbarkeit und Erweiterbarkeit von Anwendungen. Darüber hinaus ist dynamische Aspektorientierung dazu geeignet die dynamische Anpassung von mobilen Anwendungen an wechselnde Kontexte, wie z.B. den Standort, zu realisieren. Allerdings schränkt der Overhead, der durch die zusätzlichen Abstraktionsmechanismen entsteht, die Anwendbarkeit auf eingebettete Mobilgeräte ein, da diese im Vergleich zu Desktop PCs beschränkte Ressourcen (CPU, Speicher) aufweisen.

Um dieses Problem zu bewältigen, präsentieren wir eine Reihe von Optimierungen, welche den Overhead von typischen AOP-Mechanismen signifikant reduzieren und damit AOP für eingebettet Mobilgeräte anwendbar machen. Die Grundlage unserer Arbeit ist eine gründliche Analyse des Overheads, der typischerweise durch die Realisierung von AOP-Mechanismen entsteht. Die Kernidee unseres Ansatzes ist eine tiefe Integration der AOP-Mechanismen in die virtuelle Maschine. Zu diesem Zweck verschieben wir diese Mechanismen, wie etwa die Registrierung von aktiven Aspekten, auf die Ebene der JVM. Weiterhin optimieren wir die Ausführung von AOP Programmen durch die Einführung von Cache-Mechanismen, sowie spezialisierten Bytecode-Instruktionen, die auf die Ausführung von AOP-Mechanismen zugeschnitten sind. Darüber hinaus analysieren wir AOP-spezifische semantische Code-Eigenschaften und entwickeln darauf aufbauend Optimierungen, die sich diese Eigenschaften und typische AOP Benutzungsschemata zunutze machen. Zusätzlich zu den AOP-Optimierungen realisieren wir einen effizienten dynamischen Aspekt-Deploy Mechanismus. Wir wenden unsere Optimierungen auf die Java-basierte aspektorientierte Programmiersprache ObjectTeams [HHM07] an, indem wir die extrem kleine und für eingebettete Systeme portierte JamVM [Lou] erweitern.

Um unseren Ansatz zu evaluieren führen wir Mikro-Benchmarks aus und untersuchen den Effekt unserer Optimierungen auf eine echte Anwendung. Unsere Evaluierung zeigt eine erhebliche Performanz-Steigerung für die *Aspekt-Aktivierung* und die *Aspekt-Ausführung* von ObjectTeams. Unsere Optimierungen verbessern die Performanz von häufig verwendeten AOP-Konstrukten um bis zu 90%. Gleichzeitig verringern wir die Code-Größe adaptierter Klassen, was ebenfalls wichtig ist für kleine Geräte. Schließlich zeigen wir mit unserer Fallstudie, der Spieleanwendung OTPong, dass unser Ansatz in der Lage ist, die Ausführungszeit von echten Anwendungen signifikant zu optimieren. Unser wichtigster Beitrag ist die signifikante Reduzierung des Overheads von höheren AOP-Konstrukten, die wir durch unserer Experimente untermauern konnten. Der Erfolg der Optimierungen zeigt, dass fortgeschrittene höhere Abstraktionstechniken wie AOP effizient auf eingebetteten Mobilgeräten verwendet werden können. Außerdem zeigt unserer Arbeit, wie effizientes dynamisches Aspekt-Deployment auf der Ebene der JVM umgesetzt werden kann. Dadurch konnten wir die dynamischen Fähigkeiten von ObjectTeams maßgeblich erweitern.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit im DFG Aktionsplan Informatik Verifikation und Optimierung bei der Ubersetzung höherer Programmiersprachen, geleitet von Prof. Dr. Sabine Glesner. Bei ihr möchte ich mich herzlich für die Betreuung meiner Arbeit bedanken und dafür, dass sie mich immer unterstützt und ermutigt hat. Außerdem danke ich meinem Zweitgutachter Prof. Dr. Robert Hirschfeld für seine Unterstützung und die konstruktiven Anmerkungen von ihm und seiner Arbeitsgruppe am Hasso-Plattner-Institut Potsdam.

Bei meine Kollegen bedanke ich mich herzlich für die nette Arbeitsatmosphäre und den tollen Promotionsfilm. Stephan Herrmann ist maßgeblich dafür verantwortlich, dass ich den Weg des wissenschaftlichen Arbeitens eingeschlagen habe. Paula Herber und Lars Alvincz haben mir immer mit Rat und Tat und viel Fröhlichkeit zur Seite gestanden. Außerdem möchte ich meinen Diplomanden Daniel Stöhr und Julian Bischof danken, deren Arbeiten zum Erfolg dieser Arbeit beigetragen haben.

Schließlich möchte ich meiner Familie dafür danken, dass sie immer für mich da war und mich unterstützt hat. Meinen Bruder Andreas danke ich für die Hilfe beim Cross-Kompilieren.

Contents

1 Introduction

During the last years, embedded mobile devices like hand-helds and mobile phones have become much more popular and have gained increasing computational power. This development has also stimulated the demand for more and more complex custom applications to be run on these devices. At the same time, such mobile devices are ruled by mass-market laws. This means that time to market as well as price per unit are critical success factors. To keep time to market small, software engineering methods are necessary that facilitate reuse, adaptability and extensibility of software systems, rendering low-level programming languages like C as no longer sufficient for this growing complexity. As in the area of desktop applications, advanced concepts for abstraction and modularization as e.g. object- or aspect-orientation are needed. At the same time, the hardware of such embedded devices will always be limited compared to desktop PCs, not least to keep their price per unit small. Hence, many software engineering methods are not directly applicable because they require too much memory space and/or computation time. This conflict between hardware restrictions and the need for more advanced software engineering concepts is characteristic for embedded systems. We appraise aspect-oriented programming (AOP) [KLM$^+$97] to be highly qualified to modularize software adequately in order to achieve the above mentioned goals of maintainability, extensibility and reusability, also in the context of embedded devices.

In this work, we address the problem of allowing the advanced software engineering mechanisms of aspect-oriented programming (AOP) to be applied to applications running on embedded mobile devices. This is a challenge because such devices have limited resources and the advanced modularization mechanisms of AOP typically cause increased demands on computation power and memory, by extra indirections for dispatching to the aspect code, and by additional infrastructural code. AOP is an extension of the object-oriented programming paradigm that facilitates a better *separation of concerns*, i.e., for the separate implementation of core functionality from system-specific concerns representing e.g. configuration decisions that may come up at many places in a program. A subsequent weaving process combines functional and aspect code. These advantages of aspect-orientation do not come for free, as they introduce a significant overhead. The main goal of our work is to reduce this overhead and make AOP applicable for embedded devices.

To make AOP applicable for embedded device applications, we reduce the mentioned overhead by optimizing the aspect-oriented execution mechanisms. We solve the efficiency problems by adapting the object-oriented execution mechanisms to meet the particular needs of aspect-oriented programs. In this work, we focus on aspect-oriented programming languages extending the object-oriented language Java. In these languages, aspect-oriented constructs are typically mapped to Java bytecode which is then executed on a standard Java virtual machine (JVM). This approach offers only restricted optimization potential because it is restrained entirely on the limited operations of the Java bytecode. All potential optimizations can only be applied to the bytecode level and not directly target the performance of the aspect-oriented execution mechanism. In contrast, approaches like [Hau06, HM05] investigate the extension of the executing JVM by aspect-oriented features. It turns out that this approach offers much better optimization gains. Supporting aspect dispatch mechanisms at the level of the JVM is more promising as this facilitates efficient weaving policies tailored to the specific needs of static and dynamic weaving in systems with only limited memory and computation power. Therefore, our approach extends the execution mechanisms at the level of the Java virtual machine in order to optimize and flexibilize the AOP execution. Additional challenges for the efficiency arise from the demand for dynamic weaving, when aspects need to be added at run-time, e.g. to update software or to adapt software to changing contexts. The adapted execution environment of our approach accounts an efficient realization of such capabilities.

We require our solution to facilitate better *maintainability*, *extensibility* and *reusability* of the embedded software. Moreover, it shall be able to cope with an increased *variability* among the various static and dynamic versions of an application. Static variability arises for example when the same application is running on similar, yet different platforms of a product line. Dynamic variability comes up when the device changes its context and the application running on it needs to adapt to new boundary conditions. Our approach takes into account the need for additional variability in the context of mobile devices. Finally, our approach is required to be *efficient* with respect to memory usage and execution time in order to meet the tight constraints of embedded hardware.

We implemented our approach in the run-time environment of the aspect-oriented programming language ObjectTeams/Java (OT/J) [HHM07]. OT/J is an extension of the Java programming language, performing the weaving of aspects at class loading time. The aspect dispatch logic is entirely realized at the level of Java bytecode, which is eventually executed by a JVM. In our work, we are extending the extremely small and portable JVM *JamVM* [Lou] such that different parts of the aspect execution mechanism for OT/J are optimized. We evaluated our approach with micro benchmark and with an aspect-oriented game application that we executed on an Intel Core2 desktop PC as well as on the Linux-based smart phone FreeRunner [Fre]. The results of our experiments show that we significantly reduce the execution time with our approach and, hence, improve run-time performance.

Our main contribution in this work is a significant reduction of the overhead of high-level AOP constructs. We achieve this by a deep integration of AOP mechanisms into the virtual machine. The success of the optimizations provides evidence that advanced high-level abstraction techniques like AOP can be efficiently used in embedded mobile devices. Furthermore, our work shows that efficient dynamic aspect deployment can be supported on the level of the JVM.

This work is structured as follows: Chapter 2 gives an overview of the foundations of this work. In Chapter 3, we discuss related work. Chapter 4 and Chapter 5 constitute the main part of our work. Here, we detail our approach of optimizing the execution environment of the JVM for aspect-oriented program execution. In Chapter 4, we identify sources of overhead for AOP mechanisms and develop optimizations to reduce this overhead. Moreover, we realize an efficient dynamic aspect deployment mechanism on the level of the JVM. Chapter 5 investigates aspect-specific semantic conditions of code. Here, we propose optimizations that utilize these conditions to further optimize the execution of AOP mechanisms. In Chapter 6, we explain the implementation of our optimizations and its integration with the existing execution environment of OT/J. Chapter 7 evaluates our approach and presents our experimental results. In Chapter 8, we conclude and outline future work.

2 Background

In this chapter, we describe the background on which this work is based. We introduce the underlying concepts of aspect-oriented programming and the relevant details about virtual machines.

2.1 Aspect-oriented Programming

A common ambition of all programming paradigms is to maximize the *separation of concerns* in the developed applications. Ideally, each concern should be implemented in its own module. The better a program is modularized, the better it can be understood, maintained, extended, and reused. In the object-oriented paradigm, this is achieved by encapsulating concerns in classes and methods. Hierarchical structures and inheritance between classes provide for reuse and flexibility. However, it is often not possible to clearly separate every concern of an application, simultaneously. When the structure of the application is determined, some concerns have to be implemented "in between". Such concerns are called *crosscutting concerns* [KLM+97] because they crosscut the overall application design. Thus, in a purely object-oriented implementation, crosscutting concerns are *tangled* with the base application code. Often these concerns are furthermore *scattered* around multiple locations. Both is problematical and contradicts the separation of concerns: tangling reduces the quality of the base code and scattering reduces the quality of the crosscutting concern implementation. This situation is illustrated on the left hand side of Figure 2.1: the gray concern and the shaded concern crosscut the implementation of the white base module.

Aspect-oriented programming (AOP) [KLM+97, RFSC04] has been developed to solve this problem by facilitating the modularization of crosscutting concerns. With AOP, crosscutting concerns are implemented in separate *aspect modules* (*aspects* for short). The integration with the base code is realized via a binding mechanism that makes it possible to specify the connections of the aspect code and the base code. The right part of Figure 2.1 shows a separation of the two crosscutting concerns from the base module together with aspect bindings, indicated by the arrows. Now, every single concern is appropriately

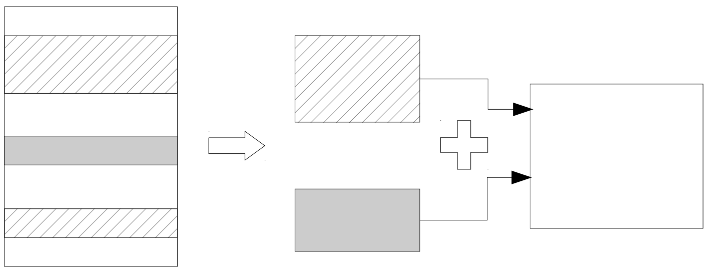


Figure 2.1: Separation of Concerns with AOP

modularized. Usually the aspects functionality is *unanticipated* (neither prepared nor expected) by the base code and the binding is declared within or in addition to the aspect modules. Several AOP languages, like AspectJ [ajH] also support a *quantification* mechanism, which uses a wild-card-like notation to describe multiple locations for aspect bindings.

In the following, we take a more detailed look at the concepts and mechanisms of AOP and introduce the aspect-oriented programming language ObjectTeams, which will be the target of our optimizations. Further, we discuss different kinds of aspect-oriented weaving mechanism, and, finally, introduce the concepts and mechanisms of dynamic AOP.

2.1.1 AOP Mechanisms

In the aspect-oriented programming paradigm, the core functionality is implemented in *base* modules, while the crosscutting concerns are defined in separate *aspect* modules (see Figure 2.2). These aspects define the crosscutting functionality together with *aspect bindings*, specifying the points in the execution of the base program (*join points*) at which they should be executed.

There is no closed definition of aspect-orientation. This results in a large variety of aspect-oriented programming languages, e.g., AspectJ [ajH], JBossAOP [JBo], AspectC++ [SLU05], and ObjectTeams [otH]. Most of these languages

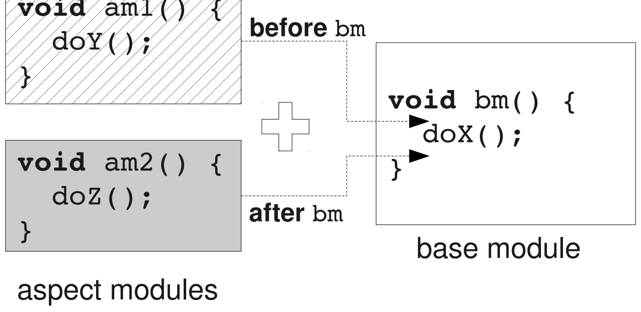


Figure 2.2: AOP Concepts

are extensions of existing object-oriented programming languages like Java [GJSB05]. A common mechanism is the adaptation of base methods by aspect methods. This adaptation can be specified in different ways. The exact execution point can be stated as *before*, *after*, or *instead* of the base method. In the example of Figure 2.2, the aspect method `am1` is defined to be executed before the method `bm` of the base module. The aspect method `am2` is executed after the base method, respectively. The *before* and the *after* variant facilitate pure additions to the base code while in the *instead* case, more intervention is possible. The base method can be completely replaced by the aspect method, which can be used to drastically change the program semantics at this point. But usually, the original base method can also be called from within the replacing aspect method. In doing so, the arguments of the original call can also be adapted. This is less invasive than completely omitting the base method, as it does not eliminate the subsequent control flow.

Several AOP languages allow for dynamic *activation* (and *de*activation) of aspects. This also applies to *ObjectTeams* (OT/J, see 2.1.3), which is in the focus of this work. Moreover, the adaptation can be constrained by further conditions like the execution context, the active thread, or the dynamic activation state of an aspect. Dynamic activation of aspects can be used to support dynamic and context-sensitive changes in program behavior. Depending on certain constraints (like the location, or the battery power) an application can run in different modes realized by different aspects activated at different times. If it is even possible to add additional aspects during the run-time of a program, this is called *dynamic AOP*, as introduced in Section 2.1.4.

2.1.2 Aspect Weaving

While aspect bindings specify the points at which an aspect should be effective, we need a mechanism that actually forces the execution of the corresponding aspect methods. This *aspect weaving* mechanism inserts calls to aspect methods as well as various infrastructural elements in terms of fields, methods, or classes. Aspect weaving usually involves program transformation techniques and can be done by using different approaches, as discussed in [PZ03].

Most weaving approaches translate the aspect-oriented parts of a program by mapping them to constructs of the programming language the AOP language is based on, for example Java. This can be done at different levels, as summarized in Table 2.1.

Pre-compile-time weaving operates on source code level by performing source-to-source transformation, as, e.g., applied by AspectC++ [SLU05]. The woven target code conforms to the source code of the base language and can be compiled by a standard compiler, afterwards.

Compile-time weaving requires a customized compiler. It takes source or bytecode and generates a woven executable. This can be for example Java bytecode that contains the necessary aspect-specific parts and that can be

Table 2.1: Aspect Weaving Overview

Moment	**Target**	**Technique**
pre compile-time	source code	source-to-source transformation
compile-time	source code / bytecode	compilation / bytecode transformation
load-time	bytecode	bytecode transformation
run-time	VM-internal object code	object code transformation

executed on every standard Java virtual machine (JVM). This, e.g., is possible with AspectJ [ajH].

Load-time weaving, as applied, e.g., by ObjectTeams [otH], intercepts and transforms the bytecode during the class loading phase of the JVM. It can be realized with bytecode transformation frameworks like JMangler [GK04] or by using the *Java Programming Language Instrumentation Services* (JPLIS) API, which is part of Java SE 5. In contrast to compile-time weaving it is not necessary to maintain different class file versions because the woven byte code is never stored on the disk.

Run-time weaving is performed during the execution of the program inside the JVM. Therefore, the VM-internal object code has to be adaptable, as, e.g., in Steamloom [BHMO04]. Because it is very complex to change the memory image of a running application, this is often realized with some kind of pre-existing hook mechanism (as detailed in Sec. 2.1.4).

(Pre-)Compile-time weaving is also called *static weaving* because it is performed before the program has been started. Load-time weaving and run-time weaving are called *dynamic weaving*, in contrast. Real dynamic weaving necessitates a dynamic deployment mechanism which facilitates adding and removing aspects at any point in time. As dynamic run-time weaving is a prerequisite for dynamic AOP, it will be evaluated in more detail in Section 2.1.4.

The later the aspect weaving is performed, the more dynamically the aspects can be added to the base program enabling more flexible context-aware adaptation of the executed applications. Unfortunately, dynamic aspect weaving suffers from more effort at run-time. Static weaving, on the other hand, increases the code size, which is also critical for small devices. Another drawback of statically inlining aspectual code is that the aspects cannot be located in later phases, which is necessary for debugging or analyzing AOP applications. Moreover, it restricts the flexibility of the AOP language because it is complex if not impossible to remove aspects later. This indicates the need for aspects remaining run-time entities during the execution of the program [MJV^{+}97].

2.1.3 ObjectTeams

Existing AOP languages share common concepts as described in Section 2.1.1 but the used terminology as well as the realization are different. We have chosen the programming language *ObjectTeams* (OT/J) [Her02b, HHM07, otH] to demonstrate the achievements of our approach. OT/J is an extension of the Java programming language, adding support for advanced collaboration-based modularization combined with aspect-oriented adaptation mechanisms. OT/J supports independent development of reusable aspects which can be a-posteriori integrated into an existing base system. This is achieved by subclassing and fully polymorphic overriding of aspect methods. In the context of this work, we focus on the aspect-oriented parts of the language and only touch other features of OT/J if needed.

In OT/J, aspect functionality is defined in *role* classes that are *played by* (bound to) individual *base* classes. To adapt the base functionality, *role methods* can be bound to corresponding *base methods* with *callin* bindings. Such binding causes the execution of the role method *before*, *after*, or instead of (indicated by OT/J keyword *replace*) the base method. In the *replace* case the original base functionality can be called by a *base-call*.

Roles are contained in *team* classes, which define a collaboration context for them. Via the surrounding teams, aspect functionality can be programmatically activated and deactivated at run-time (see [HHM07], §5). The activation of a team is necessary to enable the callin bindings of its contained roles. It can be *global* or *thread-local*. Automatic *implicit* team activation for the current thread guarantees a coherent aspectual control flow when a public method of a team or a role is called. This is significant if the role functionality is deeply integrated with the base functionality and calls from the role object to the base object necessitate subsequent callins to be activated. Implicit activation can be enabled globally or configured with annotations for individual methods and team types. Although a team in OT/J means more than just a collection of aspects in the context of this work, the term "team" is used synonymously with "aspect".

Figure 2.3 shows a small OT/J example using the *UML for Aspects* (UFA) notation introduced in [Her02a]. The role class `MyRole` is bound to the base class `MyBase`. The role method `rm` adapts the base method `bm` with a *before-callin* binding. To make the aspect functionality effective, we need to create an instance of the team class `MyTeam` and activate it by calling the `activate` method. Whenever the method `bm` is called on an instance of the class `MyBase` the role method is executed before the base method is executed. To this end, the base object is *lifted* ([HHM07], §2.3) to the same role object each time.

Aspect Inheritance in ObjectTeams

Besides the classical object-oriented inheritance between classes, in ObjectTeams additional aspect-specific inheritance relations exist.

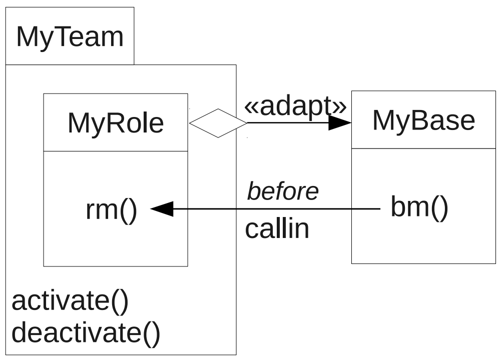


Figure 2.3: ObjectTeams Concepts

1. Aspect inheritance at team/role hierarchies: If a team extends another team, it *implicitly* inherits all its roles (cf. [HHM07], §1.3.1) together with the corresponding playedBy and aspect (callin) bindings (cf. [HHM07], §4.9.2). Bound role methods can be redefined in sub role classes.

2. Aspect inheritance at base hierarchies: Inherited base methods (also overridden ones) are also adapted by aspects bindings defined for super base classes (cf. [HHM07], §4.9.1). To achieve this, also the activated team instances are "inherited" to sub base classes.

Execution of Multiple Aspects

Each base method can be adapted by an arbitrary number of aspects. On the one hand, a single team class can define more than one aspect binding to the same base method. A typical pattern is the definition of a before- and an after-callin. This way, aspects perform some preparation and finalization tasks. In this case, first the before-callins are executed, then potential replace-callins, and finally the after-callins. If multiple callins of the same type (e.g., before) are defined for the same base method, the programmer needs to define their execution order with a precedence declaration (cf. [HHM07], §4.8).

On the other hand, multiple aspect bindings to the same base method can be defined in different team classes. When executing a base method every active team instance with aspect bindings to this method has to be considered. In this case, the order of activating instances of these team classes is relevant for the execution order of the adapting aspect methods (role methods). A team instance that is activated later has a higher "priority". This means that its before- and replace-callins are executed before those of every other active team and its after-callins are executed after the after-callins of every priorly activated team. The same applies if multiple instances of the same team class are activated. Figure 2.4 illustrates the execution order for different callins of multiple active team instances. The ObjectTeams source code of this example is shown in Listing 2.1. The team instances `t1`, `t2`, `t3`, and `t4` have been activated in this order. As a consequence, `t4` has the highest priority regarding execution of its aspect functionality when the adapted base method is called (`bm call`). First `t4`'s before-callin is executed and subsequently its replace-callin, which proceeds the execution with a base-call. The next team defining before- or replace-callins is `t2`. After executing `t2`'s replace-callin and the base-call, the original base method code is executed. Now, after-callins of the active team instances are executed corresponding to the activation order. Initially, the team instance `t1` is considered, as it has been activated first. The latest activated team instance `t4` has the "privilege" to have its after-callins executed at the very end.

Note that the base-call of replace-callins is not only necessary for the original base method to be executed. Also the execution of the remaining (earlier activated) team instances depends on the base-call. This is illustrated in Figure 2.5. The replace-callin of team instance `t4` now omits the base-call. This interrupts the execution flow we have observed for Figure 2.4. The aspect functionality of the earlier activated team instances `t3`, `t2`, and `t1` is skipped, as well as the original base method functionality.

When we adapt the realization of the AOP functionality, in order to optimize its execution, we have to guarantee the semantics of the execution order of aspects.

Implementation of OT/J

The programming language ObjectTeams is an implementation of the ObjectTeams programming model. It is a purely additive extension of the Java programming language. In the following the term ObjectTeams is also used when referring to the language implementation. The toolchain used to create and run OT/J programs is shown in Figure 2.6. To facilitate the development of OT/J applications at a high level of convenience and productivity, an extension of the Eclipse Java development tools (JDT) [JDT], the *Object Teams Development Tooling* (OTDT), is available. As illustrated in Figure 2.6, the OTDT compiles the OT source code to class files with Java bytecode, which is augmented with OT-specific bytecode attributes (cf. [HH10]). These attributes

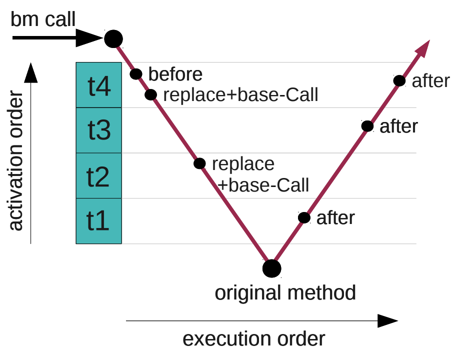


Figure 2.4: Callins with base-calls

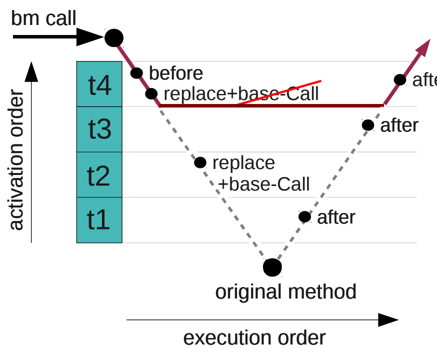


Figure 2.5: Missing base-call interrupts Execution

Listing 2.1: ObjectTeams Code for the Multiple Activation Example

```
public class B {
  void bm() {...}
}

public team class T1 {
  protected class R1 playedBy B {
    void rm() {...}
    rm <- after bm;
  }
}

public team class T2 {
  protected class R1 playedBy B {
    callin void rm() {...}
    rm <- replace bm;
  }
}

public team class T3 {
  protected class R1 playedBy B {
    void rm() {...}
    rm <- after bm;
  }
}

public team class T4 {
  protected class R1 playedBy B {
    void rm1() {...}
    callin void rm2() {...}
    rm1 <- before bm;
    rm2 <- replace bm;
  }
}

public class Main {
  public static void main(String[] args) {
    T1 t1 = new T1(); T2 t2 = new T2();
    T3 t3 = new T3(); T4 t4 = new T4();
    B b = new B();

    t1.activate();
    t2.activate();
    t3.activate();
    t4.activate();
    b.bm();
  }
}
```

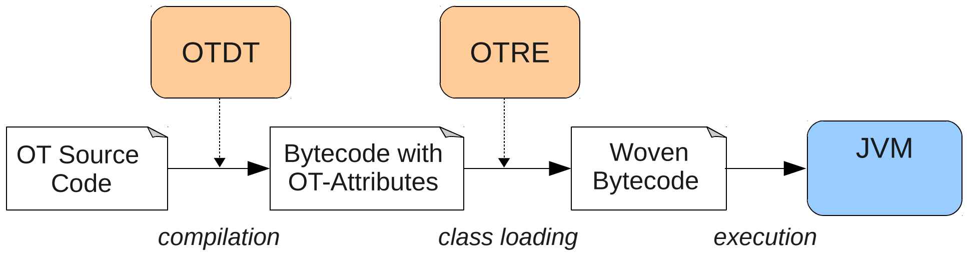


Figure 2.6: ObjectTeams Toolchain

contain all information needed to correctly execute the aspect-specific parts of the OT program.

When the program is started, the bytecode of every loaded class is passed to the *Object Teams Run-time Environment* (OTRE) (cf. [Hun03]) before it is passed to the JVM. The OTRE performs load-time bytecode transformation to weave the aspects into the bytecode of the base classes. The resulting woven bytecode is subsequently executed by a standard JVM. Typically, the Sun (recently Oracle) JVM is used.

In order to weave an aspect into a base method, the OTRE adds a number of fields and methods to a base class. As illustrated in Figure 2.7, every base class gets an `active teams` list that is used for aspect registration and activation. Furthermore, every adapted base method is surrounded by a *wrapper* method (`bm wrapper`). This wrapper is responsible for looking up active teams (1) and for dispatching to the adapting team objects (2). Next, a `callin wrapper` in the team class assigns a role object to the executed base object (3:*lifting*). Then, the role method is called (4). In case of a replace-callin, finally a base-call can be performed (5).

The mechanisms in OT/J that are touched by our optimizations are aspect *activation* and aspect *execution*. To realize *aspect activation*, the team has to be registered with all base classes it contains roles for. The reverse applies for team *deactivation*. During *aspect execution*, for all base method calls the involved aspects have to be looked up, checked for activity, and get executed.

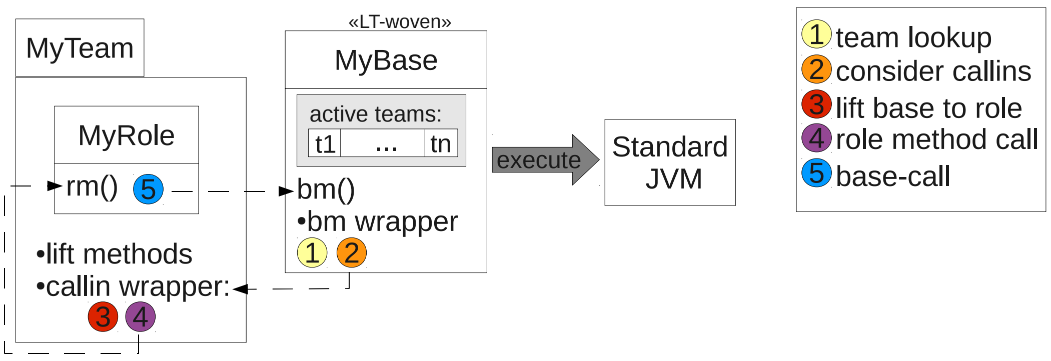


Figure 2.7: Aspect Dispatch in ObjectTeams

The overhead and optimization potential of these AOP mechanisms is analyzed in Section 4.2.

2.1.4 Dynamic AOP

As one goal of our work is the support of *dynamic AOP* for mobile device applications, this section gives an introduction to dynamic AOP. To this end, we motivate the need for dynamic AOP and we examine existing approaches.

Dynamic AOP approaches facilitate aspects to be plugged into a running application. New aspects can be added (and removed) at run-time without restarting the application. In this regard, the degree of dynamism has to be considered, as illustrated in Figure 2.8. In all four cases, an `AspectClass` adapts a `baseMethod` of a `baseClass`. The `aspectMethod` is bound *before* and *after* the `baseMethod`. For simplicity, the example does not include method replacement, as introduced in Section 2.1.1. In static AOP approaches, aspects are hard-wired at pre-run-time (1.). Hence, the aspect calls at the beginning and at the end are fixed parts of the base method. A weak, but still useful dynamic mechanism is the activation and deactivation of aspect during run-time (2.). Again, the aspect calls are statically woven, but they can be turned on and off during run-time. More dynamic are approaches that facilitate the addition of further unanticipated aspects at join points, which are predefined at pre-run-time (3.). In the example, it is not possible to weave an aspect call at the end of the `baseMethod` because no join point has been defined here. The maximum dynamism is accomplished when it is even possible to identify new join points at run-time and attach aspects to them (4.). In the following, we present some scenarios that demand for dynamic AOP.

Motivation for Dynamic AOP

Dynamic AOP can be used to realize dynamic *reconfiguration* of applications [KB08], which is desirable if software systems must adapt at run-time to changing computing needs or environments. This is reasonable if a (long) running application has to be changed, but shutting down and restarting it would be problematic. Dynamic reconfiguration is very useful in several scenarios, for example if software should be updated without restarting the system, for on-demand debugging, and for software that adapts to dynamic contexts.

The three scenarios can be summarized as follows:

Online Updates Some long-running systems like telecommunication systems, scientific or bioinformatics systems, or financial transaction processors have high *availability* requirements. If changes are necessary in such systems, a possible solution is to use redundant hardware and software to ensure availability, while the primary system is updated with new software. This results in significant additional business costs. Another way is to use dynamic AOP

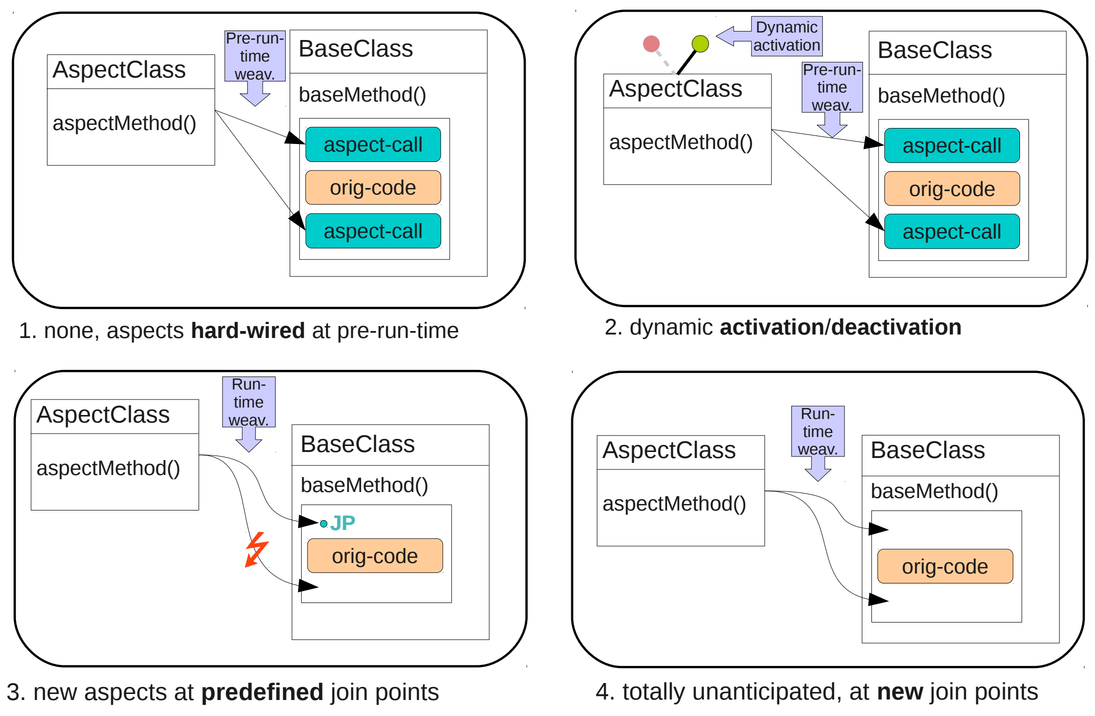


Figure 2.8: Degree of Dynamism

to reconfigure the system without restarting it. Thus, continuous service availability is ensured without increased costs [JTSJ07], [GB04]. A typical example are security-fixes during run-time.

On-demand Debugging Another application for dynamic reconfiguration via AOP is *data preservation* while debugging long running applications with complex internal data structures. Often, a bug only manifests after a longer time period or with a particular data configuration. Shutting down the system and restarting it with debugging support would cause the loss of data and intermediate results. To reconstruct the system configuration that triggered the bug can be difficult and time-consuming. With dynamic AOP, in contrast, we can dynamically add and remove various logging aspects, as soon as the system shows wrong behavior. The dynamically attached aspects can be used to check and manipulate the application state. So, we can dynamically debug the application and examine which fix could solve the problem, without restarting the application many times.

Context-Aware Adaptation For mobile device applications, dynamic AOP can be used to realize on-the-fly adaptation to *changing contexts*. This context can be defined by the *location* of the user as well as by the *subjects* which are next to the user. Also the *daytime* or the presence of other electronic *devices* can necessitate an adaptation of the application. Because of the restricted system resources of mobile devices, it may be preferable to avoid the pre-installation of code for every possible application variant. With dynamic AOP it is further possible to react on changes an application was not designed for.

As we have seen, dynamic AOP has a wide application scope. In Section 3.3, we discuss related work concerning weaving techniques and strategies used by existing dynamic AOP approaches. In this work, we develop a VM-internal dynamic aspect deployment mechanism that is efficient in execution time and space consumption (see Sec. 4.4.2). Furthermore, we address the usability of dynamic AOP for context-sensitive mobile device applications in Section 7.3.

2.2 Java Virtual Machine

Java programs are compiled to Java bytecode which is eventually executed by a *Java Virtual Machine* (JVM). Method bodies are translated to sequences of relatively abstract, machine-independent bytecode instructions. Virtual machines [SN05] are programs designed to execute *architecture-independent* code on concrete platforms. In the process, they translate abstract byte code instructions to platform-specific machine code.

Conventional compiler-based approaches, in contrast, generate architecture-dependent object code, which can only be executed on compatible platforms. Thus, every application has to be recompiled (ported) for every target architecture. With virtual machine approaches, only the JVM program itself has to be ported, but not all the applications running on it. This *portability* is an important advantage since Java programs only have to be compiled once and can then be executed on arbitrary platforms.

Further advances of JVMs are the inherent security model (*sandboxing*), and the automatic memory management via *garbage collection.* Furthermore, with the *dynamic class loading* mechanism of the JVM, classes are *lazily* loaded on demand, which minimizes the amount of consumed memory at runtime (*resource protectiveness*).

As our goal is to optimize aspect execution mechanisms on the level of the JVM, this section introduces the general architecture and functionality of the JVM. Thereby, we address different bytecode execution mechanisms, and finally motivate our selection of a JVM implementation to integrate our optimizations.

2.2.1 The JVM Architecture

Java programs are compiled to Java byte code which is stored in *class files*, in a standard format as defined in [LY99, Chap. 4]. A class file is loaded into the JVM when the corresponding Java class is used and resolved by the running program. The JVM uses various run-time data structures to store the information included in the class file. In addition, a number of data structures

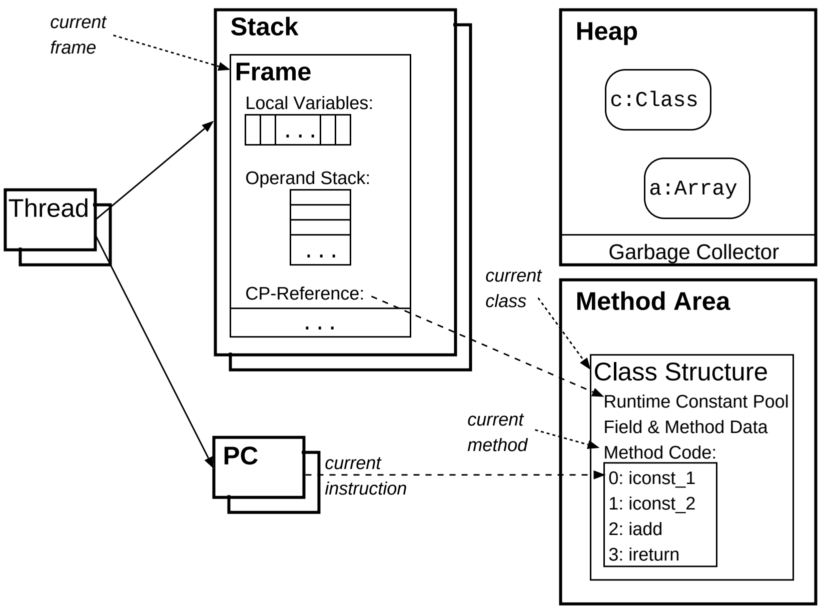


Figure 2.9: Overview of the JVM Rutime Data Areas / Architecture

is maintained at run-time to guarantee a correct execution of the program, e.g., stack frames for method execution. While the structure of the JVM is only abstractly specified by the JVM specification [LY99, Chap. 3], it describes the mainly used VM-internal runtime data structures. This section describes the abstract JVM architecture as far as it is a basis for the subsequent chapters of this work. We discuss the main data structures and the interactions between them. Figure 2.9 gives an overview of the overall architecture of the JVM and illustrates the data structures that are detailed in the following.

Stack The JVM *stack* is a storage area used to store stack *frames*. A new stack frame is allocated and pushed on top of the stack when a method is invoked. If the method returns, the frame is popped again and the control flow returns to the calling method, whose frame is now the topmost.

Heap The *heap* is a global storage area used to allocate and maintain Java objects and arrays, which are accessed via typed references. Storage for new data objects is usually allocated from a pool of free storage cells. A *garbage collector* is used to automatically maintain the life-cycle of objects and to remove them if they are no longer used by the running program. Thereby, the corresponding storage cells are restored to the free-pool. As there is no particular garbage collection technique constituted by the JVM specification [LY99], different strategies can be realized by a JVM implementation. Examples are the *mark-sweep* algorithm and the *copying* algorithm, which both have different assets and drawbacks as extensively discussed by [JL96].

Method Area The *method area* stores the content of the loaded class files in *per-class structures*. For each class the *field* and *method data* is stored as well as the *method code* represented as sequence of byte code instructions. Furthermore, the *runtime constant pool* of the class is stored. It contains constant values and symbolic references which are *dynamically linked* (cf. [LY99, Chap. 5]) when the corresponding data is used and resolved for the first time (*late binding*).

PC The *pc register* points to the currently executed bytecode instruction. Only if the currently executed method is *native*, the value of the pc register is undefined.

Frames A new *stack frame* is created and pushed onto the stack when a method is invoked. Frames contain an array of *local variables*, an *operand stack*, and a *reference* to the runtime constant pool of the corresponding method area. As indicated by its name, the *local variables array* contains the local variables of the method. In addition, also the method parameters passed by the calling method are located here, at the beginning of the local variables structure. For instance methods, the first entry is the `this` reference, pointing to the object which is the target of the method invocation. The *operand stack* is used by the bytecode instructions, while executing the bytecode instructions corresponding to the methods body. They can *load values* on it, *take operands* from it and *put results* on it. The operand stack is also used to pass parameters to other methods and receive their results. A very simple example is the code of a method returning the sum of two constant integer values, illustrated in the method of Figure 2.9. The first two instructions (`iconst_1` and `iconst_2`) are used to push two constants (`1` and `2`) onto the operand stack. Next, the `iadd` instruction takes these two values as operands and puts the result (the sum: 3) onto the stack. Finally, the `ireturn` instruction returns the result to the calling method, by putting it on the operand stack of the preceding stack frame. The *constant pool reference* is used to access constant values and perform dynamic linking if symbolic references are not yet resolved, e.g., by previous method executions.

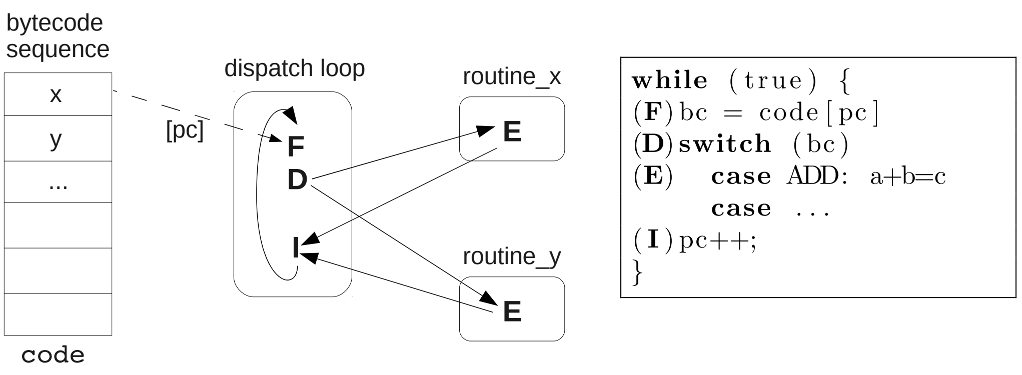
Threads A Java program can use different *threads*. While the heap and the method area are shared among all threads, each thread uses its own stack and PC register. The current point of execution is thus exactly defined for every thread. The currently executed method corresponds to the active (*current*) *frame* of the thread's stack and the currently executed bytecode instruction is stored by the thread's pc register. In Figure 2.9 this thread-locality is illustrated by "stacked" boxes.

2.2.2 Bytecode Interpretation

When the Java compiler translates Java programs to Java bytecode, method bodies are translated to sequences of machine-independent bytecode instructions. A Java bytecode instruction “consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation” ([LY99, 3.11]). While executing a program, these instructions, in some way, have to be “translated” to platform-specific native code. A typical approach is the *interpretation* of Java bytecode. This section introduces the classical interpretation approach. Subsequently, established optimizations to enhance the performance of bytecode execution are discussed in Section 2.2.3. Finally, Section 2.2.4 describes how Java can be used in the context of embedded systems.

Basic interpreters, as illustrated by Figure 2.10, use a central dispatch loop to process the instructions of the bytecode sequence one by one. First, the current instruction is *fetched* (F), then it is translated or *decoded* (D) into corresponding native code instructions and finally, it is *executed* (E). Afterwards, the bytecode pointer is *incremented* (I) to reference the next bytecode instruction and the loop is restarted to process the next instruction. As shown in the code excerpt on the right hand side of Figure 2.10, basic interpreters are implemented as large switch statements traversing the sequence of bytecodes, and thus called *switched interpreters*. For every bytecode instruction a case label exists in the interpreter switch. Thus, the opcodes are looked up to identify the corresponding native code instructions. If an operation uses one or more operands, these also have to be fetched (from the operand stack) and passed to the native routine. For simplicity, this has been abstracted away in the algorithm descriptions.

Basic interpretation is highly portable and has a fast startup time. Furthermore, its memory consumption is low. Unfortunately, the execution performance of basic interpretation is rather low.



```
while (true) {
(F)bc = code[pc]
(D)switch (bc)
(E)   case ADD: a+b=c
      case ...
(I)pc++;
}
```

Figure 2.10: Basic Interpretation

2.2.3 Optimized Bytecode Execution

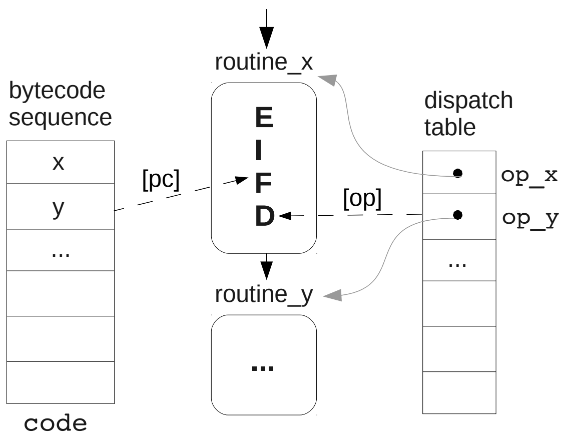
The basic interpretation algorithm presented in Section 2.2.2 is concise, but not very efficient. Early versions of Java have thus been called "slow". By now, many optimization strategies have been developed, to optimize the execution of bytecode (e.g., [DW03], [AFG$^+$05]). Typically, they target the reduction of dispatch overhead, or a lower frequency of branches or stack and instruction pointer updates. Often, there is a tradeoff between efficiency and portability of the interpreter code. This section gives an overview of the common approaches in this regard.

Threaded Interpretation

Threaded interpretation, as introduced by [Bel73] and described by [SN05, Chap. 2], is a technique used to improve the performance of basic interpretation, by reducing the amount of branches. To this, instead of being located in the main interpreter loop (see 2.2.2), the code for dispatching the next instruction is appended to the end of each native routine. Thus, the code pieces for executing the individual instructions are "threaded" one after another. According to the strategy used for locating the next instruction, a distinction between *indirect* and *direct* threading is made.

Indirect Threading *Indirect threading* uses a *dispatch table* to lookup the actual address of the routine implementing the next bytecode instruction, as visualized in Figure 2.11. Each routine first executes (E) the functionality of its bytecode instruction, then increments (I) the code pointer and fetches (F) the next bytecode instruction. Afterwards, the corresponding opcode is used to lookup the address of the next routine and jump to it (D).

The advantage of the "indirect" jumps using the dispatch table is that "interpreter routines can be modified and relocated independently" ([SN05, p. 34]). This simplifies the portability of the interpreter code.



```
routine_x {
(E)<do_x>
(I)pc++;
(F)bc = code[pc]
(D)next_routine =
     dispatch_table[bc]
   goto *next_routine;
}
```

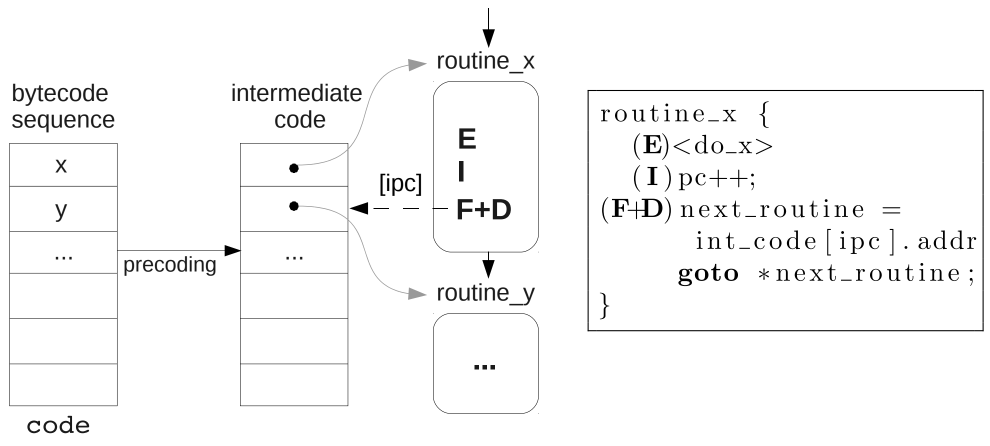
Figure 2.11: Indirect Threaded Interpretation

Direct Threading *Direct threading* further reduces the overhead of interpretation by eliminating the access to an extra dispatch table. As the address of the next interpreter routine needs to be determined nevertheless, a technique called precoding is applied. Precoding transforms the bytecode instructions into a VM-internal *intermediate representation*, which is optimized for being processed by the interpreter. Tasks, which would otherwise be repeated each time the code is executed, are optimized this way. To support direct threading, the actual addresses of the interpreter routines are stored as part of the intermediate representation, as illustrated in Figure 2.12. In contrast to indirect threading, with direct threading the fetch and decode phases (F+D) are merged and the jump target for the next instruction can be directly accessed from the intermediate code of the current instruction. To navigate through the intermediate code, an extra instruction pointer (ipc) is used.

While direct threading is fast, it limits the portability of the interpreter code.

Superinstructions

Superinstructions are used to reduce the number of instruction dispatches by combining sequences of common bytecode instructions. This concept was initially introduced as *super operators* by [Pro95]. In addition to reducing the number of instruction dispatches, superinstructions enable further optimizations of the interpreter code. Thus, unnecessary *load* and *store* operations can be saved and also the number of stack and instruction pointer updates can be reduced. An example, given by [CGEN03], is the combination of an `ILOAD` and an `IADD` instruction. Table 2.2 summarizes the benefit of the superinstruction `ILOAD-IADD` compared to the single instructions. The amount of stack access has been halved, the stack pointer does not need to be updated at all, and the instruction pointer only needs to be updated once, at the end of the routine implementing the superinstruction.



```
routine_x {
   (E)<do_x>
   (I)pc++;
(F+D)next_routine =
         int_code[ipc].addr
       goto *next_routine;
}
```

Figure 2.12: Direct Threaded Interpretation

Table 2.2: Benefit of Superinstructions

	ILOAD	**IADD**	**ILOAD-IADD**
stack access	store	load, load, store	load, store
		sum: **4**	**2**
stack pointer updates	sp+=1	sp-=1	–
		sum: **2**	**0**
instruction pointer updates	ip+=1	ip+=1	ip+=2
		sum: **2**	**1**

Usually, the sequences that can be properly combined to a superinstruction are chosen via statistically counting their frequency. This can be done via *static* or *dynamic* program analysis and also includes the evaluation of standard benchmarks. The substitution of selected instructions with superinstructions is done at runtime when a method is executed the first time.

Stack Caching

As introduced in Section 2.2.1, the operand stack is used by the bytecode instructions to access arguments (or operands) and to pass their results. *Stack caching* ([Ert95, EG04]) reduces the stack access overhead by *caching* the topmost stack items in registers. The number of items located in registers corresponds the size of the stack cache. The other items remain on the stack.

The highest reduction of native instructions is achieved if the number of items kept in registers *dynamically* depends on the behavior of the preceding instructions. Using a *fixed* cache size would require the update of cache registers with stack items (and a stack pointer update), whenever the number of items consumed by an instruction differs from the number it produces. This can be avoided by dynamically switching between states with different cache sizes, depending on the stack effect of an instruction.

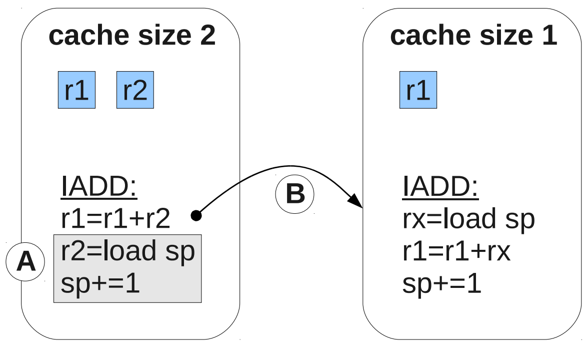


Figure 2.13: Stack Caching: Fixed (A) vs. dynamic (B) Cache Size

Figure 2.13 illustrates the difference, using an example from [EG04]. The IADD instruction *consumes two* items and only *produces one* item. If the interpreter is in a state with cache size *two* while IADD is called, the values of the two registers r1 and r2 can simply be added and stored to r1. With fixed cache size (A), the second register r2 has to be updated with the topmost stack item and the stack pointer has to be incremented, afterwards (see gray area). Using a dynamic cache size (B), the interpreter simply changes to a state with cache size *one* (see arrow), causing the following interpretation to expect only one cached item[1].

The routines implementing the functionality of the bytecode instructions need to be different, depending on the current stack cache state. The consequential increase of the interpreter code size is the trade-off for enhancing the interpretation speed.

Quick Instructions

Quick instructions [LY96, Chap. 9][ETK06] (also called *quickening*) are used to speed up the execution of instructions, which have to do some complicated initializations the first time they are executed. After the initialization, they are replaced by their quick counterparts. Such instructions typically reference the constant pool. The initializations include checking whether a referenced class is already loaded and initialized, as well as resolving information needed for the execution of an instruction. E.g., for a getfield instruction the offset of the accessed field has to be resolved. While the original instruction receives the constant pool reference to the field as operand, the operand of the quickened instruction (getfield_quick) is the offset of this field.

Prefetching

Prefetching, also called interpreter pipelining [HA00], [WNGG08], is a technique to improve the performance of interpreters running on processors, which allow to execute multiple instructions in parallel (super-scalar or very long instruction word (VLIW) processors). While interpreting a bytecode instruction, the *execute* and the *increment-fetch-decode (IFD) part* (see 2.2.2) are usually independent. The latency for many simple bytecodes is determined by the IFD part. With prefetching, the jump to the successor can be started in parallel to the actual bytecode execution.

Dynamic / Just-in-time Compilation

An alternative to the optimizations of bytecode interpretation presented so far, is to let a *compiler* translate the bytecode of methods into native code. To preserve Javas *portability* and *security* properties, this compilation has to be

[1]Note: After executing an instruction like *ILOAD*, producing one item, the interpreter returns to the *cache size two* state.

performed *dynamically*, at runtime. Thus, the compilation overhead impacts the runtime performance. To minimize this drawback, each method is compiled not until it is executed for the first time. Hence, this kind of compilation is called *dynamic* or *just-in-time* (JIT) *compilation* [CFM+97, Ayc03].

Compilers typically perform various optimizations before emitting the compiled code. For dynamic compilation, this is a trade-off between the speed of the optimization algorithms and the performance gain for the application code. Thus, expensive conventional compiler optimizations cannot be applied because of their overhead at runtime. Nevertheless, [CFM+97] describe, how bytecode can be transformed into efficient native code. So, the compiler can optimize the order of evaluation, as long as the new order also conforms to the JVM specification. Also, redundant computations in basic blocks can be avoided, like multiple loading of values or repeated array bounds checking. Adaptive optimizations try to assure that compilation costs pay off, by only compiling frequently executed methods.

JIT compilation results in high performance gains, in particular for programs which are largely concerned with the execution of bytecode instructions. In contrast, other parts of the JVM, like exception handling, synchronization, and memory management are not optimized by JIT compilation.

JIT Compilers vs. Interpreters JVMs realize the execution of bytecode instructions by *interpretation* or *just-in-time compilation.* This section discusses the pros and cons of the two approaches. The comparison criteria are *execution performance*, *memory efficiency*, *simplicity* and *portability.*

As pointed out by [Ayc03], there is a *time-space* tradeoff between JIT compilation and interpretation. While the *execution performance* is generally faster using JIT-compilation, interpreters are more *memory efficient.* In addition to the extra memory needed for the JIT compiler code, the compiled native application code uses significantly more memory than the corresponding bytecode. According to [CFM+97], one byte of bytecode on average corresponds to four bytes of Intel machine code instructions. Also [CGEN03] discusses a number of advantages of interpreters, which especially apply to small platforms like embedded systems. Beside the memory efficiency, interpreters are also more *portable* to different architectures and significantly smaller and *simpler* to maintain than JIT-compilers. Table 2.3 summarizes the results of comparing JIT compilers and interpreters.

The higher memory consumption of JIT compilers is problematical for platforms with small memory, as embedded systems. Also simplicity and portability are important for such systems, as there is a large variability of hardware device targets. Hence, using an interpretation approach incorporating optimization techniques as presented above, is a good choice for small platforms.

Table 2.3: JIT Compiler vs. Interpreter

	JIT ompiler	**Interpreter**
execution performance	⊕	⊖
memory efficiency	⊖	⊕
simplicity	⊖	⊕
portability	⊖	⊕

Optimized Method Invocation

The invocation of a method involves passing of parameters, managing of an additional stack frame, and transferring the control before the actual method code can be executed. Method inlining reduces this overhead by replacing method calls with the code of the corresponding methods, thus inlining the code into the calling method. This optimization is especially profitable for small methods because here the dispatch overhead is high compared to the execution of the actual method code. An additional benefit of inlining is that it increases the scope for later optimizations. However, a drawback is a larger binary program due to the code repetition.

In most object-oriented languages, the target of a virtual method call via `invokevirtual` is dynamically chosen by dynamic method table lookup. This dynamic method dispatch constitutes a major part of the overhead of object-oriented program execution. Furthermore, it complicates the use of inlining because it cannot be exactly determined which method code has to be inlined. [SN05, 6.6.2] illustrates strategies to face this problem. In some cases, it can be determined that a method cannot be overridden because it is `final` or no subclasses have been loaded (so far). If subclasses are loaded later, the inlining has to be undone. Another possibility is *speculative method inlining.* The most likely called method can be determined by profiling. Its code can be inlined together with a *guard instructions*, checking if the actual target object type corresponds to the inlined code. If the guard fails, the correct method is called via normal `invokevirtual`, instead.

The dynamic method dispatch has been optimized by various other approaches. For example, the method table lookup is optimized by compressing the lookup tables ([VH96]) or by introducing a cache to store the target of a method invocation, as e.g., *polymorphic inline caching* [HCU91].

As presented in this section, a wide range of optimizations have been developed in order to improve the performance of bytecode interpretation and (dynamic) method invocation. When designing optimizations for aspect-oriented execution mechanisms, we have to take into account these existing optimizations. On the one hand, we have to ensure that our optimizations do not interfere with them. On the other hand, we investigate how existing optimizations can be extended to become applicable to AOP mechanisms (see Sec. 5.4).

2.2.4 Java for Embedded Systems / Small Platforms

Modern *desktop computers* are used for a wide range of purposes and come with extensive resources, like CPU power and memory capacity. The user can install various operating systems and, on-top of them, arbitrary application programs. However, the situation is totally different for *embedded systems*, like digital cameras, cars, and mobile devices like phones or hand-held computers. They have to be small, lightweight and inexpensive. Often, they need to be portable and have no permanent access to power supply. For these reasons, embedded systems have *restricted resources*, like limited CPU power, less memory and low power capacity. Embedded systems are special purpose computers, often providing a restricted user interface. Furthermore, the user often has no or limited access to the software installed on the device. In addition to resource limitations, [LR05, page 25 ff.] identifies further challenges for the context of embedded systems. These can imply additional non-functional quality criteria like *safety* and *fault-tolerance*, as well as *real-time* requirements, or more *reliability*.

Modern mobile devices provide the subsequent installation of various applications from different domains. Typical examples are mobile games, travel guides and maps, or utilities that provide the user with additional information, like the weather forecast. Today, in the mobile market the variety of hardware platforms is still high. To be competitive, application providers intend to support as many as possible of them.

A widely-used software platform (and operating system) for mobile applications is *Symbian* [Nokb]. Here, the applications are typically developed with standard C++ using Qt [Noka] as SDK. When the user installs a new application, the hardware specification is automatically submitted to the application server. This guarantees that the compatible binary is downloaded. Thus, the application server must maintain all the different binaries for every supported platform.

The *Android* platform [Gooa] for mobile devices also features the development of applications written in Java. However, instead of using a standard JVM, the compiled Java class files are converted into a proprietary class format and executed by Android's *Dalvik VM* [Nic09, Goob].

Because of the high variability of devices, developing mobile applications *platform-independently* is extremely relevant. As *portability* is one key feature of Java as a programming language, this is an appropriate alternative for developing mobile applications. The *Java Micro Edition* (Java ME) [Sch04] is a Java platform specialized for limited resources of embedded systems. It uses *configurations* and *profiles* to support the features of different target devices. A configuration summarizes devices with similar characteristics and defines minimal requirements and available Java features. It contains specialized libraries and a virtual machine. Two different configurations have been developed: The *Connected Device Configuration* (CDC) and the further restricted *Connected Limited Device Configuration* (CLDC) for mobile devices. *Profiles* more specifically define Java APIs (application programming interfaces) supported by the designated devices. Most known is the *Mobile Information Device Profile* (MIDP), which includes functionality for developing *graphical user interfaces* (GUIs) and the *persistent storage* of data. Furthermore, additional optional packages can be supported by individual devices. Current mobile phones typically support the CLDC configuration together with MIDP.

Embedded JVMs

Different JVMs are used in embedded systems. In this section, we introduce some of them and finally, choose one for implementing our optimizations for the execution of aspects.

The *phoneME VM* [pho] is an open-source reference implementation for Java ME. It is a variant of the Java HotSpot virtual machine, which is highly optimized featuring adaptive JIT-compilation.

The *Dalivk VM* (DVM) [Nic09, Goob] is part of the Android platform. It does not execute Java class files, but requires transforming them into the *Dalvik executable* (dex) format. According to this, the DVM is not a JVM in the classical meaning, however due to its hight popularity it demands to be mentioned, here. In contrast to the JVM which is a stack-based machine, the DVM is *register-based.* Thus, bytecodes and operands are read from virtual registers instead of from the stack. This is especially beneficial if the virtual registers can be mapped to real *processor registers.* Unfortunately, this hardware-specific realization *degrades* the *portability* of the VM implementation. The most relevant differences between the dex-format and the Java class file format [LY99, Chap. 4] concerns the design of the *constant pools* and *granularity* of the files. The dex-file format is optimized for *memory usage.* In contrast to a class file, a dex-file may contain *multiple* classes. While the Java constant pool is heterogeneous, the dex-file contains *type-specific* areas, e.g., for types, signatures and method identifiers. Due to the combination of multiple classes, no repetition of strings is necessary for all classes contained in one dex-file, reducing the average file size by 35% percent [Nic09, page 9]. The DVM is written in C. Recent versions also include a JIT-compiler.

The JamVM [Lou] supports the full JVM specification version 2 [LY99], although it is extremely small and applicable for embedded devices. It has been ported to different architectures, like ARM and MIPS. The JamVM is implemented in C, with a small amount of platform-dependent assembler code. It does not contain a JIT, but its interpreter is highly optimized, implementing many state-of-the-art techniques such as *direct-threading*, *quick instructions*, *stack caching* and *superinstructions*, as introduced in Section 2.2.3. A comparison of Java VMs on ARM based systems [bug08] shows that the JamVM is a good choice for embedded Java development.

The VM implementation we select as basis for implementing our optimizations has to fulfill a set of *criteria.* The most important ones are *availability* (*open-source*), good *extensibility*, sufficient *performance*, applicability to *em-*

bedded architectures and *compatibility* to the existing OT/J. We have selected the JamVM [Lou] for implementing our optimizations.

Other candidates we considered were the KVM [KVM00] and the phoneME VM. In contrast to the JamVM, both only support Java ME. The KVM does not seem to be maintained during the last time. phoneME features a JIT compiler, but this makes its implementation much more complex than the JamVM and not suitable for prototypically evaluating our optimizations. As the Dalvik VM is not a classical JVM, it cannot be directly used for executing ObjectTeams programs. Nevertheless, in Section 7.4.1, we investigate the applicability of our optimizations to this virtual machine.

3 Related Work

In this chapter, we describe the work related to our approach. We start by examining existing approaches that target the optimization of aspect-oriented programming in Section 3.1. After that, in Section 3.2, we present approaches that utilize high-level semantic information for optimizations. As we aim at supporting dynamic aspect deployment, we further investigate existing dynamic AOP approaches in Section 3.3. In Section 3.4, we discuss a machine model for AOP that influenced our approach, and finally, we conclude with a summary in Section 3.5.

3.1 Optimization for Aspect-oriented Programming

Optimizations for the execution of aspect-oriented programming languages have been proposed in various approaches.

Some of them improve the run-time performance by restraining the dynamic capabilities of a language. For example, the aspect dispatch can be optimized by replacing dynamic aspect method lookup by static method calls [GK07]. This is possible because in AspectJ aspect methods (advice) cannot be polymorphically redefined. Such optimizations are not applicable to our problem because (as [EL03]) we appraise the dynamic method lookup of aspect methods important for an adequate modularization of software. Like our approach, Aspect C++ [SLU05] is also intended for the generation of small and efficient code for constrained environments. However, it does not focus on reusability of the aspects and uses a purely static weaving mechanism only.

Other approaches integrate optimizations into the execution environment of aspect-oriented languages. For example, [Hau06], [HM05] extend the Jikes Research Virtual Machine to directly support general AOP language mechanisms. The introduced *advice instance tables* have inspired our team activation infrastructure. In this approach, the dynamic weaving of aspects is realized by an extension of the adaptive optimization system (AOS) of the Jikes VM. The AOS comprises a runtime profiling component and a recompilation component that makes use of a JIT compiler. Due to its dependence on the rather complex AOS, this work is not directly transferable to embedded device applications.

In [BADM06], well-known virtual machine optimization techniques are adapted to improve aspect-oriented run-time performance. These optimizations are based on a JIT compiler and primarily aim at eliminating overhead caused by previous optimizations of the compile-time. As motivated in 2.2.3, we chose a virtual machine without a JIT compiler, by contrast. Furthermore, compilation-time reduction, although desirable, is not our focus because usually applications for embedded systems are compiled on more powerful host systems.

3.2 Using High-level Semantic Information for Optimization

It is a widely accepted fact that the use of high-level abstraction mechanisms induces an overhead compared to semantically equivalent lower-level realizations. At the same time, these abstractions enable the development of programs with higher qualities like maintainability and reusability. To resolve this conflict, a number of approaches aim at utilizing additional semantic information to enable so called *abstraction-aware* optimizations. These approaches focus on object-oriented language abstractions, like polymorphism and dynamic method dispatch and propose a set of annotations for describing special properties of abstractions introduced by the programmer.

[TM99] proposes an *annotation language* used to make the intentions of the developer explicit. In addition, a *transformation language* is used to describe how parts of a program with annotated intentions can be optimized. The compiler performs source-to-source transformations and uses the additional information to apply more sophisticated optimizations.

[QSYS04] and [QSVY06] extend the applicability of predefined compiler optimizations by taking into account additional semantic information gained by developer annotations. They differentiate between function annotations, data annotations, and object-oriented annotations. Function annotations are used to express the intended semantics of functions, e.g., by specifying which variables/parameters of a function are modified or aliased. Furthermore, they can restrict the possible values of variables and describe relations between variables. Data annotations are used to specify the semantics of types. For example, types can be specified to conform to a general array concept together with a mapping of common operations, like the determination of the array's length. Object-oriented annotations are used to express to what extent the properties expressed by annotations are inherited to subtypes. The high-level semantic information about abstractions expressed by these annotations serves as additional input for the optimizing component of the compiler.

In this work, we also aim at reducing the overhead caused by high-level abstraction mechanisms. However, our approach focuses on the optimization of aspect-specific abstractions. Furthermore, we do not use annotations but infer the information from the syntactical structures of the aspect-oriented program.

Complex annotations may enable more specialized optimizations, but require disciplined usage by the programmer. To guarantee a correct compiler output it would be necessary to verify the user defined annotations against the actual properties of the program. Our contribution to utilize aspect-specific semantic information for the optimization of AOP mechanisms is presented in Chapter 5.

3.3 Dynamic AOP Approaches

Known (pre-run-time) AOP weaving approaches are typically realized in some form of transforming existing programs. More precisely, this requires techniques like adding methods/fields to classes as well as the possibility of changing the implementation of existing methods (see Sec. 2.1.2). Dynamic AOP systems will technically have a similar weaving realization. Adapting an already running Java application means accessing classes that are already loaded into the executing JVM. From this follows the need to intervene the normal operation of the JVM. For this purpose an appropriate interface with the JVM is needed.

3.3.1 Dynamic Weaving Techniques

Run-time weaving, as introduced in Section 2.1.2, is a prerequisite for dynamic AOP. Existing dynamic AOP approaches use different mechanisms to interact with the JVM, in order to adapt the code of a running application.

Early approaches like PROSE1 [PGA02], Axon [AH03], and Wool [SCT03] in its initial phase, utilize the debugger interface to realize dynamic AOP (see Figure 3.1). Join points are mapped to break points in these approaches. If a break point is reached, the running program is intercepted and the control is redirected to a dynamic AOP component. This component is responsible for calling the aspect functionality defined for the current join point. Afterwards, the control is returned to the application code. This implies that only purely additive adaptations (after/before) are possible. The advantage of this debugger-based approaches is a separation of the aspect binding declaration and the aspect functionality. Moreover, the existing Java Platform Debugger Architecture (JPDA) [JPD] can be reused. This is a disadvantage at the same time: applications have to run in the debug mode in these approaches, causing significant performance overhead [Meh03]. Another source of overhead is the need for context switches between the application process and the process running the dynamic AOP component.

Other approaches like RtJAC [Esp03] and Wool [SCT03] in its later phase use run-time code replacement to subsequently weave aspectual code into classes which are already loaded into the JVM (see Figure 3.2). These approaches use the HotSwap mechanism [Dmi02] of the JPDA or the more recent Java Programming Language Instrumentation Service (JPLIS) API [JPL] to update the bytecode of classes inside the JVM. The performance is better

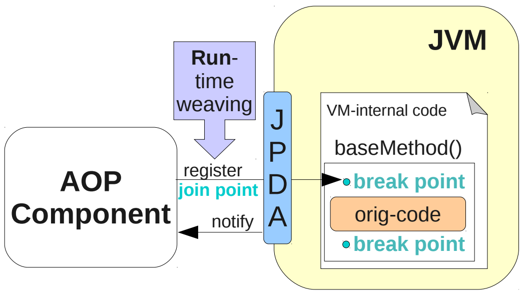


Figure 3.1: Debugger-based Dynamic AOP

than for the previously described debugger approaches. Yet, a disadvantage is the restriction of purely schema conserving modifications, forbidding the addition of methods or fields. This reduces the applicable weaving approach to pure in-method weaving, complicating the adoption of existing approaches. Furthermore, this code replacement is very coarse-grained, as it only allows to redefine complete classes and not, for example, single methods.

Finally, some approaches directly incorporate the dynamic AOP support into the JVM (see Figure 3.3). They either extend the JIT (just-in-time) compiler to insert the aspect code directly into the generated native code [PAG03] or extend the execution model of the JVM to achieve native support of AOP mechanisms [BHMO04, Hau06]. A disadvantage of this approaches is that they demand a specialized JVM. But in return, they allow efficient realizations of dynamic AOP and a continuous presence of aspects even at run-time.

3.3.2 Dynamic Weaving Strategies

Different strategies exist to enable the dynamic addition of aspects at run-time. Approaches with the highest degree of dynamism facilitate the unanticipated addition of aspects at arbitrary join points. Many approaches use some kind of *hook weaving* to prepare the application code for the possible addition of aspects. *Total* hook weaving prepares every possible join point, enabling an arbitrary adaptation, later. However, this approach produces a general

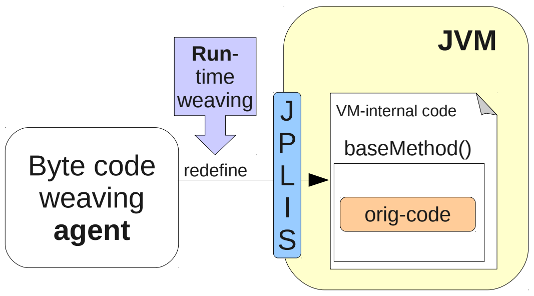


Figure 3.2: Dynamic AOP with Code Replacement

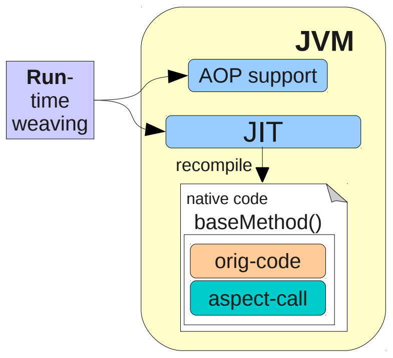


Figure 3.3: Dynamic AOP Integrated in the JVM

overhead, independent of the actual adaptation with aspects. A totally unanticipated weaving, on the other hand, requires a complete redefinition of the affected classes, or rather, a complex run-time evolution of VM-internal data structures like method blocks or stack frames. Therefore, some approaches like JBossAOP [JBo] require the predefinition of possible join points at program start to restrict the amount of preparation costs.

As our work targets mobile devices with limited resources, we need a solution which is efficient in execution time and space consumption. Thus, we cannot reuse existing approaches that use a debugger or which are based on JIT compilers. Furthermore, we do not want the set of join points to be restricted, and thus avoid a static predefinition. In this work, we develop a VM-internal mechanism to meet these requirements (see Sec. 4.4.2).

3.4 Machine Model for AOP

Delegation-based AOP [HS07] proposes a machine model for aspect-oriented languages that are based on object-oriented ones. The authors aim at closing the semantic gap between an AOP-language expression and its realization, as AOP-specific dispatch code should not be realized as part oft the application code but be machine model inherent. In this approach, every object is indirectly referenced via a proxy object that delegates messages to the object. This introduction of additional indirection facilitates the (dynamic) deployment of aspect objects. The activation of an aspect results in the insertion of an aspect proxy object between the initial proxy and the actual object. In this delegation chain, the aspect proxy is able to intercept all the messages for which it defines aspectual functionality. The remaining messages are just delegated to the original object. The existence of the initial proxy makes the deployment of aspects transparent, in that no objects have to be updated in order to point to a subsequently introduced aspect proxy. In [SHH09], the authors present an implementation kernel for the more general *delegation-based multi-dimensional separation of concerns* (MDSOC), which also comprises *context-oriented pro-*

gramming [HCN08]. However, they point out that this implementation is a proof-of-concept that is not designed for efficient execution.

The delegation-based AOP approach has influenced our delegation proxy approach (see 4.4). Yet, we decided to introduce proxies at the level of methods instead of at the object level, in order to better meet the ObjectTeams-specific requirements. Here, aspects are represented by a combination of team objects, which can be dynamically activated and deactivated, and of role objects, to which base objects have to be lifted.

3.5 Summary

In this chapter, we presented related work of our approach. We have seen that existing approaches for optimizing AOP are either too restrictive regarding dynamic language capabilities or too complex for being applied to embedded mobile devices. Previous work on abstraction-aware optimizations only addresses object-oriented language abstractions and depends on programmer annotations. In our approach, we develop optimizations for AOP mechanisms that are compatible with the restrictions of embedded mobile devices and that incorporate AOP-specific semantic information. Furthermore, we realize dynamic aspect deployment that neither requires static predefinitions nor depends on expensive techniques like a debugger or a JIT compiler but uses dynamic method proxies that are related to the approach presented in Section 3.4.

4 Optimized Aspect Execution Mechanisms inside the JVM

Most of the existing AOP languages are extensions to existing programming languages like Java. The introduced aspect-oriented mechanisms are typically mapped to constructs of the underlying language, e.g., the Java bytecode. For example, for aspect activation, registration data structures are usually introduced on the level of the base class bytecode. Aspect execution is typically realized by additional wrapper methods that initiate and coordinate the aspect dispatch and execution as well as the execution of the original method. Realizing common AOP language mechanisms on the level of the application code causes a lot of overhead that cannot be sufficiently reduced at this level. Analogously to the realization of other execution mechanisms like dynamic method dispatch, also AOP mechanisms should be natively supported by the execution environment. For languages based on Java, this is the JVM. At this level, the optimization potential is much higher than at the bytecode level. Therefore, in this work we optimize common AOP mechanisms on the level of the JVM. Among other things, we realize a native aspect activation mechanism and develop bytecode instructions for efficient aspect dispatch and execution.

In this chapter, we start by motivating our decision to optimize the execution mechanisms of AOP at VM-level. In Section 4.2, we then investigate the sources of overhead typically generated by the realization of AOP mechanisms. After that, we present our optimizations for the different kinds of overhead in Section 4.3. Furthermore, in Section 4.4, we describe our approach for dynamic aspect deployment. Finally, we conclude this chapter with a summary in Section 4.5.

4.1 Optimizing on Virtual Machine Level

In this section, we motivate our decision to focus on the run-time environment when designing our optimizations for the overheads that degrade the efficiency of aspect-oriented program execution. As we target AOP languages which extend the Java programming language, like ObjectTeams, the run-time envi-

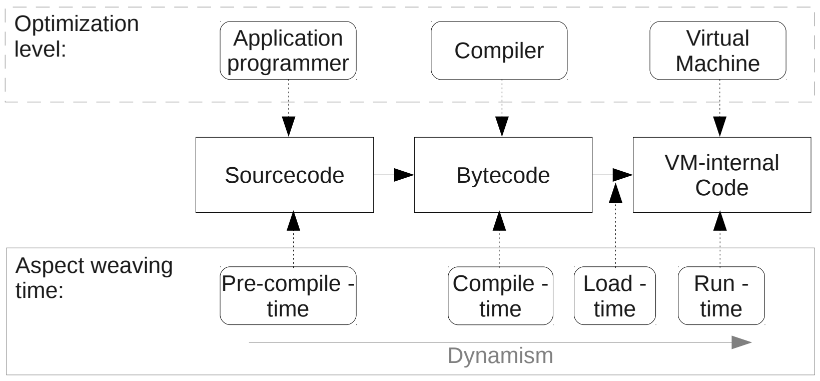


Figure 4.1: Optimizations and Aspect-weaving at different Levels

ronment we focus on is the JVM. We discuss the advantages of optimizing at the JVM level as well as the influence on aspect weaving.

Eventually, optimizations of the execution time of programs are applied to code. When we talk about optimizations for Java, we can consider code representations at different levels. Figure 4.1 identifies three different code representations as target for optimizations: the source code, the bytecode, and the code executed by the virtual machine. At the source code level, the application programmer can optimize an application by avoiding cost-intensive Java constructs and data structures. Of course, AOP programs can profit from these optimizations in the same way that regular Java programs do, but in this work we are not interested in such optimizations. We consider them as part of the design decision, which we do not question.

The compiler translates the source code to bytecode. In the process, basic optimizations such as *constant folding* and *dead code elimination* are applied. In case of an AOP language, the compiler also generates aspect-specific bytecode for the AOP parts of the program. Although this is not our main focus, we analyze this generated code and propose alternatives if we identify the use of costly constructs. In some cases, we also have to adapt the generated code in order to enable optimizations which are implemented at the virtual machine level.

The bytecode of programming languages like Java, which is executed by a virtual machine (VM), is machine-independent and rather abstract. Therefore, a lot of optimizations (cf. 2.2.3) can only be applied at the VM-level. In addition, many optimizations are only reasonable if they are applied to parts of the code that are frequently executed (*hot spots*).

In Section 2.1.2, the process of aspect weaving and the different phases at which weaving can occur are discussed. As stated there, one goal of our approach is to support dynamic weaving of aspects at runtime. To realize this, an interface to the run-time environment is necessary that allows for some kind of class redefinition. In this case, the aspect-specific code can only be optimized on the level of the virtual machine because it is not yet available in earlier phases. Figure 4.1 summarizes the different levels at which optimizations and aspect weaving can be applied. It becomes apparent that the degree of dynamism increases towards the right hand side of the diagram.

In this work, we are interested in the optimization of aspect-oriented parts of applications. Furthermore, we intend to increase the dynamic potential of the AOP language. We expect that the overhead of AOP mechanisms can be reduced much better at the level of the JVM than at the level of Java bytecode. Inside the JVM, we have direct access to the mechanisms and data structures necessary to support method execution, as well as to the thread-handling mechanism. Moreover, we can adapt the execution (interpretation) of bytecode instructions to natively incorporate AOP mechanisms. In addition to this, implementing functionality at the VM-level rather than at the bytecode level has the general advantage that the VM code will be natively executed instead of being interpreted. Beside the increased optimization potential, considering the JVM level is a precondition for our intent to support the dynamical weaving of aspects at run-time.

According to this, we propose to optimize the execution of AOP mechanisms on the JVM-level. In doing so, we successively shift the *weaving mechanism* to the JVM-level. We achieve this by adapting the JVM code, making it aware of AOP-mechanisms in order to execute them more efficiently. We augment the VM-internal application code by storing additional information e.g., at data structures representing classes, objects, or methods. This information is used to optimize the execution of aspectual mechanisms like aspect registration and activation, or role lifting. Furthermore, we introduce additional bytecode instructions that make use of the newly introduced data structures and which are significantly faster than the corresponding Java bytecode.

4.2 Overhead of AOP

Purely additive realizations of aspect-oriented programming languages per se produce a certain amount of overhead compared to the programming language they extend. This overhead can be relevant with respect to space or time. Both kinds of overhead are critical especially for embedded devices with limited memory and computation power and have to be addressed by our effort to optimize aspect-oriented program execution. Hence, we aim at reducing the size of additional code as well as at optimizing its execution. In this section, we analyze the overhead caused by the different tasks performed during aspect execution. We also published this analysis in [HG09].

The overhead of aspect-oriented program execution is caused by additional control flow using additional data structures. Furthermore, the overhead of the weaving process itself has to be considered, especially for languages that apply post-compile time weaving, like OT/J. One could argue that the execution time is more relevant because weaving is done only at an initial phase. Nevertheless, dynamic class loading blurs the border between these two phases. Thus, the overhead of the weaving process can also be critical during program execution.

Taking this consideration into account, we can classify the sources of overhead for the language OT/J by the following categories.

4.2.1 Aspect Registration/Activation

Aspects have to be linked in some way to the base classes they adapt. This link is needed when looking up the adapting aspect instances while executing a base method. For this, some kind of registration mechanism is needed. It is usually realized by list data structure associated to the adapted base class. In aspect languages that support dynamic *activation* and *deactivation* of aspects, these data structures have to be dynamically updatable during run-time.

Commonly, the necessary data structures and methods are added at the level of Java bytecode to the involved classes. This increases the code size, and the activation methods have to be interpreted just as normal application methods. Furthermore, OT/J supports sophisticated activation policies. Aspects can be activated explicitly/implicitly and globally/thread-locally (see [HHM07], §5). These policies cannot be efficiently implemented at the level of Java bytecode. Thread-local activation, for example, makes the execution of an aspect dependent on the current thread context. In Java, this can be realized by using the Java class `ThreadLocal` but with direct access to the thread context, this would be more efficient.

4.2.2 Aspect Execution

The execution of aspectual behavior is typically realized by additional calls to aspect instances. In OT/J, this includes looking up the next adapting team instance, determining the corresponding role object (*lifting*), and calling the bound role method. The team lookup depends on dynamic aspect activation. Dispatching from a base method to an adapting aspect method adds additional dynamic method lookup(s). This is necessary if the aspect language supports polymorphic method overriding also for aspect methods. Dynamic method dispatch is always expensive and should be avoided due to performance reasons if possible. The identification of avoidable dynamic dispatch constitutes good potential for possible optimizations. The execution of aspectual code is subject to special conditions, which does not hold for method dispatch in general. Possible optimizations can exploit these conditions (see Chapter 5).

4.2.3 Run-time Infrastructure

In the face of dynamic class loading, a post-compile time weaving process is always incremental. Therefore, information about aspects, adapted base methods, inheritance structures etc. has to be maintained during run-time. This necessitates additional data structures and causes extra overhead.

In OT/J, for example, the base-side aspect inheritance requires additional run-time information. As stated in Section 2.1.3, aspects adapting a base class do also affect subclasses. If affected methods are overwritten, also the subclasses have to be woven accordingly. In a dynamic weaving setting, the subclasses could have already been loaded by the virtual machine. To locate them, some kind of *subclass lookup* is necessary. In the current version of OT/J, run-time information needed for the weaving process is stored by means of Java data structures.

4.2.4 Advanced Modularization Mechanisms

OT/J features inheritance on the level of collaborations (teams). As we referred to in Section 2.1.3, for the roles inside a collaboration this results in a special inheritance relation to their counterpart in a super collaboration, called *implicit inheritance*, cf. [HHM07], §1.3.1. This mechanism is realized by a copy of the 'inherited' bytecode and the generation of additional interfaces. This increases the code size and necessitates the use of less efficient method invocation mechanisms. While regular methods are invoked with the bytecode instruction `invokevirtual`, for interface methods `invokeinterface` is used. The former is more efficient as in the entire class hierarchy, a constant index is used to locate the method's code in the *method lookup tables*. This cannot be guaranteed for interface methods because the implementation of additional interfaces can be declared at arbitrary points in the inheritance hierarchies of different classes. Thus, instead of direct access with a constant index, the method table has to be searched in order to locate the code of the method.

The main overhead of AOP languages is induced by the realization of aspect activation mechanisms at the level of Java bytecode of the adapted classes and by the additional dynamic dispatch for aspect execution. In the following section, we first present optimizations for the aspect activation and subsequently for the aspect execution. Moreover, in Chapter 5, we develop optimizations that utilize special conditions, which apply to the code that realizes AOP mechanisms.

4.3 Optimization of AOP Mechanisms

In this section, we present our optimizations of the aspect activation and aspect execution of the AOP language ObjectTeams. As argued in Section 4.1, we enhance the JVM to natively support AOP mechanisms to achieve this.

We propose the following extensions for the support of aspect execution inside the JVM:

- We introduce additional *data structures* to store additional information that is useful for the execution of aspectual code (e.g., aspect registration, implicit inheritance information, role caching).

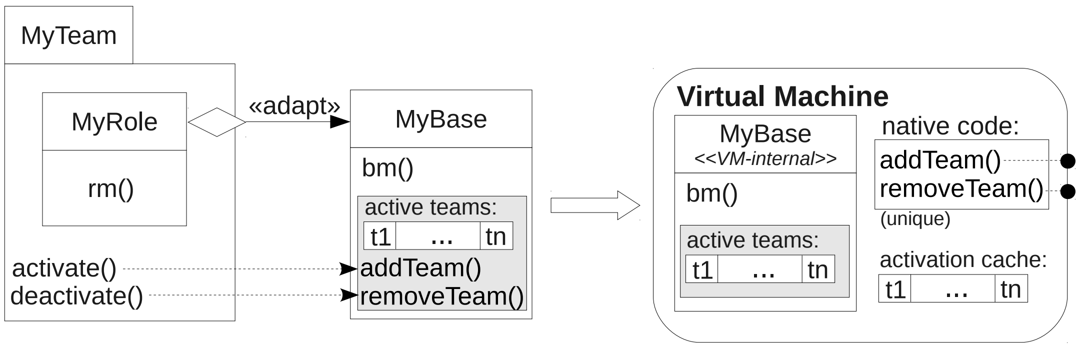


Figure 4.2: Optimized Team Activation Infrastructure

- We create and integrate additional (or adapted) *bytecode instructions* that are used to support optimized aspect execution. These instructions will make use of the introduced data structures.
- We provide an *interface* (API) to allow the weaving process to access information stored in the JVM data structures (e.g. from the run-time infrastructure).

Concrete optimizations are detailed in the following sections. To optimize aspect activation, we develop a VM-internal team activation infrastructure (see 4.3.1) and a native team activation mechanism (see 4.3.2). Furthermore, we improve the aspect execution by the introduction of special bytecode instructions for fast aspect dispatch (see 4.3.3) and efficient callin execution (see 4.3.4). We published parts of these optimizations in [HSG10].

4.3.1 Optimized Team Activation Infrastructure

In the original OT/J implementation, the infrastructure for team (aspect) activation is added to every adapted base class. The left side of the hollow arrow in Figure 4.2 illustrates that arrays for storing aspect instances and methods to access these data structures (`addTeam(Team)` and `removeTeam(Team)`) are added to the class file of every adapted base class. These methods are called by the de-/activation methods of the teams.

Similar to the *advice instance tables* of [HM05], we propose to move the aspect registration mechanism to the JVM-level, enabling various optimizations, as sketched on the right side of Figure 4.2. First, we can move the data structures to the VM-internal data structures representing classes. The methods for adding and removing teams to or from a base class can be uniquely implemented in the native code of the JVM. To allow access to this data during aspect activation, we need to extend the interface of the VM.

In a second step, we introduce a caching mechanism as optimization for consecutive activation and deactivation of the same aspect instance. This is a common execution pattern during implicit team activation. Implicit team activation has the purpose of guaranteeing a coherent aspectual control flow if a public method of a team or a role is called (cf. [HHM07], §5.3).

For future work, a promising approach is to populate the aspect registration data structures in a more context specific way. Currently, aspect instances are stored per class, but during aspect execution only a subset actually adapts the currently executed base method. Moreover, in the case of thread-local activation, this subset is further restricted by the current thread. At VM-level, the thread context is used for several purposes. Thus, it is more easily accessible also for thread-local team activation mechanisms.

By adding the aspect registration and activation mechanism to the VM-level, we can reduce the code size and enhance the run-time performance by making the corresponding methods native. Residing in the VM, these methods have access to internal data structures and mechanisms, which is another performance advantage.

Inheritance of the Team Activation Infrastructure

While moving the activation infrastructure to the VM-level, we have to guarantee the correct inheritance of active aspects to subclasses of an adapted base class. In Section 2.1.3, we mentioned that base classes also 'inherit' aspects that are defined for their super classes. To ensure this, the activation of team instances must also apply to sub base classes. At the same time, a sub base class can be adapted by additional activated aspects. However, these aspects only affect the sub base class but not the super base class. The example in Figure 4.3 illustrates such a scenario. The base class `MySuperBase` is adapted by the role `R1` of the team `T1`. More precisely, the adaptation affects the base method `bm()`. A subclass of our base class, `MySubBase`, is additionally adapted by the team class `T2`. This team also adapts the same base method `bm()`, which is inherited and potentially redefined. The subclass needs to inherit the adaptation by `T1` and it has to be guaranteed that the order of aspect execution correlates to the order of team activation (cf. Sec. 2.1.3).

In the original OT implementation, this is realized by only adding the activation infrastructure to a base class if it has not been already added to any super class. This means that in every inheritance hierarchy only the topmost base class that is adapted by an aspect contains the activation infrastructure. To guarantee this, it is necessary to analyze whether a base class has a bound base parent when the class is loaded. This information is available because when a bound role class is loaded, the role-base binding is stored and linked with its super and sub class bindings. However, because this information is stored in a global data structure it is necessary to iterate over all existing role-base bindings to access it. Since the activation infrastructure is realized by Java fields and methods, it is automatically inherited by subclasses via regular Java inheritance.

Thus, the activation infrastructure (`active teams`) in Figure 4.3 is shared among the two base classes. At the same time, this guarantees the correct order

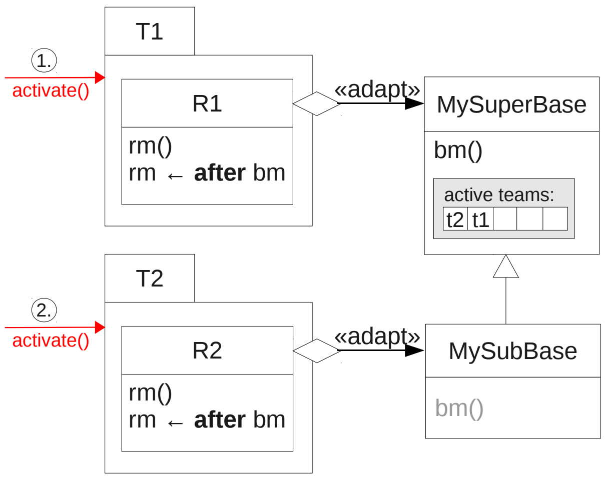


Figure 4.3: Aspect Inheritance for Base Classes

of aspect execution. In the example, first an instance of `T1` and then an instance of `T2` is activated. As detailed in Section 2.1.3, this implies that `T2` has a higher priority. The team instances are registered with the `active teams` structure of `MySuperBase`. During execution of the base method `MySubBase.bm()`, first the aspect functionality of `T2` and afterwards these of `T1` is executed. When the method `bm()` is executed at an object of type `MySuperBase` the `T2` aspect is silently ignored because in the corresponding wrapper method no code for its execution has been woven in.

With our optimization, the activation infrastructure is no longer realized by Java fields and methods, but by VM-internal C data structures related to the class representations. These data structures are not subject to Java inheritance mechanisms. Thus, we developed another approach to ensure the correct aspect inheritance at base hierarchies. We decided to realize the sharing of the team activation infrastructure among base class hierarchies, as described in Section 6.2.1.

4.3.2 Optimized Team Activation Mechanism

The activation of a team instance includes more than just registering it with the corresponding base classes. For instance, it has to be ensured that re-activation of an already active team instance does not have any effect (cf. [HHM07], §5.2.c). For the support of thread-local and global team activation, as well as for (nested) implicit team activation, more complex mechanisms are required. For example, it is necessary to keep track of individual threads and the nesting depth of implicit activation. In the original OT/J implementation, the team activation mechanism is implemented in the super class of all team classes `org.objectteams.Team` in Java. This requires several fields, such as hash maps, locks, counters and booleans.

While explicit de-/activation may occur rather infrequently, implicit activation and calls to the method `isActive`, which states whether a team is active for a given thread, constitute a relevant part during the execution of an OT/J program. To optimize the execution time of the activation mechanism, we re-implement the team activation as native VM methods. Furthermore, we move the necessary team fields to the VM-internal representation of team objects. The methods we make native cover explicit and implicit team activation and deactivation as well as the `isActive` method. Since we do not change the method signatures from the original to the native implementation, there is no difference for the programmer when using the optimized OT/J implementation.

4.3.3 Optimized Aspect Dispatch

To optimize the aspect execution, we develop a specialized *aspect dispatch* mechanism that works on the VM-internal activation infrastructures introduced in Section 4.3.1.

In the original OT/J implementation, the base class array that contains all activated teams (see Section 4.3.1) are traversed whenever an adapted base method is executed. This is necessary to call the corresponding aspect methods. Therefore, every base class contains specific *wrapper* methods for each adapted base method, as presented in Section 2.1.3. This procedure causes significant overhead because the wrapper methods have to browse the active team instances of the base class by the use of Java structures and methods. Furthermore, during the search, each contained team has to be checked for thread-local activity before an aspect can be executed.

Based on the optimization that moves the team activation infrastructure into the VM (see Sec. 4.3.1), we improve the base method execution by shifting this lookup procedure to the VM-level, too. The VM-internal information can then be accessed through a new bytecode instruction (`nextaspect`), which can be called repetitively during base method execution. With each call, this instruction returns the next active team instance in order to execute its aspect methods.

Figure 4.4 illustrates how this optimization integrates into the overall aspect execution process. Overall, we can avoid the overhead in execution time, produced by the afore used Java structures, and we can reduce the bytecode size for each base method as well.

4.3.4 Optimized Callin Execution

While the introduction of the `nextaspect` instruction in Section 4.3.3 optimizes the aspect dispatch, the arrangement and execution of the necessary aspect method calls is still implemented in the Java wrapper method (cf. `bm wrapper` in Fig. 4.4). To further optimize the aspect execution, we pro-

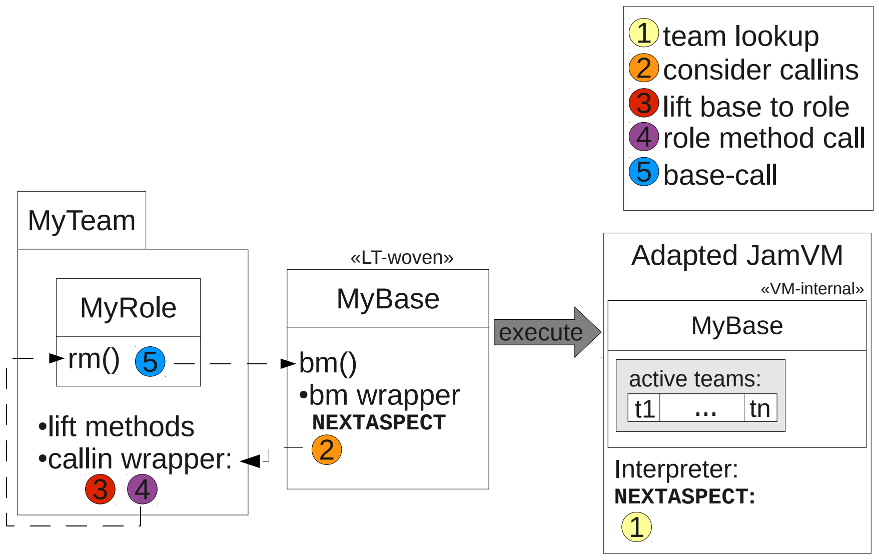


Figure 4.4: Aspect Lookup with 'nextaspect'

pose to completely shift the aspect execution mechanism into the JVM, as illustrated in Figure 4.5. We realize this by introducing the bytecode instruction `invokeaspects` ([Stö10]). This instruction is responsible for everything the base method wrapper has done before. It has to lookup the active team instances and to initiate the execution of the corresponding aspect (role) methods and the base functionality in the correct order (see Sec. 2.1.3 (Execution of Multiple Aspects)). Thus, the base method wrapper implemented in Java becomes obsolete. Instead of that, the OT runtime environment (OTRE) needs to weave the `invokeaspects` instruction into every adapted base method. As also the base-calls in replace-bound role methods need to reenter the aspect execution mechanism, they also must invoke the `invokeaspects` instructions.

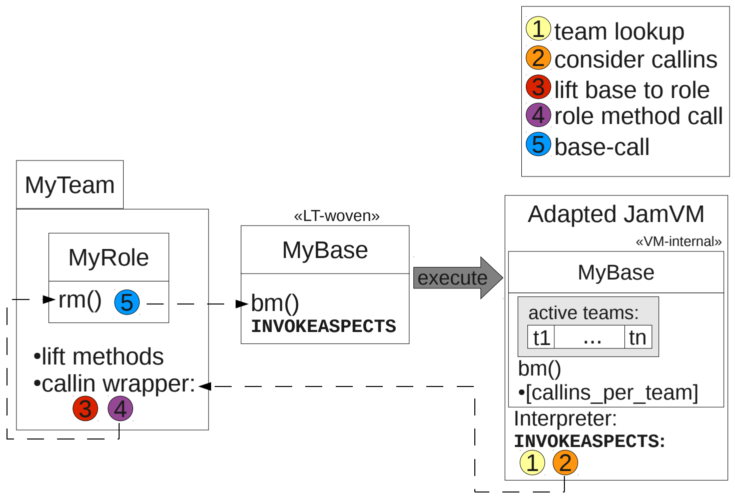


Figure 4.5: Aspect Execution with 'invokeaspects'

In order to execute an `invokeaspects` instruction, the virtual machine needs information about the role methods bound to each base method. When compiling team and role classes, the OT compiler stores this information in bytecode attributes of the corresponding class files (cf. [Hun03, Sec. 3.1] and [HH10]). We adapt the class loading mechanism to extract this information when teams and roles are loaded. For each adapted base method, we store which role methods have to be executed for each adapting team (see `[callins_per_team]` in Figure 4.5). We decided to use a *lazy mechanism* to construct this data structure. It is only constructed if the base method is executed and no entry exists for the current active team instance. On the one hand, this is a performance advantage because the overhead of constructing the entry is only spent if an instance of the corresponding team class is activated and the base method in question is actually executed. On the other hand, this approach also expands the dynamic capabilities of the language, as it facilitates the subsequent addition of teams to adapt a base method. However, at the moment this only applies to base methods which already have been adapted by some team, which is known when the program has been started. Only for such base methods, the `invokeaspects` instruction has been woven in.

In addition to this, we further optimize the lookup of active team instances (with `nextaspect`) by storing the indices of team instances that are active in the current thread. This avoids the overhead of considering team instances that are only active for other threads.

With the JVM-internal aspect execution mechanism, we are able to significantly reduce the code size of adapted base methods, as the code responsible for the execution of adapting callins is now implemented only once: in the interpreter routine of the `invokeaspects` instruction.

4.4 Dynamic Delegation Proxies

This section describes our idea to realize aspect deployment and execution by dynamically activatable proxies. In this approach, aspect dispatch is accomplished by delegation from method calls to relevant aspect code. Here, our achievements are the complete integration of aspect dispatch into the JVM, as well as efficient aspect execution and memory efficiency. Furthermore, we enhance the runtime flexibility with dynamically addable aspects and independence from the class loading order.

We investigated this approach in [Bis10]. The main idea is to introduce a method header, called *delegation proxy*, which is responsible for the aspect dispatch inside the JVM. The general concept of using delegation to realize aspect execution has been inspired by the *Delegation-based AOP* approach [HS07] which is discussed in Section 3.4.

4.4.1 Delegation Proxy Deployment

With this approach, aspects are deployed by adding and enabling a *delegation proxy* header on top of the method code. This is accomplished by the following steps, illustrated by Figures 4.6 and 4.7. When a base class `MyBase` is loaded into the JVM the code blocks of its methods are left untouched. The delegation proxy is added on top of the method code block while passing through the *prepare* phase. This happens, when a method `bm()` is executed by an `invokevirtual` instruction for the first time (see Figure 4.6). Initially, the code pointer `pc` still points to the beginning of the original code. Therefore, the delegation proxy is still ineffective and the `invokevirtual` leads to a normal execution of the called method.

The actual aspect deployment happens when a team class `MyTeam` is loaded (see Figure 4.7). If the team declares an aspect binding (`rm <- after bm`) to a method of an already executed method of a base class, the code pointer is redirected to the delegation proxy header. The next time the method is called the control is passed to the delegation proxy, which is now responsible for assuring the correct execution of aspect and base code.

Alternatively, if the team class is loaded prior to the base class, the code pointer is redirected to the delegation proxy in the prepare phase, directly. Thus, we achieve an independence from the class loading order. With the load-time weaving approach of the original OT implementation it is necessary to load teams prior to the base classes they adapt. Every team class needs to be known when starting a program and the class loading order has to be manipulated, to ensure that teams are loaded first. This manipulation is now obsolete and teams can be loaded anytime, which is a precondition for dynamic addition of aspects/teams.

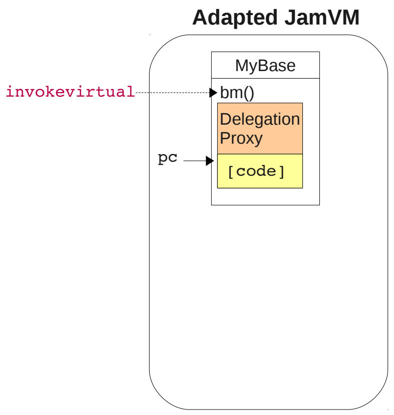


Figure 4.6: Methode Code Preparation

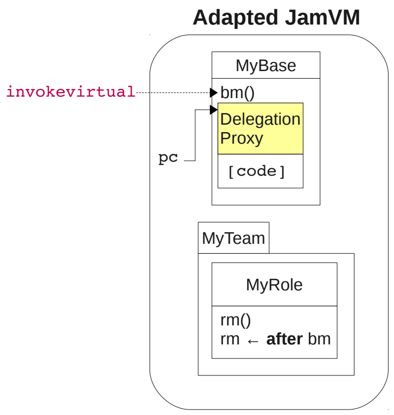


Figure 4.7: Delegation Proxy Deployment

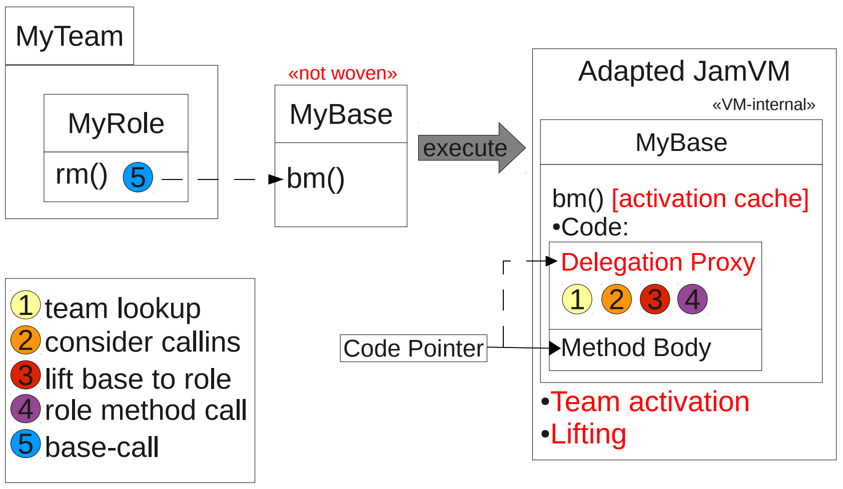


Figure 4.8: Aspect Dispatch with the Delgation Proxy Approach

4.4.2 Delegation Proxy Execution

If deployed (see Sec. 4.4.1), the delegation proxy header is responsible for executing aspect and base functionality in the specified order. Furthermore, it is responsible to build a valid activation cache containing all active team instances with aspect bindings to the current method. Compared to the base-class-wide team activation lists (see Sec. 4.2) in the original OT implementation, this is a significant improvement, as only the team instances that are relevant for the current method execution are considered.

Figure 4.8 summarizes the aspect dispatch with the delegation proxy approach. The changes regarding the original OT aspect dispatch are summarized in the following. In Figure 4.8 they are marked in red. Base classes (`MyBase`) are no longer woven before execution inside the JVM. Thus, we do not need a load-time weaving component anymore. The method-level delegation proxy is now responsible for team lookup, callin consideration, lifting and role method calls. In addition, we integrated the lifting mechanism with fast role lookup support into the JVM. Also, the team activation mechanism is natively implemented inside the JVM and activation caches per base method further optimize the lookup of active team instances.

Because the order in which team and base classes are loaded is not relevant anymore, the delegation proxy approach facilitates dynamic deployment of new teams/aspects at run-time. When a new team class is loaded, the binding information is extracted from its OT-specific attributes. They are necessary to construct VM-internal data structures that are used in the later aspect activation. This process is accomplished independent of the moment the team class is loaded. The actual dynamic deployment for a method reduces to the redirection of the code pointer from the beginning of the actual method to the delegation proxy header. This suggests a high efficiency of our dynamic weaving mechanism.

4.5 Summary

In this chapter, we presented our approach to optimize AOP mechanisms on the level of the JVM. We identified the sources of overhead that are caused by the implementation of aspect activation and aspect execution at the level of the base programming language, namely Java. To reduce this overhead, we developed optimizations for common AOP mechanisms by extending the data structures and the execution mechanisms of the JVM.

To optimize aspect activation, we introduced a VM-internal team activation infrastructure that also maintains the correct inheritance of active aspects to subclasses of adapted base classes. In addition, we proposed a native team activation mechanism that enables the efficient activation and deactivation of aspects by avoiding the use of inefficient Java means. Moreover, we improved the aspect execution by the introduction of specific bytecode instructions for fast aspect dispatch and efficient execution of aspect functionality. Hereby, we were able to make the originally used method wrapper of base methods obsolete. Further, we introduced an efficient dynamically deployable method header (delegation proxy) that is responsible for the complete aspect dispatch and establishes the foundation for dynamic AOP.

The implementation of our optimization is presented in Chapter 6 and in Chapter 7 we evaluate the benefit of our optimizations with benchmarks.

5 Semantic Information for Optimization

In Chapter 4, we suggested optimizations for the execution of aspect-oriented mechanisms. Our main focus were performance improvements achieved by the relocation of aspect-specific infrastructure code from the bytecode into the virtual machine. In this chapter, we examine how the overhead provoked by the additional abstraction mechanisms of AOP can be reduced by taking *aspect-specific semantic information* into account.

The enhanced abstraction mechanisms of high-level programming languages in general, and AOP in particular, facilitate the explicit expression of the programmers intent. When this intent is made explicit, additional information regarding the run-time behavior of a program can be inferred. We suggest optimizations that utilize the semantic conditions that apply for the execution of ObjectTeams programs. Those conditions are not necessarily valid for equivalent bytecode in general. An example is the fact that a base object is frequently lifted to the same role object. This qualifies the role object for caching, as detailed by Section 5.3.2.

This chapter is structured as follows. We start by analyzing the characteristics of abstraction mechanisms in high-level programming languages. Next, we examine general and OT-specific AOP concepts, in order to identify aspect-specific semantic information, in Section 5.2. In Section 5.3, we propose a number of concrete optimizations for ObjectTeams using the aspect-specific semantic information. Finally, we investigate how existing optimizations can profit from this additional information in Section 5.4. An overview of our approach for incorporating semantic information into the optimization of aspect-oriented mechanisms is also given in Figure 5.1, together with an indication of the corresponding sections.

5.1 Characteristics of Abstraction Mechanisms

In principle, each application can be developed with any existing programming language. However, there is a tradeoff between human intelligence and

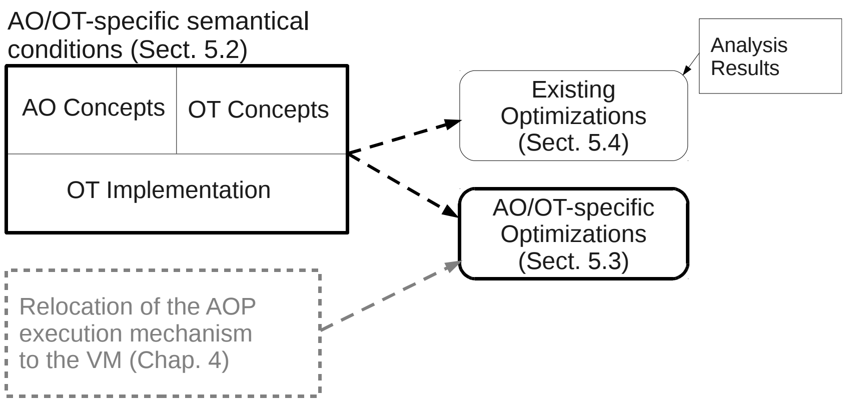

AO/OT-specific semantical conditions (Sect. 5.2)
AO Concepts
OT Concepts
OT Implementation
Relocation of the AOP execution mechanism to the VM (Chap. 4)
Analysis Results
Existing Optimizations (Sect. 5.4)
AO/OT-specific Optimizations (Sect. 5.3)

Figure 5.1: Overview of AOP-specific Optimizations

computer "ignorance". Typically, processors can only execute *concrete instructions* and are not able to cope with *abstract statements*. Machine code exactly specifies the sequence of individual instructions, which have to be processed to execute a program. Human individuals, in contrast, tend to think more abstractly, trying to *generalize* problems and to *reuse* existing solutions. This fact has always been the main motivation for the development of *high-level programming languages*. With less effort, it should be possible to develop applications that are easier to understand. Modularity concepts facilitate distributed development and better reusability. Beyond that, the source code of high-level programming languages is more *machine-independent* and better systematically *analyzable*.

Coming from *machine code*, the *imperative*, the *object-oriented* and the *aspect-oriented* paradigm each provides more and more enhanced *abstraction mechanisms*. Examples of abstract information in the (source) code of high-level programming languages are *control structures*, *methods with parameters*, *modules*, *dynamic binding*, or *aspects* with bindings to *join points*. In the following, we illustrate selected abstraction mechanisms of the different paradigms. Afterwards, we investigate the implications for the overhead of higher-level languages and how it can be reduced.

In the following, we use an example to demonstrate the consequences of introducing additional abstraction mechanisms to the (semantic) characteristics of program code. The topic of our example is a *bank account* with different functionalities to query or manipulate the *account balance*. We start by examining the transition from *imperative* to *object-oriented* mechanisms, and then look at the additional abstraction potential of the *aspect-oriented* paradigm.

5.1.1 Imperative → Object-oriented

In our example, an *account* offers functionality to query the *balance*, and to *debit* or *credit* money. In addition to basic accounts, we want to further differentiate between *savings accounts* and *checking accounts*. The functionality

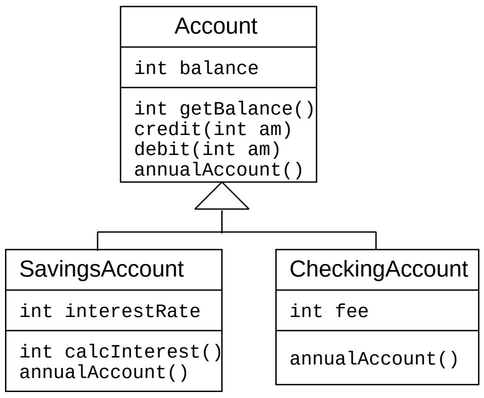


Figure 5.2: UML Diagram Account

annual account is responsible to *credit interest* to savings accounts and to *debit fees* for account management for checking accounts.

Listing 5.1 shows the C code of the account example. The common ground of the different kinds of accounts are stored in the `account_t` structure. The tag `account_tag` defines whether we have a savings account or a checking account. The specific qualities of these account types are defined in the structures `savings_t` and `checking_t`, respectively. The functions `debit` and `annual_account` each use a `switch` statement, which evaluates the value of the `account_tag` to realize the different requirements. Although the code in Listing 5.1 correctly implements the requirements, a number of problems can be observed. To introduce an additional account type, the code has to be adapted at many locations. Furthermore, the use of the enumeration type `accountType_t` can provoke failures if an account (`account_t`) is initialized with an undefined integer. In this case, the *gcc* compiler, for example, does not emit an error message and the `default` case is silently used. This can also happen if a *case* has been omitted unintentionally.

With object-oriented programming languages, like Java, the different account types would typically be arranged in a type hierarchy, as shown in Figure 5.2. The general superclass `Account` is refined to a `SavingsAccount` and a `CheckingAccount`, respectively. At runtime, the appropriate `annualAccount` method is chosen by the *dynamic type* of the account instance. The C code (see Listing 5.1) simulates dynamic binding. It is, however, not obvious that the structure of the switch is used to realize the different behavior for the account types. In the object-oriented variant, by contrast, the intent of the programmer is clearly expressed. Thus, the dynamic binding has to be implemented only once as part of the programming language and can be optimized more easily.

Listing 5.1: C Code of the Account Example

```c
typedef enum {CHECKING_ACCOUNT, SAVINGS_ACCOUNT} accountType_t;

typedef struct {
    double fee;
} checking_t;

typedef struct {
    double interest_rate;
} savings_t;

typedef struct {
    accountType_t tag;
    double balance;
    union {
        checking_t checking;
        savings_t savings;
    } kind;
} account_t;

int get_balance(account_t *account) {
    return account->balance;
}

int credit(account_t *account, double amount) {...}

int debit(account_t *account, double amount) {
    switch(account->tag) {
      case CHECKING_ACCOUNT: {
        // debit amount,
        // if balance OR credit facility do allow this
        return 0;
      }
      case SAVINGS_ACCOUNT: {
        // debit amount,
        // if balance AND savings conditions do allow this
        return 0;
      }
      default:
        return -1; /* Invalid tag */
    }

int annual_account(account_t *account) {
    switch(account->tag) {
      case CHECKING_ACCOUNT: {
        double fee = account->kind.checking.fee;
        account->balance -= fee;
        return 0;
      }
      case SAVINGS_ACCOUNT: {
        double rate = account->kind.savings.interest_rate;
        double interest = account->balance * rate;
        account->balance += interest;
        return 0;
      }
      default:
        return -1; /* Invalid tag */
    }
}
```

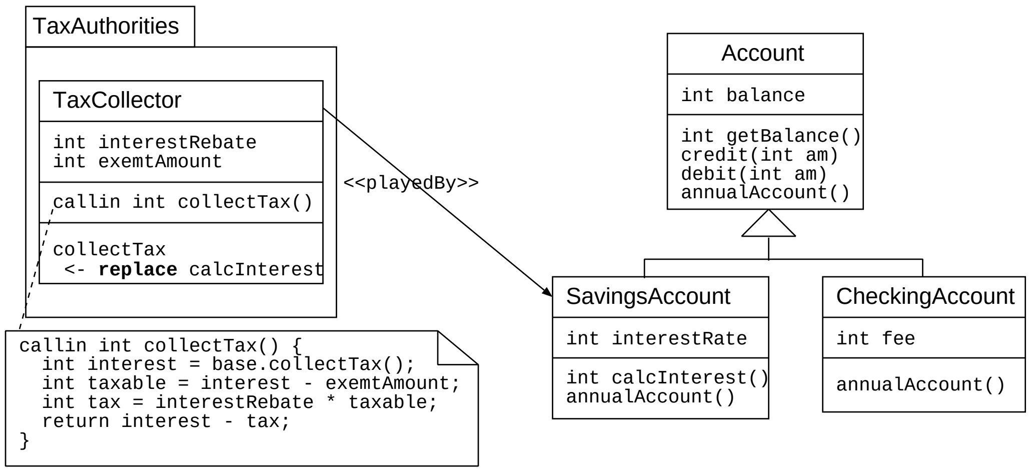
Listing 5.2: Java Code of the 'annualAccount' Method

```
void annualAccount() { // in class SavingsAccount
  double interest = calcInterest();
  // start tax concern ->
  double taxable = interest - exemptAmount;
  double tax = interestRate * taxable;
  TaxAuthorities.collectTax(tax);
  interest -= tax;
  //<- end tax concern
  this.balance += interest;
}
```

5.1.2 Object-oriented → Aspect-oriented

Now, we add the requirement that the *interest* for savings accounts has to be automatically taxed by the *tax authorities*. Every time an *annual account* is prepared, a certain *tax rate* is withdrawn from the interest, while allowing for an *exempt amount*. In Listing 5.2, this is illustrated by a call of the method `TaxAuthorities.collectTax(double t)`.

A bigger part of the method `annualAccount` is now realizing the *tax authorities* concern. In fact, the tax authorities are independent from the bank, which manages the accounts. Therefore, the *tax collection* concern should not be *tangled* (see Section 2.1) with the bank concerns. With the aspect-oriented mechanisms of ObjectTeams (see Section 2.1.3), the separation of both concerns can be realized, as illustrated by Figure 5.3. The *team* `TaxAuthorities` contains a *role* `TaxCollector`, which adapts the *base* class `SavingsAccount` by defining a *playedBy* relation. The role method `collectTax` *replaces* the base method `calcInterest`, which is used by the method `SavingsAccount.annualAccount` to calculate the interest. The role method uses a base-call to the original



```
callin int collectTax() {
  int interest = base.collectTax();
  int taxable = interest - exemtAmount;
  int tax = interestRebate * taxable;
  return interest - tax;
}
```

Figure 5.3: UML Diagram Account

method to calculate the original interest. Then, the tax is calculated and the method returns an interest value reduced by the tax.

This example illustrates how the modularization of separate concerns benefits from AOP mechanisms. Hence, the tax authorities only require the bank to provide a method `annualAccount`, responsible for crediting the interest. The `TaxAuthorities` aspect can be reused in the context of other taxes, like the *income tax*. To this end, it can be bound to the corresponding entities, like a *payroll office*.

For the aspect bindings to be effective, at a point before the execution of the affected base method, a call to the aspect code has to be woven-in (see Section 2.1.2). With a direct translation to Java bytecode, aspect-specific abstract information is flattened and can no longer be used as input for later optimizations.

5.1.3 Semantic Conditions of the AOP Example

Now, we examine the semantic conditions of the ObjectTeams example in Figure 5.2. The *tax* requirement can be reviewed at different levels of abstraction:

1. *Functional*: The interest for a savings account has to be taxed.
2. *OT program*: Every call to `SavingsAccount.calcInterest` has to be replaced by `TaxAuthorities.collectTax`.
3. *OT language implementation*: A `SavingsAccount` object is lifted to a role of the `TaxAuthorities` team. There, the `collectTax` method is called, which also calls the original functionality of `calcInterest` (*base-call*).

The following conditions apply for the corresponding OT program, but not necessarily for equivalent byte code:

- `collectTax` is only called from corresponding (playedby) base objects.
- The base object will be accessed again by the base-call.
- In the context of the `TaxAuthorities`, each `SavingsAccount` object will always be lifted to the same role object.

This analysis gives us first hints about the nature of semantic conditions of ObjectTeams programs. In Section 5.2, we investigate semantic conditions of AOP concepts more generally.

5.1.4 Summary

The additional mechanisms of abstraction, added by each programming paradigm increase the means to directly implement abstract requirements. Further-

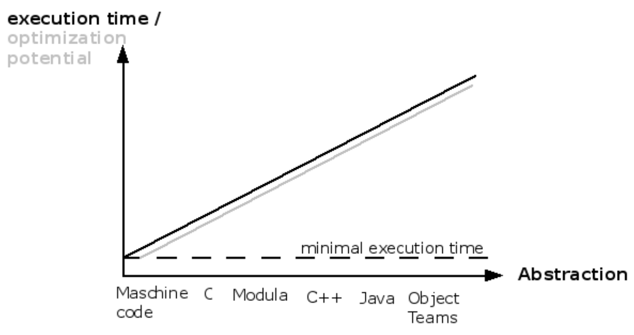


Figure 5.4: Abstraction vs. Optimization Potential

more, the additional information from the abstraction mechanisms facilitate an improved automatic error detection (cf. *dynamic types* vs. *case structures*).

A well-known drawback is the *performance overhead* provoked by additional indirections and more complex execution mechanisms (e.g., *dynamic dispatch*). Generally, more abstract programming languages are less efficient than less abstract ones. In Figure 5.4, this is illustrated by the constant black line. However, we assume that the potential for optimization increases together with the degree of abstraction, as illustrated by the gray line in Figure 5.4. This is due to the fact that the developer's intent can be expressed more explicitly by the constructs of more abstract programming languages. Therefore, more assumptions can be derived regarding the intended program behavior. These assumptions increase the potential for optimizations and allow us to reduce the provoked overhead and to reduce the execution time (see dashed line in Figure 5.4).

In the following sections, we investigate this assumption and analyze how aspect-oriented programs, especially ObjectTeams programs, can be made more efficient if the additional semantic information of the abstraction mechanisms is utilized.

5.2 Aspect-specific Semantic Information

In Section 5.1.3 we already identified semantic conditions of the AOP variant of our account example. In this section, we generally investigate semantic conditions of AOP concepts. Each optimization is eventually applied to concrete program code. Nevertheless, it is reasonable to regard aspect-specific properties at different levels of abstraction. We present the relevant properties, starting with general AOP concepts, coming to ObjectTeams-specific concepts, and finally regarding conditions specific to the ObjectTeams language implementation. By this means, we maintain a classification, indicating which properties are *language-independent*, and which are only valid for ObjectTeams, or languages with equivalent concepts. This classification can help us to identify which optimizations are potentially transferable to other approaches.

Aspect-oriented Concepts As mentioned in Section 2.1, there is a large variety of AOP languages, supporting different subsets of AOP mechanisms. However, the following characteristics are common for the majority of existing AOP languages:

- Aspects are modules, used to define *crosscutting* concerns separated from the base functionality.
- Aspect methods (*advices*) are woven into the control flow of base modules at certain points (*join points*).
- Aspect methods can be woven to *multiple* points in base modules (*quantification*).
- Base modules are not aware of potentially adapting aspects (*obliviousness*).

ObjectTeams Concepts As detailed in Section 2.1.3, the programming model ObjectTeams realizes aspect-oriented concepts by role objects, which can adapt base objects they are bound to. Role objects are contained in a surrounding team instance. Furthermore, a team and thus all its aspects can be activated and deactivated at runtime. Join points are defined by callin bindings, relating role methods to base methods. The following characteristics can be relevant for optimization purposes:

- Role objects (aspects) always belong to a surrounding team instance.
- Role classes are bound to specific base classes.
- Each role object is bound to a specific base object.
- *Lifting* ensures that each base object always relates to the same role object (relative to a given team instance).
- Callin role methods (bound with replace-callins) can only be called from the bound base method of the *playing* base class (or a subclass).

Some characteristics only apply under certain conditions. The following *heuristics* can be used for speculative optimizations:

- Many team instances are activated once and not ever deactivated again.
- Replace-callins usually make a base-call (call of the original base method).
- A callin binding ensures that a call of the bound base method implicates a call of the bound role method (provided team activation and positive guard predicates).

ObjectTeams Realization The concepts of ObjectTeams are implemented in the programming language ObjectTeams (OT/J). To realize the execution of aspects, the base code is extended by different methods and data structures, as described in Section 4.2. The bytecode generated for this purpose involves a *team registration* mechanism, *wrapper* methods, as well as tags and ids used for the *aspect dispatch*. Here, we are interested in portions of the generated code that feature different *semantic properties* than equivalent bytecode in general. Examples for this are:

- If a role defines multiple callins (after, before, replace) to the same base method, the same lifting is performed multiple times.
- Every initial wrapper is identical, except for the chaining wrapper it calls.
- The target of a base-call is exactly the same method that initiated the role method call.

5.3 AOP-specific Optimizations using Semantic Information

In this section, we propose a number of concrete optimizations for ObjectTeams, using the aspect-specific semantic information gathered in Section 5.2. In doing so, it is our goal is to optimize the aspect execution mechanism by reducing the need of dynamic dispatch. In some cases this can be achieved using caches for elements which will be used again shortly. The two major sources of (additional) dynamic dispatch are the execution of the aspect functionality and the call of the original base functionality. In the following the optimization potential of these parts is evaluated.

5.3.1 Base-Call Caching

In most cases, in the course of the aspect execution, the original base functionality is invoked at some point in time. In ObjectTeams, for replace-callins this is done by base-calls (see Sec. 2.1.3). For a base-call, the base object for the current role object has to be looked up. This is simple because every role object maintains a reference to its base object. Next, the adequate base method is called. At the bytecode level, this call is mapped to an `invokevirtual` instruction including regular dynamic dispatch. In contrast to a method call in general, a base-call is always calling exactly the method which was originally called. This means that actually, it is not necessary to dynamically look up the method which should be called. Instead, the location of the base method (e.g., its address in memory) can be stored in a *cache*, which is accessed when the base-call is processed.

Although a base-call always targets the method that triggered the execution of the corresponding role method, it might not be unique. This is the case if a role method is bound to more than one base method (of the same base class), as illustrated in Figure 5.5. For the base-call cache, this means that caching a constant single item is not sufficient. We could either dynamically update the cache, involving cache validation, or cache multiple values, one per bound base

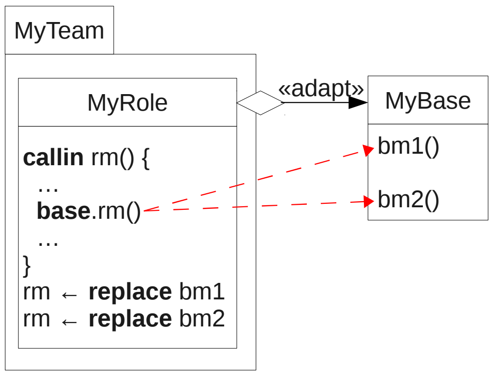


Figure 5.5: Base-Call with multiple Targets

method. Another design decision is the scope in which the cache is stored. On the one hand, this can be done relative to the role object, facilitating the repeated use during all method calls. On the other hand, the cache can be temporarily stored relative to the individual role method call (e.g., in the stack frame), reducing the memory consumption.

By adapting a base method with a role method, we introduce additional dynamic method dispatch because the role method has to be looked up dynamically. In return for this, we propose to cache the location of the base method which is a promising optimization, as it avoids dynamic dispatch.

5.3.2 Role Object Caching

Aspect execution in ObjectTeams implies the call of a *role method* implementing the *aspect functionality*. Before the role method can be called, the appropriate *role object* has to be determined. This process includes *lifting* the base object to a role object in the context of a given team. In Section 5.4.3 an optimization for the lifting mechanism itself is proposed. Here, we aim for optimizing the management of the resulting role object. Theoretically, a base object can be lifted to n role objects but in practice it is very often the same role object a base is lifted to. Thus, *caching* of the role objects is a potential source of optimization.

Lifting Scenarios

Optimal for caching in general is the *repeated* use of the *same value*. In our case, the relevant value is the cached *role object*. Because each role object belongs to a well-defined base object, at least one role object should be cached per base object. Of course, it cannot be guaranteed that an aspect execution requires the same role object than the preceding one. Thus, using a role cache has to include a check whether the cache is (still) *valid*. To identify whether role objects are repeatedly used and whether the cache becomes invalid, we examine the possible scenarios for lifting.

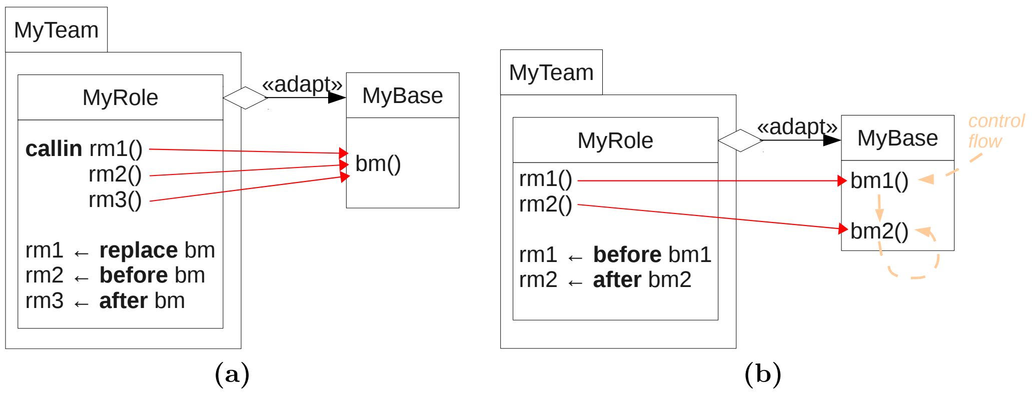


Figure 5.6: Successive lifting to the same role object.

The semantics of the lifting mechanism is defined in [HHM07], §2.3(a):

> Lifting is guaranteed to yield the same role object for subsequent calls regarding the same base object, the same team instance and the same role class. [1]

The conditions for successive lifting to the same role object are fulfilled in the following scenarios.

(i) During a single base method call (intra) if a role defines multiple aspect bindings to the same base *method*, as illustrated by the example in Figure 5.6a.

(ii) Across different base method calls (inter) if multiple callins are defined from one role object to successively called base methods. This can apply to calls of different base methods, as well as to multiple calls of the same base method (e.g., iteration or recursion) (see Figure 5.6b).

In these cases, caching the latest used role object is very profitable because every single role method call includes lifting the base object to the same role object.

Now, we consider the cases where *different* role objects are involved in the aspect execution. From the definition of lifting above follows that a base object can be lifted to another role object if the aspect execution concerns another *team instance* or another *role class*. In this case, the role cache of a given base object can be *invalidated*. Such a situation occurs if multiple callins from different role objects are defined (i) for the same base method or (ii) for different base methods of the same base object. Different scenarios for the corresponding role objects are possible. They can belong to

(a) the same role class (and team instance)

(b) another role class of the same team instance

[1]Note that smart lifting ([HHM07], §2.3.3) can result in a role object with a more specific dynamic type than the one statically requested, but this is no problem because it is the same every time.

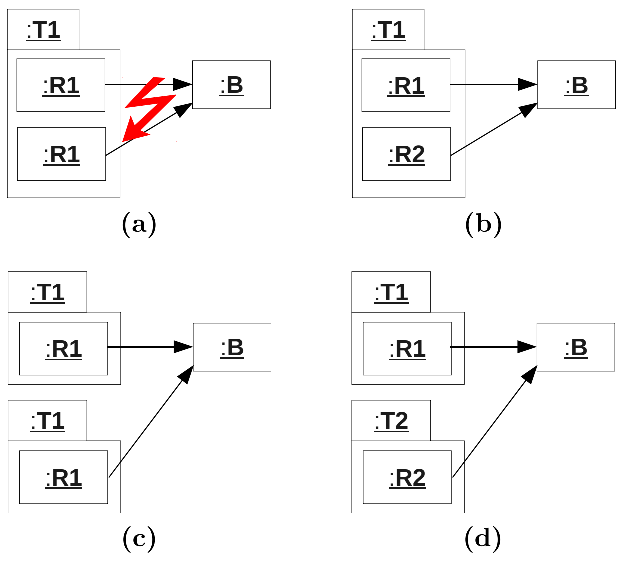


Figure 5.7: Scenarios: Different Roles for a Base Object.

(c) another team instance of the same team class

(d) another team class

Now, we look at the conditions leading to the different scenarios. The different scenarios are illustrated by Figure 5.7. Because lifting operates on the level of instances, in this figure, we use *base*, *role*, and *team objects*, denoted by the UML notation **:Type**.

(a) Multiple role objects of the same role (and team) class are bound to the same base object. As per definition of lifting, this seems to be impossible because the definition guarantees to yield the same role object, in this case (see Figure 5.7a). An exception exists due to the fact that a role object can be explicitly removed by the programmer by calling the API method `unregisterRole(Object aRole)`. In this case a subsequent lifting request causes the creation of a new role object.

(b) Role objects of multiple role classes of the same team can be bound to the same base objects, as illustrated by Figure 5.7b. However, in practice this is rather uncommon: If the roles contain separate functionality they should be in different teams because then they can be separately activated. If they contribute to the same aspect functionality they can usually be implemented as one role. Only if the aspect binding is separately defined in an inheriting team such a scenario can be essential.

(c) Role objects of the same role and team class that belong to different team instances can be bound to the same base object (see Figure 5.7c). This happens if multiple instances of the same team class are activated. However, even if possible, in practice most of the team classes are instantiated and activated only once.

(d) If different role objects adapt the same base object, they most reasonable belong to different team classes, as shown in Figure 5.7d. An arbitrary number of aspects (teams) can independently adapt the same base class (and method), each implementing a specific crosscutting concern.

The preceding analysis gives us an overview about the scenarios in which different role objects are bound to the same base object, which can lead to the invalidation of a role cache. At the same time, we give an estimation about how likely these scenarios will occur in real programs. On the basis of these considerations, we design the cache validation for our role cache, as presented in the following section.

Cache Validation

For *cache validation*, we have to identify the scenarios (a)-(d) from Figure 5.7 that imply the usage of different role objects. Because the cache validation has to be very efficient to minimize the overhead in case of cache misses, we have to carefully choose the *order* of the individual checks.

If a base class has callins from different roles, those will mostly belong to different team classes (d). Thus, we first have to check whether the cached role belongs to the expected *team class*. Next, we can either check the *role class* (b) or the *team instance* (c). Because the team class name is part of the role class name, we decided to combine the check of the role class with the check of the team class. Only if this check passes, we have to check for the correct team instance. Finally, we have to guarantee that we do not use a cached role object which has been explicitly *removed*.

This results in the following order of necessary checks:

1. team class name
2. role class name
3. team instance
4. has role been removed?

The check if a role has been removed would require access to the lifting cache inside the team. As a better alternative, we use a notification mechanism, which clears the role cache in case the `unregisterRole` method has been called.

Simulation: Overhead of Lifting

To decide whether this optimization is profitable, we did a *simulation* to estimate how expensive *regular lifting* is compared to a *cache access*. To simulate the role cache, we use a simple Java class `RoleCache` to *set*, *get* and *validate* a single cached role object. The method `setCachedRole` takes three arguments: the *role object*, the *role class name*, and the *team instance*. The `validate`

Listing 5.3: Role Lookup Simulation

```
 1 MyTeam.MyRole lookupRole(MyTeam t, MyBase b, RoleCache roleCache) {
 2   MyTeam.MyRole r = null;
 3     if (roleCache != null) {
 4       if (roleCache.validate("MyTeam.MyRole", t)) {
 5         r = roleCache.getCachedRole();
 6       } else {
 7         r = t._OT$liftTo$MyRole(b);
 8         roleCache.setCachedRole(r, "MyTeam.MyRole", t);
 9       }
10     } else {
11       r = t._OT$liftTo$MyRole(b);
12     }
13   return r;
14 }
```

method first compares the *role class name* and subsequently checks the *team instance*.

In the simple example we used for the experiment, a role class `MyRole` of the team class `MyTeam` is bound to the base class `MyBase`. Note that for simplicity, the signatures of our simulation methods contain the exact types of our example classes.

To simulate role lookup under different conditions, we implemented the method `lookupRole`, as shown in Listing 5.3. If a role cache is used, the cached role is validated with the role class name and the team instance (see Line 4). If the cache is valid, the cached role object is returned. Else, regular lifting is performed by the method `_OT$liftTo$MyRole`, in Line 7. Afterwards, the received role object is stored in the role cache (see Line 8). If no cache exists, we simply call the regular lifting method.

To simulate *cache misses*, we instantiate the `RoleCache` with a *hit ratio*, defining how often a cache access hits. A hit ratio of '1' means that the cache *always* contains the right role, with a ratio of '2' *every second* cache access is valid, and a ratio of '0' implicates that the cache *never* contains the expected role.

In our simulation, we measured the time spent to lookup role objects with no cache (*regular lifting*), a cache which is always valid (*best case*), a cache which is hit every second time, and finally, a cache which is alway invalid (*worst case*). The resulting values are the averages of 1000 runs, each with 10,000 role lookups performed on an Intel Core2 Duo processor with 2.93 GHz and 2GB RAM. We executed the role cache simulation with the Sun VM, with and without the JIT and with the JamVM. The results are shown in Figure 5.8. In the best case, the execution time for the JamVM improves by a factor of about 9 (89%). If the cache contains the right role every *second* time, the improvement is still 37% and for a hit every *fifth* time, the improvement is 7%. In the worst case, the performance degrades by 11.5%. These results indicate

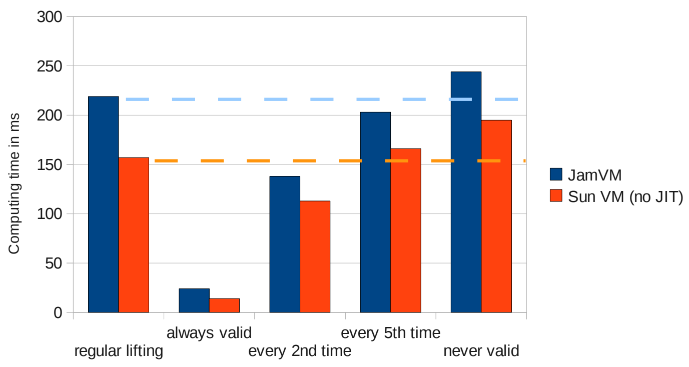


Figure 5.8: Role Cache Simulation Results

that a role cache is a worthwhile optimization to be implemented within the VM.

Discussion

After we analyzed the conditions for using a role object cache and identified that such a cache is qualified to improve the performance of the lifting process, we complete this section with a consideration about the scope for which a role cache can be established.

The simplest form of a role object cache just caches the last role object for every base object. Such a cache will use a minimal amount of memory and is profitable for base methods with multiple callins from the same role object (i) as well as for successive calls to base methods with callins to the same role object (ii).

Above, we state that in the normal case the team class is sufficient to validate cached roles. This indicates that caching one role object per team class for every base object could be reasonable. Of course, this variant has the drawback of a more complex cache structure.

Alternatively, we could introduce a separate role cache for every base method. Although this is only useful for multiple callins from one role to a base method (i), it has advantages if the individual base methods are rather adapted by different role objects. Thus, calls to other base methods do not invalidate the cached role of a base method.

5.4 AOP-specific Enhancements of Existing Optimizations

In this section, we investigate how existing optimizations can profit from additional AOP-specific information. We propose enhancements or additional applicabilities of interpreter optimizations introduced in Section 2.2.3. Afterwards, we discuss how conventional optimizations like *constant propagation*, which are typically applied by a compiler can be enhanced for AOP-specific needs.

5.4.1 Super Instructions

Super instructions (see Sec. 2.2.3) combine frequent sequences of instructions in order to reduce the number of instruction dispatches. Usually, the appropriate sequences are detected via statistical counting. Regarding certain OT-specific sequences of instructions, this process can be assisted by the knowledge that those bytecode sequences have special semantics and are not coincidently grouped together. This fact can be used as input for the algorithm that is responsible for detecting code sequences which are candidates for super instructions. An example is the implementation of the base method wrapper for adapted base methods (see Sec. 2.1.3). Here, the code sequences used to implement the team lookup, and thus realizing the aspect method calls, can be combined. In this case it is even possible to go one step further and directly merge the functionality of such sequences into a new bytecode instruction. In this sense, we introduce the super instruction `nextaspect` (see Sec. 6.3.1) in order to optimize the aspect dispatch.

5.4.2 Stack Caching

Stack caching (see Sec. 2.2.3) is an optimization for all kinds of bytecode instructions. It only depends on the stack effect of an instruction, which is determined by the number of items which are put to and taken from the stack. This optimization can be directly applied to additional aspect specific bytecode instructions. We only have to add the implementations of the instructions for the different stack states. Because stack caching is independent of the specific semantics of bytecode it is not possible to further exploit this optimization for OT specifics. The concept of caching in general is however useful in various other situations of OT-program execution, as demonstrated in Section 5.3.

5.4.3 Quick Instructions

Quick instructions are introduced in Section 2.2.3. They target instructions which involve complex initializations, like constant pool resolving of fields, methods, or classes. As the results of this initializations do not change during the ongoing execution of the program, they only need to be performed when an instruction is executed for the first time. As we propose to add aspect-specific bytecode instructions, we need to investigate which OT functionality involves initializations and could benefit from quickening. In addition to the OT-specific bytecode instructions introduced in this work, we also consider the team activation and the lifting mechanism, which could potentially be realized by additional bytecode instructions.

Team Activation At least if the activation infrastructure (see Sec. 4.3.1) is located at the base class, the first time a team instance is registered at a base class an initialization of the data structures is required. In our implementation this is done by the method `assignTAI`, which ensures that a base class also “inherits“ team instances activated for its super classes, as described in Section 6.2.1. This initialization only needs to be performed the first time a team is added to a base class. Thus, if team registration is implemented as a bytecode instruction, the quick variant can omit the initialization.

Team Lookup When executing an adapted base method the currently active teams need to be looked up. As preparation, it can again be necessary to firstly initialize the activation infrastructure. Thus, before the bytecode instruction `nextaspect` (see Sec. 4.3.3) can be used, the activation infrastructure needs to be initialized, to ensure that a team activation is also passed to sub base classes. As this instruction returns the next active aspect of the current base class, this initialization only has to be performed once. Thus, we can implement a `nextaspect_quick` instruction without the initialization.

Consider Callins When an adapted base method is called and a team class is considered for defining aspect functionality for the first time, we need to lookup every aspect binding defined in the team class for the current base method. Again, this information does not change later on and succeeding executions of the code implementing the handling of a team class can reuse it when replaced by a quick variant. This can also be applied to the `invokeaspects` instruction, which we introduced in Section 4.3.3.

Lifting If we realize lifting as a bytecode instruction, we can optimize it with a quick variant as well. The initial execution of a LIFT instruction can involve the creation of a role object. Any further execution can then be sure that a proper role object exists and thus be replaced by a quick variant that just looks up the role objects. This assumption is true except for the case that a role object has been removed by `unregisterRole` (see Sec. 5.3.2). In this case, we have to ensure that either the bytecode instruction has to be restored to the original version, or at least that a new role object is created.

5.4.4 Discussion

Conventional optimizations can be applied by classical compilers as well as by just-in-time compilers (see Sec. 2.2.3) inside a virtual machine. Examples for such optimizations are the reduction of dynamic method lookup (see Sec. 2.2.3 (Optimized Method Invocation)), the caching of values, which are (potentially) used multiple times, the re-layout of code, or the optimization of loops.

Normally, these optimizations need certain information about the *program* or *data flow*, often incorporating complex analysis techniques. Examples for information provided by such analyses are:

- A variable is never *aliased* (for *code reordering* and *constant propagation*).
- A virtual method is not *redefined* in a subclass (to avoid dynamic lookup).
- An object is local to a method (for allocation on stack).

It has to be investigated, whether the application potential of conventional optimizations can be extended if the input from the aspect-specific semantic information is taken into account. Maybe, the complexity of analyses necessary for applying such optimizations can be reduced because some conditions inherently apply for aspect-specific parts of the code. However, this is not in the scope of this thesis.

5.5 Summary

In this chapter, we investigated how the overhead provoked by the additional abstraction mechanisms of AOP can be reduced by taking aspect-specific semantic information into account. After analyzing the characteristics of abstraction mechanisms in high-level programming languages, we identified aspect-specific semantic conditions of AOP in general and of ObjectTeams in particular. Moreover, we developed a number of optimizations for ObjectTeams that utilize these semantic conditions. We proposed to use a cache to avoid the overhead of dynamically looking up the base method during a base-call of a replace-callin. Furthermore, we introduced a caching mechanism for role objects in order to optimize the lifting of base objects during aspect execution. We analyzed possible lifting scenarios in order to optimally design the structure and the validation method of the role cache. Finally, we discussed how existing optimizations, like quick instructions, can profit from the aspect-specific semantic information.

6 Implementation

We implemented the optimizations we developed in Chapter 4 and Chapter 5 as extensions to the virtual machine *JamVM* [Lou] (cf. 2.2.4). For every optimization, we incrementally adapted the weaving component of OT/J to use our new VM-internal mechanisms. With this approach, we are able to sustain the full functional range of the OT/J language at any time.

In this chapter, we start by outlining important characteristics of the implementation of the toolchain that we use as basis for implementing our optimizations. Thereby, we address the JamVM as well as the implementation of OT/J. Afterwards, we describe the prototypical implementation of our optimizations for aspect activation (Sec. 6.2) and aspect execution (Sec. 6.3). Finally, in Section 6.4, we present details of our implementation of the delegation proxy approach.

6.1 Toolchain Implementation

As motivated in Section 2.2.4, we selected the JamVM [Lou] as basis for implementing our optimizations. We used the latest version, namely version 1.5.4. The JamVM provides an efficient mechanism to access internal data and functions like class loading and reflection. At the Java-side, a class with native method declarations serves as an interface. Calls to these internal native methods are directly forwarded to implementations in the VM code. This is comparable to JNI (Java Native Interface), with a little less overhead and without dynamic libraries. We can exploit this mechanism to make new aspect-specific functionality available to the Java code, thus integrating it with the existing aspect execution mechanism.

The Object Teams Run-time Environment (OTRE) is responsible for aspect weaving in the OT/J programming language. As described in Section 2.1.3, the base code is transformed at class loading time and subsequently executed by the JVM. The OTRE works together with any standard JVM. Typically, the Sun JVM is used, which is not suitable for small devices because it is too large. We chose an adequate JVM implementation, namely the JamVM (cf. Sec. 2.2.4), which we connected to the OTRE. This configuration provides a basis for our optimizations to be implemented in the VM. This approach facilitates a direct comparison with the original OT/J implementation, allowing a precise measurement of the performance improvement. Furthermore, we can add our optimizations incrementally, while at any time sustaining the full functional range of the OT/J language for benchmarking.

The JamVM does not directly support the load-time weaving mechanism of OT/J, which uses the JPLIS API of the `java.lang.instrument` package. To facilitate the transformation of bytecode with the OTRE when executing an application with the JamVM, we had to slightly adapt the regular toolchain configuration. To this end, we use *OTEquinox* [HM07, HHP06, Her10] which is an integration of OT/J into the Eclipse Equinox framework. OTEquinox has actually been developed to support the aspect-oriented adaptation of Eclipse plug-ins with OT/J mechanisms. It uses an alternative mechanism to transfer loaded classes to the OTRE that is compatible with the JamVM. With this configuration, OT applications need to be encapsulated in Eclipse plug-ins. This produces a certain overhead that is mainly relevant for the startup phase. Note that this workaround is only necessary until the complete functionality of the OTRE is taken over by our adapted JamVM.

6.2 Optimized Aspect Activation

As we described in Chapter 4, in order to optimize the execution time of aspect activation, we developed a VM-internal mechanism for aspect registration (see Sec. 4.3.1) as well as a native team activation mechanism (see Sec. 4.3.2). In this section, we present the implementation of these optimizations.

6.2.1 VM-internal Activation Infrastructure

In [HÖ6], we identified (implicit) team activation as a significant element for the overhead of OT/J program execution. Hence, as first optimization, we have chosen the team activation infrastructure (TAI), as described in Section 4.3.1. We start by describing our realization of the VM-internal activation infrastructure, and afterwards outline our implementation of the activation cache. Furthermore, we detail our realization of the inheritance of the team activation infrastructure, which is necessary to maintain the correct inheritance of active aspects to subclasses.

Team Activation Infrastructure at VM-Level

In a first step, we moved the team registration mechanism to the VM-level. The data structure representing a class inside the JamVM was extended by a reference to the newly introduced registration data structure. In contrast to the previous weaving strategy of OT/J, now **every** class has this structure. Initially, this causes a very small overhead (20 bytes on x86 architecture). Only

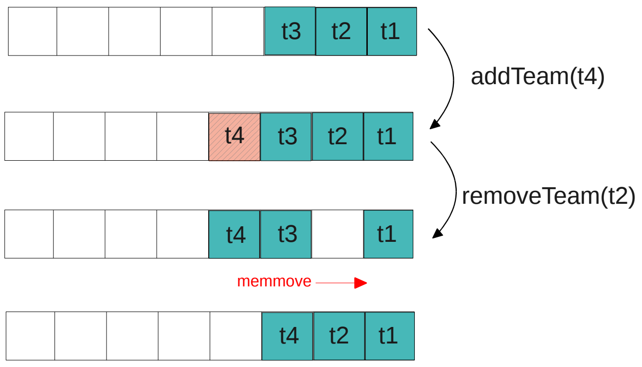


Figure 6.1: Adding and Removing Teams

if a class is actually adapted by an aspect (team), advanced initialization and memory allocation take place.

To integrate the new data structures with the remaining aspect execution mechanism of OT/J at bytecode-level, we provide an interface for adding and removing team instances to/from a base class. Additional functionality to access the array of active team instances is necessary because the remaining aspect execution mechanism resides at bytecode-level at this stage of optimization. We were able to adopt the native interface provided by the JamVM (see 6.1) to implement the interface for the team registration mechanism. Finally, we adapted the weaving strategy of the OTRE to use the new VM-internal mechanism.

Cache for Implicit Team Activation

Next, we implemented a caching mechanism for team activation. Our cache is used to optimize the consecutive activation and deactivation of the same team instance. As argued in Section 4.3.1, this is a typical execution pattern in the event of implicit team activation.

As described in Section 2.1.3, the activation semantics of OT specifies that teams which are activated at last, have to be processed at first. The latest activated team is thus added at the front of the activation infrastructure, as illustrated by the first two rows of Figure 6.1 (`addTeam(t4)`). If a team is to be removed in the event of a deactivation, the TAI is searched for this team and it is removed. Then, the remaining team instances are moved in memory to guarantee the alignment of the TAI entries. This is shown in the lower part of Figure 6.1 (`removeTeam(t2)`).

However, if a team is removed again before any other activation is accomplished, this team is still at the beginning of the TAI. In this case, as illustrated by Figure 6.2 (`removeTeam(t3)`), our caching mechanism is applied. No team is removed and no entries need to be moved in memory. Instead, we just increase the start index (`start_index`) and decrease the counter of active aspects (`aspect_count`) of the TAI. When the next team is added (`addTeam(t4)`), it

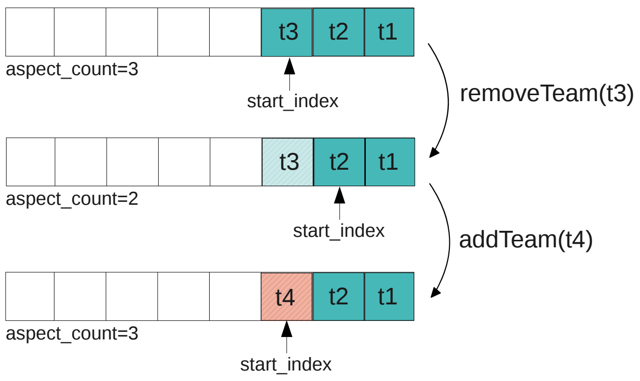


Figure 6.2: Caching Mechanism for the latest activated Team Instance

simply overwrites the afore removed one (`t3`) and the start index and the aspect counter are adjusted.

This cache stores the previous state of the activation structure when a new team instance is added to a base class. If the same team instance is removed again, before any other 'add' or 'remove' operation is performed on the same base class, the cache is written back to the corresponding activation structure. Thus, no team instances have to be moved in memory.

Initially, we proposed to use a global cache that only stores the activation state of a single adapted base class, which is written back to the corresponding activation structure if the same team instanced is directly removed again (cf. [HG09]). However, in the mean time, we discovered that it is more reasonable to directly restore the activation state. Thus, we do not need any extra cache structure and the caching mechanism works for an arbitrary number of adapted base classes.

Inheritance of the Team Activation Infrastructure

To maintain the correct inheritance of active aspects to subclasses of an adapted base class, we have to implement the sharing of the team activation infrastructure among base class hierarchies. As described above, every class features a TAI, which is uninitialized at the start. The TAI of a base class `BC` has to be *assigned* before it is used for the first time. This happens a) during team activation, when a team is added to `BC`, or b) during aspect execution, when teams are looked up for `BC`. Note that b) occurs when teams have been added to a super class of `BC` only.

When assigning the TAI of a base class `BC`, we need to check whether a bound super class `SBC` exists. If this is the case, we redirect the TAI reference of `BC` to the TAI of `SBC`. Thereby, the TAI of `SBC` might be uninitialized, as well. In this case, we first initialize the TAI of `SBC` and then redirect the TAI of `BC` to it. If no bound super class exists, we assign a freshly initialized TAI to `BC`.

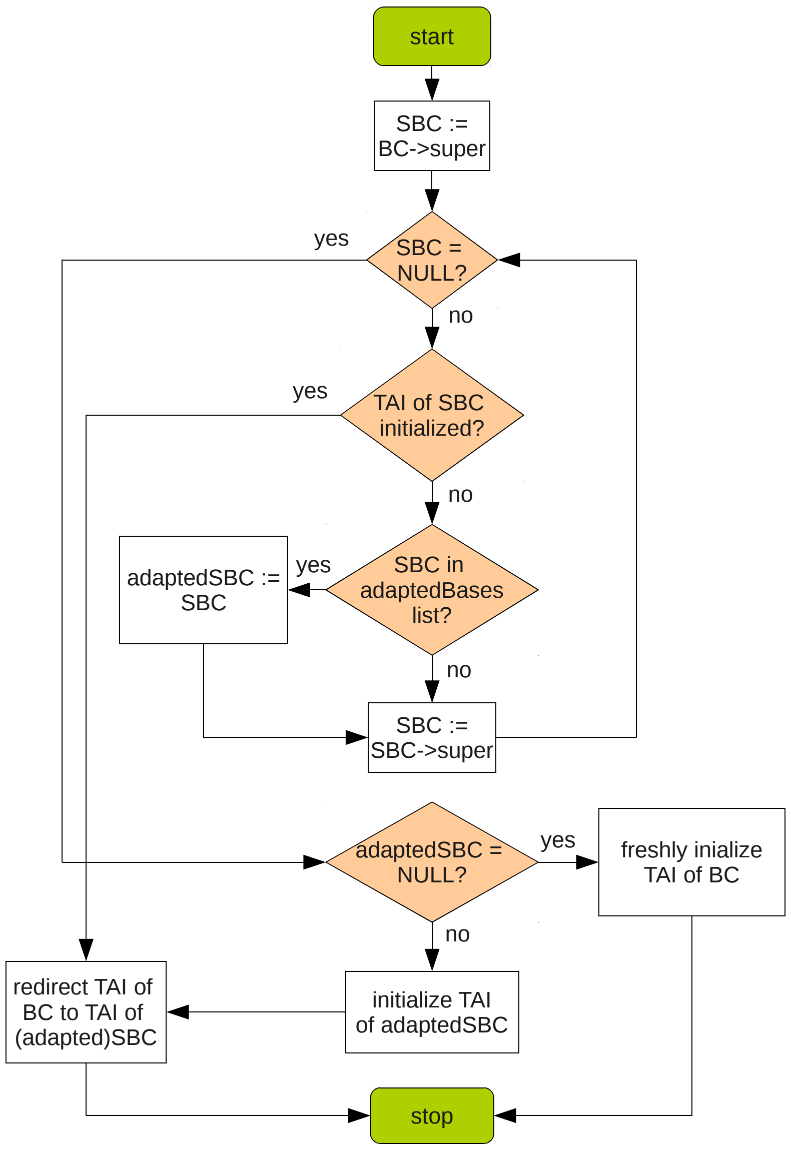


Figure 6.3: Sharing the Team Activation Infrastructure among Base Hierarchies

The algorithm we developed to realize the inheritance of the TAI is illustrated by the flowchart in Figure 6.3. Initialized TAIs of super classes can be detected by navigating the super link of a class in a loop. Bound super classes with uninitialized TAIs are more difficult to find. To enable this, we maintain a VM-global list of adapted base classes (`adaptedBases`). When a team class is loaded which adapts a base class `BC`, the name of `BC` is added to the list. When initializing the TAI of a base class, it is removed from the list. Thus, if we find the name of a super class in the list, we know that it is a bound super class whose TAI is yet uninitialized. Note that the variable `adaptedSBC` in Figure 6.3 points to the uppermost bound base class (if there is one) after the loop. Thus, we can be sure that the TAI of the base class `BC` is directly redirected to the root of bound base classes. Potentially uninitialized base classes that lay in between (regarding the class hierarchy) will be assigned when their TAI is used for the first time (see above). Here, another option would be to collect the classes in between and immediately redirect their TAIs to the TAI of the uppermost bound base class.

6.2.2 Native Activation Mechanism

As described in Section 4.3.2, we decided to implement the team activation as native VM methods. To integrate the corresponding team class fields into the teams' VM-internal representation, we created a *teamdata* structure holding the required hash maps, booleans and object locks. Now, every team object structure contains a reference to its teamdata, which can only be accessed from inside the VM. Likewise, a VM-global array for globally activated teams had to be added. Also, we had to intervene in the VM's thread handling to deactivate teams for ended threads and to activate globally active teams for all newly started threads. After that, we implemented the team activation methods analogously to the way it was done in the original Java `Team` class. A significant optimization we made affected the handling of the base class' map for activated threads. If a team gets deactivated for a single thread, now the hash map gets rearranged in case that the thread was involved in a collision before. Thus, we gain speed whenever the map is checked for an activated thread, which affects all native activation methods.

6.3 Optimized Aspect Execution

To improve aspect execution, we optimized the team lookup and the dynamic role dispatch. We introduced special bytecode instructions and implemented a cache for the recently used role object.

6.3.1 Bytecode Instructions for Aspect Dispatch

As described in Section 4.3.3, with a VM-level team activation infrastructure a further optimization stands to reason: Instead of returning an array of all active team instances, we can now adapt the VM to return only the one team instance needed for the current aspect execution. The iteration over the active teams is then done at the VM-level. To this end, we created the two new opcodes `getaspects` and `nextaspect`, to replace the original Java method calls, used to work on the base class' team list. They work on a new VM-internal data structure, the *aspect iterator*, which keeps track of the aspects to be executed. At the initial call of a base method the new bytecode instruction `getaspects` creates and returns the aspect iterator. Subsequently, `nextaspect` gets the iterator and returns the next team ID and next team instance.

Because of recursion and concurrency the aspect iterator has to exist per (base) method call. Hence, a base class can have several active aspect iterators at a time. That is why we decided to pass the iterator via the current operand stack, instead of directly connecting it to the base class itself. Furthermore, the possibility of activation or deactivation of team instances during the execution of a base method makes it necessary to copy the active aspects data structure for every base method call.

The data flow between the new opcodes and the aspect iterator is demonstrated in a simplified example for the execution of a base method with before-callins in Figure 6.4. When a base class checks for active aspects, `getaspects` is executed first. In step 1.1, the opcode takes a base class reference from the actual operand stack in order to initialize an aspect iterator. The iterator contains an array with all teams that are active for some thread and relevant for the base class (cf. TAI in 6.2.1). As shown in step 1.2, this array is taken from the VM-internal base class structure, which was extended by this array in the previous optimizations, mentioned above. It is copied to ensure that the activated roles of the base class do not change during the base method call. Furthermore, the aspect iterator holds an index counter, which is used to traverse the team array. After initialization, the aspect iterator is put on top of the stack (see 1.3).

The teams, stored in the aspect iterator, can be accessed through the `nextaspect` opcode. When the VM executes `nextaspect`, it takes an aspect iterator reference from the stack (see step 2.1). Afterwards, in step 2.2, it traverses the aspect iterator's team array and checks each team for thread-local activity by using the native methods, as discussed in the previous section. If a local-active team is found, a reference to it and its team ID is put atop of the stack (see step 2.3). Also, the index of the following team is stored in the aspect iterator's index counter to state where the lookup has to begin for the next `nextaspect` call. After the execution of the `nextaspect` opcode has finished, the returned team can be handled.

This procedure can be repeated by following step 4 to cover all teams adapting the base method call. If the end of the team array is reached during a `nextaspect` call, a null reference is put on the stack and the aspect iterator

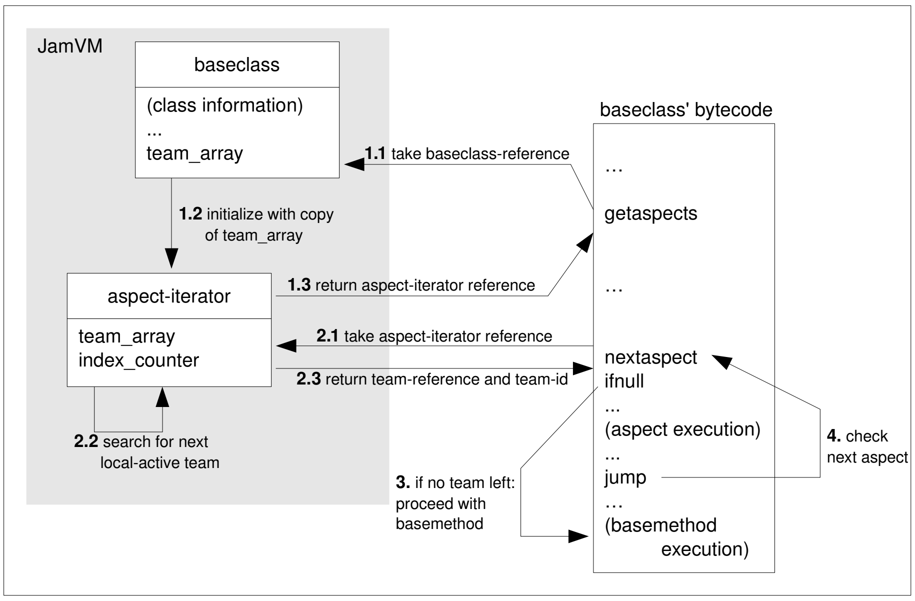

JamVM
baseclass
(class information)
...
team_array
1.1 take baseclass-reference
1.2 initialize with copy of team_array
aspect-iterator
team_array
index_counter
1.3 return aspect-iterator reference
2.1 take aspect-iterator reference
2.3 return team-reference and team-id
2.2 search for next local-active team
baseclass' bytecode
...
getaspects
...
nextaspect
ifnull
...
(aspect execution)
...
jump
...
(basemethod execution)
3. if no team left: proceed with basemethod
4. check next aspect

Figure 6.4: Aspect Dispatch with new Bytecode Instructions

is destroyed (see step 3). However, the real integration of the new opcodes into the base class' bytecode is more sophisticated than we showed in the example. For example, in our implementation, we use a recursive structure instead of a simple loop. By doing this, we achieve a more efficient coverage for all kinds of callins at once.

As an additional optimization, we implemented a *lazy copying* mechanism for the active aspects data structure of the aspect iterator. As long as no team activation or deactivation occurs for a given base class, copying the active aspects data structure can be avoided. Instead, the aspect iterator works on a reference to the original array until the activation status of a team, which is related to the base class, changes. In this case, the aspect iterator gets a copy of the active aspects before it is modified.

6.3.2 Callin Execution inside the JVM

As described in Section 4.3.4, we introduce the new bytecode instruction `invokeaspects` to shift the callin execution mechanism into the JVM. We realized this in the context of [Stö10], which comprehensively describes this optimization. Here, we only detail the implementation of the interpreter routine for `invokeaspcts`.

The OTRE generates `invokeaspects` bytecode into the wrapper methods of adapted base methods and for the base-calls of replace-callins. The algorithm we used to process the `invokeaspects` instruction is illustrated in Listing 6.1. The function `invokeAspect` receives an aspect iterator which we already introduced in Section 6.3.1. In addition to the list of active team instances the aspect iterator now maintains the arguments and the result of the base method call, as well as some indices needed for the algorithm. While executing the different kinds of callins, the algorithm is called recursively for each active team instance as well as to realize base-calls of replace-callins.

The `calling_replace_index` in Line 2 is used to differentiate between calls of `invokeaspects` which have been directly initiated by an adapted base method and those originating from base-calls of replace-callins. In the latter case, only the next replace-callin of the currently processed team instance is processed. The structure of the algorithm mainly resembles the base method wrapper implementation of the original OTRE. Initially, we check if all team instances have been processed in Line 4. In this case, the aspect execution is finished and the original base method is executed. Otherwise, as introduced in Section 4.3.4, we apply the lazy mechanism for assigning the binding information of the current team to the executed base method (see Line 10-12).

Listing 6.1: Algorithm for *invokeaspects* (Origin: [Stö10])

```
 1 void invokeAspects(AspectIterator* ai) {
 2   int calling_replace_index = ai->replace_index;
 3 
 4   if (ai->current_index >= ai->aspects->aspects_count) {
 5     ...          //execute base method
 6     return;
 7   }
 8 
 9   //teams callins not in place -> lazy lookup mechanism
10   if (list of team's callins does not exist) {
11     ...          // lookup callins in team attributes
12   }
13 
14   if (ai->calling_replace_index == 0) {
15     ...          // execute before callins for actual team
16   }
17 
18   if (replace callins are left) {
19      ai->replace_index++;
20     ...          // execute next replace callin for actual team
21   } else {
22     ai->current_index++;
23     ai->replace_index--;
24     invokeAspects(ai);
25   }
26 
27   if (calling_replace_index == 0) {
28     ...          // execute after callins for actual team
29   }
30 }
```

Next, the actual execution of the callins defined for the current team is initiated. We begin with the execution of the before-callins (see Line 15). As described above, this step is guarded by the `calling_replace_index`, which is equal to zero for regular invocations.

Starting with Line 18, the replace-callins are processed. The aspect iterator's `replace_index` is incremented to indicate that the algorithm is now in the mode of only executing replace-callins. Next, the role method of the replace-callin is called. If this contains a base-call, `invokeAspects` is recalled recursively. If no replace-callins are left (see Line 21), the iterator's index (`current_index`) is incremented to point to the next active team. The `replace_index` is reset and `invokeAspects` is recursively called to process the callins of the next active team instance. After returning from the recursive calls for processing the replace-callins, the after-callins are executed in Line 28.

6.3.3 Role Object Caching

As described in Section 5.3.2, we implemented a single role cache per base object to optimize the execution time for base object lifting. This cache had to be integrated into the existing lifting process of OT/J. When an object is allocated, we do not want to make any assumptions whether it will eventually be adapted and thus becomes a base object. This also facilitates dynamic aspect weaving (see 2.1.2). Hence, we integrated the cache into the general object layout of the VM. Initially, we do not allocate any memory, but set the field to `NULL`. The allocation takes place when the cache is used the first time. So, the overhead for normal objects is minimal (size of a pointer, e.g., 4 bytes on 32 bit architectures).

In addition to the cached role object, the cache data structure stores the qualified role class name (including the team class name) and the associated team instance. This information is necessary for cache validation, which first checks the role and team class name and afterwards compares the team instances, as argued in Section 5.3.2.

The corresponding data structure has the following form:

```c
/* The RoleCache is used to store the last role */
/* to which a base object has been lifted. */
typedef struct role_cache {
  Object *cached_role;
  char *role_class_name;
  Object *team_instance;
} RoleCache;
```

Garbage Collection Issues

The reference to the cached role points to a Java object, which resides inside the heap. When the garbage collector *compacts* the heap, it modifies the addresses of objects inside the heap. In the process, the Java references to the moved objects are updated to point to the new addresses. Garbage collection is a challenge for our role cache, as the native reference to the cached role in the `RoleCache` structure is not part of the heap and will thus not be automatically updated. To prevent the role reference from becoming invalid, we have to register this object reference from outside the heap with the garbage collector by calling the JamVM-internal function `registerStaticObjectRef`.

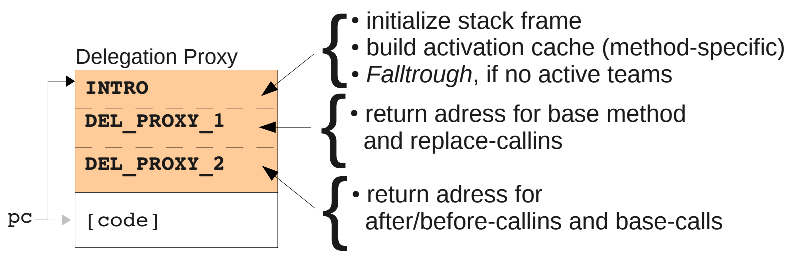

Delegation Proxy
INTRO
DEL_PROXY_1
DEL_PROXY_2
pc
[code]
• initialize stack frame
• build activation cache (method-specific)
• *Falltrough*, if no active teams
• return adress for base method and replace-callins
• return adress for after/before-callins and base-calls

Figure 6.5: Delegation Proxy Layout

Integration into the Lifting Process

To grant the lifting process access to the role cache, we introduced native interface methods for storing and retrieving the cached role. To integrate the role cache with the lifting process of OT, it has to be used at every lifting site. The role cache could be checked before every call of the lifting method, but in this case, the many places where this can happen have to be located (lifting call targets/parameters at callin, lifting objects at assignment). A simpler way of integration is to adjust the lifting method itself. The generation of the lifting method is done when a team class is compiled. So, to integrate the usage of our role cache, we adapted the code generation of the OT/J compiler. Before a role is created or looked up in the team-internal data structures, our role cache is consulted. If the cache contains a valid role object, it is instantly returned. Otherwise, the original functionality of the lifting method is executed. In this case, the calculated role object is stored in the role cache before it is returned.

If a role has been removed from a team instance by calling the API method `unregisterRole(Object aRole)`, a subsequent lifting request causes the creation of a new role object. In this case, the cached role object must be removed. In our current implementation, the unregistration method had been adapted to set the cached role of the affected base object to `NULL`. This is an over-approximation because the actual cached role object may also be independent of the removed one. In return, it is very efficient because it does not involve any additional check.

6.4 Delegation Proxy Approach

In Section 4.4, we introduced our approach to realize aspect deployment and execution with dynamically activatable delegation proxies. Here, we further describe the implementation details of this approach, which we prototypically integrated (cf. [Bis10]) into the JamVM (JamVM/OTRun).

The layout of the delegation proxy is illustrated in Figure 6.5. It is a sequence of three opcodes: an `INTRO` entry is followed by two `DEL_PROXY` entries. The `INTRO` instruction initializes additional stack frame entries needed to pro-

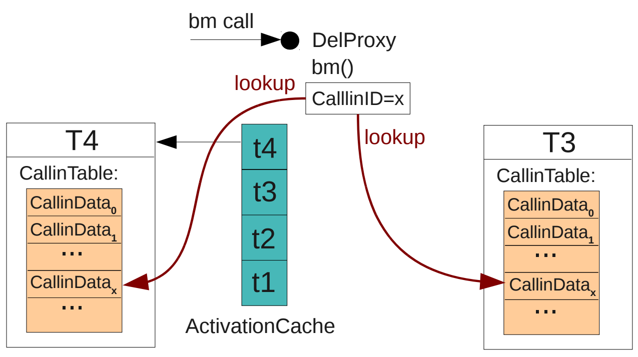


Figure 6.6: Looking up relevant callins

cess the callin execution. If no team instance is active the `INTRO` entry triggers a fall-through to the original base method code, causing no extra overhead.

The two `DEL_PROXY` opcodes are responsible for the actual aspect dispatch. The difference between the two variants lies in the handling of return values. In Object Teams, replace-callins can change (replace) the return value of their base method. After- and before-callins, in contrast, are purely additive. Thus, potential results of the bound role methods are ignored. The first one `DEL_PROXY_1` is the return address for calls which may influence the return value of the called base method. This is the case for the original base method as well as for replace-callins. The second entry `DEL_PROXY_2` handles control flow returning from after and before-callins. In addition, base-calls return to this entry.

The `DEL_PROXY` opcodes are responsible for the execution of the callins of every team instance which is active when the base method is called. This comprises two tasks:

1. The lookup of the relevant callins of the next active team instance
2. The orderly execution of the relevant callins in a switch statement

The lookup of the relevant callins (1.) is illustrated in Figure 6.6. Team classes contain a `CallinTable` with entries (`CallinData`) for every base method adapted by their roles. A `CallinData` entry contains all information about the callins to be executed. This includes the execution order and the lifting of the base object to proper role objects. To lookup the `CallinData` of the next active team instance (`t4`), the `CallinTable` of the corresponding team class (`T4`) is consulted. Our approach facilitates a very efficient access. Every base method has a unique ID `CallinID` used to directly access the `CallinData` of *every* relevant team class. To ensure this, some effort is put in the construction of the `CallinTables`, which are expanded and rearranged if necessary.

The callin execution by the delegation proxy is illustrated in Figure 6.7. In a switch statement, the entries of the current `CallinData` are executed. A typical sequence is shown in the `CallinData` for team instance `t4`. First, the base object is lifted (L) to the proper role object. Next, a before-callin (B) is

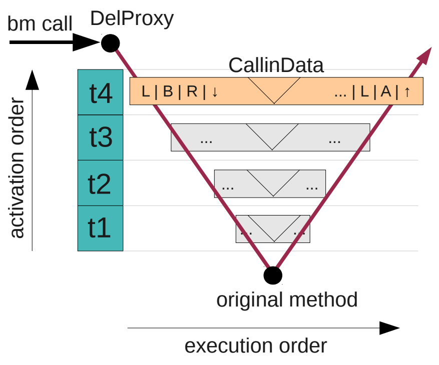


Figure 6.7: Callin Execution via the Delegation Proxy

executed. Afterwards, a replace-callin (R) with base-call is executed. Now, we descend to the next team instance (↓). When the control flow returns from this instance, the base object is again lifted (L) and an after-callin (A) is executed. Finally, we ascend (↑) to the previous team instance.

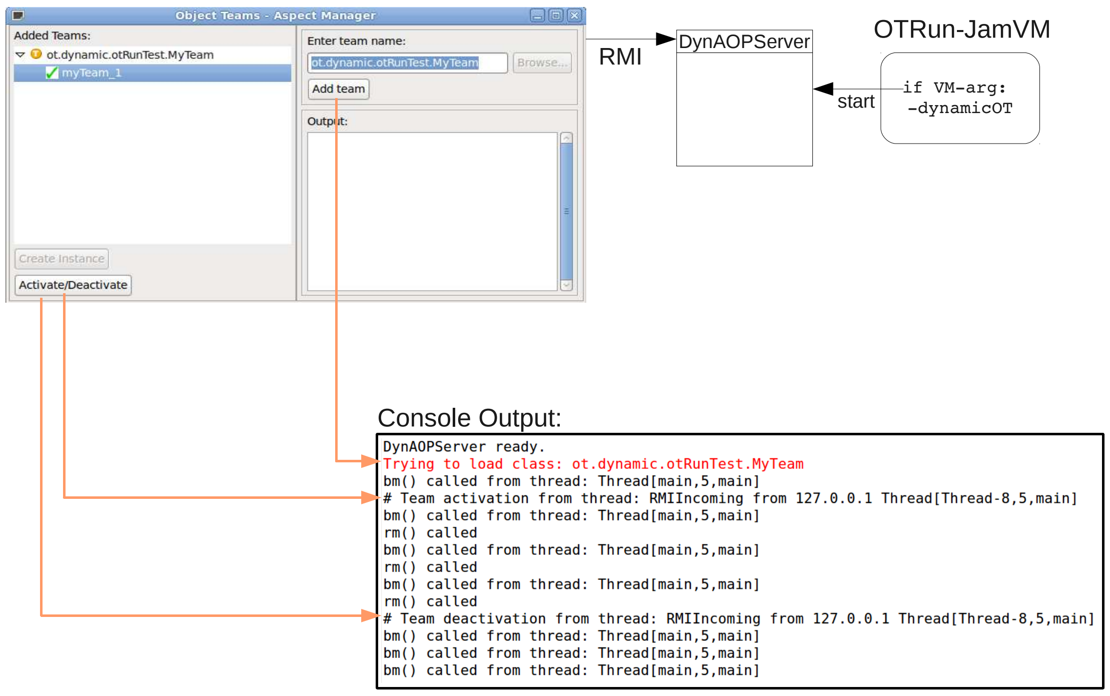
The integration into the JVM is minimal-invasive, and is mainly restricted to the following parts.

- The class loading mechanism has been adapted to integrate the analysis of the binding information, and to create and initialize the additional data structures.
- Some VM-internal data structure like the stack frame and the method block have been enhanced.
- The interpreter has been adapted to facilitate infrastructure weaving when executing the `invokevirtual` instruction.

Note that in contrast to the implementations of the optimizations presented in Section 6.2 and 6.3, we implemented the delegation proxy approach in a separate branch. This is due to the fact that here, our focus is the complete integration of aspect dispatch into the JVM and the enhanced runtime flexibility. Thereby, we completely restructured the aspect execution mechanism. The resulting prototype realizes efficient dynamic aspect deployment but yet does not support the full scope of the OT/J language. In future work, we plan to integrate the two implementation branches.

Dynamic OT Experiment

To evaluate the delegation proxy's capability for dynamically adding (and removing) new aspects at runtime, we implemented a prototypical dynamic AOP component. Figure 6.8 gives an overview of our experimental configuration. We added the VM flag `-dynamicOT` to the JamVM. If this flag is set, a *dynamic AOP server* (`DynAOPServer`) is started. This server offers an interface to *add*, *activate* and *deactivate* new aspects/teams to the running application. In addition, we implemented an `AspectManager` UI tool, illustrated by the



```
DynAOPServer ready.
Trying to load class: ot.dynamic.otRunTest.MyTeam
bm() called from thread: Thread[main,5,main]
# Team activation from thread: RMIIncoming from 127.0.0.1 Thread[Thread-8,5,main]
bm() called from thread: Thread[main,5,main]
rm() called
bm() called from thread: Thread[main,5,main]
rm() called
bm() called from thread: Thread[main,5,main]
rm() called
# Team deactivation from thread: RMIIncoming from 127.0.0.1 Thread[Thread-8,5,main]
bm() called from thread: Thread[main,5,main]
bm() called from thread: Thread[main,5,main]
bm() called from thread: Thread[main,5,main]
```

Figure 6.8: Dynamic OT Experiment

screen shot in the upper right of Figure 6.8. The aspect manager communicates via RMI with the dynamic AOP server. We are able to choose new team classes and add them to the application, as shown in the right side of the aspect manager's UI. Afterwards, new instances of these team classes can be created, activated and deactivated (see left side of the UI). Note that the aspect manager can be started on demand anytime, after the application has been started with the `-dynamicOT` flag.

We used our prototype to evaluate the dynamic capabilities of the delegation proxy approach with a simple OT application. The program output is shown in the Console Output in Figure 6.8. An initially unadapted base method is called in an endless loop, producing the output "bm() called from thread...". Then, a team with an adapting role method is dynamically added and activated. The program output shows that the callin is executed. After every base method call, we see the output "rm() called". Deactivating the team instance again results in the original output of the unadapted base method.

This experiment demonstrates that the delegation proxy approach brings in the dynamical addition, activation and deactivation of new (unknown) aspects at runtime. Thus, we could substantially enhance the dynamic capabilities of ObjectTeams.

6.5 Summary

In this chapter, we presented our implementation of the optimizations of aspect activation and aspect execution, which we introduced in Chapter 4 and 5. For example, we illustrated details of our caching mechanism for implicit team activation as well as of our realization of the inheritance of active teams to sub classes. Furthermore, we described our implementation of the additional bytecode instructions for team lookup (`getaspects` and `nextaspect`) and for callin execution (`invokeaspects`). In terms of our role cache realization, we also discussed garbage collecting issues and the integration into the lifting process. For the delegation proxy approach, we detailed the processes of looking up relevant callins and of executing callins from a number of active team instances. Finally, we presented an experiment that confirms the delegation proxy's capability for dynamic aspect deployment.

7 Evaluation of the Approach

In this chapter, we evaluate our approach and demonstrate the benefit of our optimizations. In order to assess the effect of our optimizations, we used different kinds of programs. We executed micro-benchmarks to measure the benefit for isolated AOP mechanisms, as aspect activation or aspect execution. The results of our experiments are presented in Section 7.2. However, the improvement for real-world applications depends on to what extent the optimized AOP mechanisms are used in the overall program. For this reason, we further evaluated our optimizations with a case study presented in Section 7.3. There, we executed the game application *OTPong* and measured the execution time of updating the game's view. Finally, in Section 7.4, we discuss the transferability of our approach to other VM implementations on the one hand, and other AOP languages on the other hand. Before we present our experimental results, we start this chapter by discussing the parameters and the general set-up of our benchmarks in Section 7.1.

7.1 Benchmarking Issues

In this section, we discuss the set-up of our benchmarks and experiments. We outline relevant facts regarding benchmarks for Java in general and AOP in particular. Finally, we specify the general set-up of the experiments we carried out to evaluate our approach.

7.1.1 Benchmarking for Java

There are some issues which have to be considered when benchmarking Java programs. [Boy08] discusses these factors in detail. As ObjectTeams is based upon Java, we need to consider all the factors that influence the benchmarking of Java programs. In the following, we summarize the most important influence factors together with our effort to cope with them.

Warmup Phase Usually, we are interested in the *steady-state* performance when benchmarking Java code. That is, we have to avoid the influence of startup and initialization tasks like class loading or method preparation. For this, we accomplish a warmup phase before the measurement of our benchmarks takes place. In this phase, we complete the creation of participating objects, including the initialization of teams. Moreover, we perform initial activations and subsequent deactivations of participating team instances. Finally, the methods called during the benchmark are initially executed a number of times before we start the measurement.

Resource Reclamation The results of a benchmark can also be affected by the current memory workload. To minimize this influence, we run each benchmark in a freshly started JVM instance.

Data Set Size Due to CPU caching, the size of the used data sets can affect the results of the benchmarks. To cope with this factor, we use different data sets, varying the number of active team instances, the number of callin bindings, and the number of active threads.

Preparation To avoid the effect of power management and other running programs, we stopped all other programs including the X window system and executed the benchmarks in a linux console.

Numerical Issues In order to achieve meaningful results, the figures we present in our benchmark results are the arithmetical mean of 10 runs. As recommended by [Boy08], we use the Java API method `System.nanoTime()` instead of `System.currentTimeMillis()` because of its higher resolution.

7.1.2 Benchmarking for AOP languages

For Java and the JVM, there exist standardized benchmark suites such as JavaGrande [Jav] or SPECjvm2008 [SPE]. For AOP languages the situation is different. As stated in Section 2.1.1, there is a large number of existing AOP approaches. Most of the performance evaluation for AOP has been done for the AspectJ language [ajH]. Of course, for most of the other existing AOP languages, individual benchmarks have been published, but the comparability among the different approaches is limited. Often only a comparison to AspectJ is done. This is due to the fact that even the common AOP mechanisms like *binding an aspect method to a base method* are expressed differently in most of the approaches. Thus, a general benchmark suite for AOP would have to provide many different versions of the benchmark programs.

At least for AspectJ, some attempts of preparing a benchmark suite do exist. [Asp] contains AspectJ programs from different application domains, like a product line for related graph algorithms and a variant of the arcade game

Tetris. [AOP] describes micro-benchmarks that are executed with different AOP languages to compare their performance. Most of them could be ported to an OT/J implementation but as the analyzed languages do not support the concepts of roles or team activation, they are only partly applicable to evaluate our approach.

As a consequence, we decided to use our own micro-benchmarks for evaluating the effect of our optimizations on the different AOP mechanisms. [HM04] presents a micro-measurement catalog for dynamic AOP languages. According to their suggestions, we perform the following micro-benchmarks to evaluate the performance benefits of our approach. On the one hand, we measure the cost of dynamic aspect activation in different contexts. On the other hand, we evaluate the cost of executing base methods which are:

1. unadapted
2. adapted (after, before, replace bindings)
3. only adapted in other threads

Also important in this regard is the *distributed fat* of the different AOP mechanisms. This is the inherent overhead of a language feature, when it is not used. In an optimal setting, this overhead is zero. This motivates the *zero-overhead principle*: ”what you don’t use, you don’t pay for” [Str94, p. 121]. Thus, an ObjectTeams program that only uses regular Java features should execute as fast as if it is executed with a regular Java execution environment. But also if OT/J specific mechanisms are used in a program, this should not influence the performance of parts that do not use them. For OT/J, this means that the execution time of methods that are not adapted by aspects should not degrade. In this regard, we also have to consider the dynamic activation state of aspects and minimize the overhead of inactive aspects. We analyze the distributed fat of OT/J for method execution in Section 7.2.7.

7.1.3 Experimental Set-up

In this section, we describe the general set-up of our experiments. We start by describing the toolchain we used to build, configure and execute our benchmarks. Furthermore, we discuss how we measure the execution time and specify the configuration of the machine we use to run our experiments.

As discussed in Section 6.1, the JamVM does not support the JPLIS-based weaving mechanism of ObjectTeams. Thus, we use OTEquinox to execute our benchmark programs. Figure 7.1 gives an overview of our benchmarking toolchain. To enable the use of OTEquinox, we wrap our benchmark programs inside an Eclipse Plugin (*BMPlugin*). In this plugin, we also define an *Eclipse Application* in order to enable the execution of our benchmarks from outside the Eclipse IDE. This is necessary because we want to prevent any influence of running programs on our results.

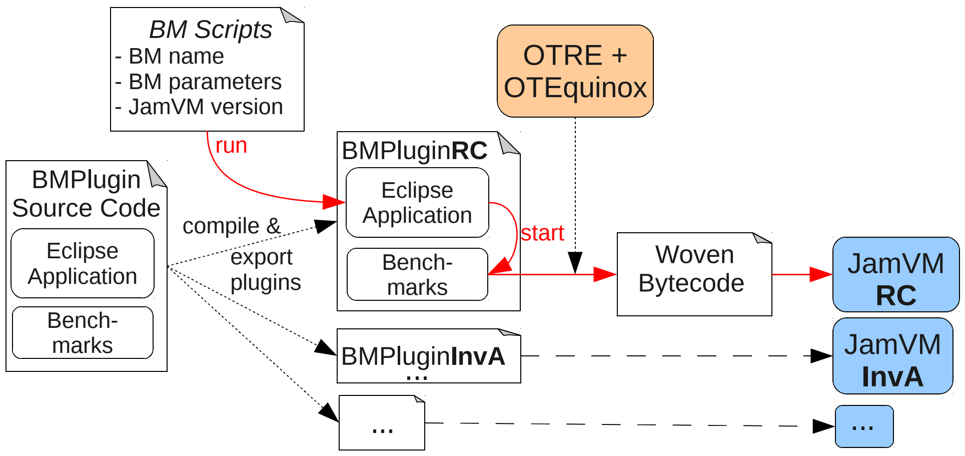


Figure 7.1: Benchmark Execution Toolchain

In our experiments, we want to identify the effects of each individual optimization. Thus, we need to maintain different versions of the JamVM, where each supports a particular subset of our optimizations. At the right of Figure 7.1, *JamVM RC* stands for a VM version with enabled role cache, while in *JamVM InvA* the `invokeaspects` instruction is supported. In addition, we also had to compile and export different versions of the benchmark plugin because some optimizations require the compiler to generate different code. Hence, in the plugin *BMPluginRC* of Figure 7.1, the lifting methods of teams are implemented to use our role cache.

The scripts (*BM Scripts*) we used to automate the execution of the benchmarks are designed to perform the independent execution of each benchmark with different optimizations enabled. We use runtime properties to configure the parameters of each benchmark run. For example, the number of active team instances or the number of involved threads can be varied. Also the name of the benchmark to execute (*BM name*) is passed via a runtime property. This way, we can combine all benchmark programs in a single BMPlugin. To execute an individual benchmark, the scripts run the application and pass all the necessary parameters. The application starts the benchmark by calling the main method of the corresponding benchmark via reflection. During class loading, OTEquinox induces the necessary bytecode weaving performed by an adapted version of the OTRE. Finally, the benchmark code is executed by the designated JamVM version.

Our benchmarks are measuring the execution time of selected code parts. In Section 7.2, we describe the experimental results of executing micro-benchmarks that are constructed to individually measure the execution time of selected AOP mechanisms. In Section 7.3, we furthermore evaluate the execution of a real application. To measure the execution time, we used the Java API method `System.namoTime()`, as illustrated in Listing 7.1.

The time is measured before and after the statements we want to evaluate. The elapsed time is calculated by subtracting the before-time from the after-

Listing 7.1: Execution Time Measurement

```
// exexute warmup phase
long tBefore = System.nanoTime();
// execute code to benchmark
long elapsed = (System.nanoTime()-tBefore)/1000000L;
```

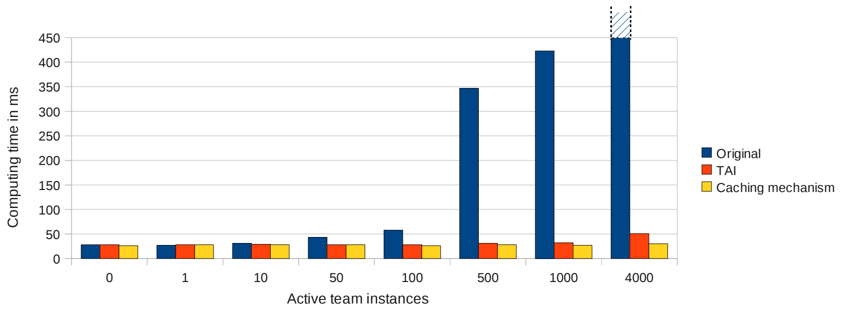
time. Thus, the startup overhead of using OTEquinox is excluded because we do not start our measurement until the application has already been started. However, with OTEquinox much more eclipse-specific classes are loaded before the actual benchmark code is executed, but we do not expect that the number of loaded classes has any influence on the execution performance of our benchmarks.

We performed each benchmark for each optimization multiple times and took the arithmetic mean of 10 runs as result for our charts. Finally, we benchmarked all optimizations together in Section 7.2.6. We executed the benchmarks on a desktop PC with an Intel Core2 Duo processor with 2.93GHz and 2GB RAM using the linux kernel version 2.6.32. As our approach aims at enabling the use of AOP mechanisms in the context of embedded mobile systems, we furthermore executed the benchmarks on Openmoko's Linux-based smart phone FreeRunner [Fre], which features a 400Mhz ARM processor, 128MB SDRAM memory and 256 MB integrated flash memory.

7.2 Evaluation with Micro Benchmarks

In this section, we present the results of evaluating the effect of our optimizations with several suitable *micro benchmarks*. As our goal is to reduce the AOP-specific overhead, each of these benchmarks measures a separate part of the AOP mechanisms, affected by one of our optimizations. To isolate the effect of the respective optimization, the involved aspect methods and base methods have empty bodies. The actual improvement for real-world applications depends on to what extent the optimized AOP mechanisms are used. In Sections 7.2.1–7.2.7, we present the results of executing our benchmarks on the Intel PC and in Section 7.2.8, we validate these results on the ARM-based smart phone.

In addition to the execution time, we also analyzed the effect on code size. In general, the size of runtime structures of classes and objects is more relevant than the woven base bytecode because the latter is only held in memory during class loading. So, if data fields (e.g., the lists of activated team instances) are shifted from the bytecode to the runtime structures, this only reduces the bytecode size. Reducing method code, on the other hand, also leads to smaller runtime structures. Therefore, we focus on method code when describing the effect of our optimizations on code size.



Active team instances	0	1	10	50	100	500	1000	4000
Original	28	27	31	43	58	347	423	2427
TAI	28	28	29	28	28	31	32	51
Caching mechanism	26	28	28	28	26	28	27	30

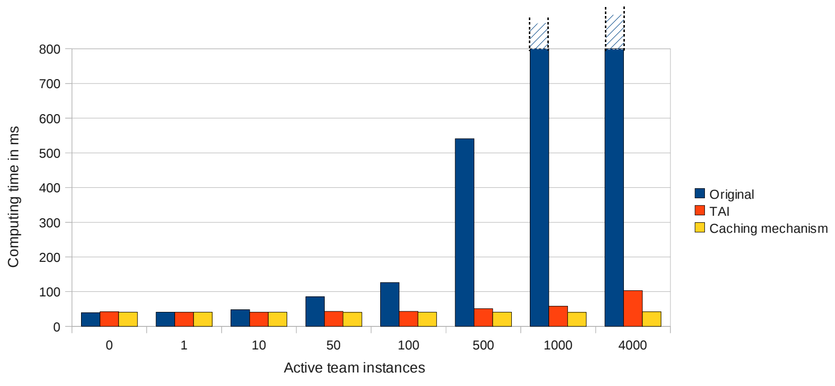
Figure 7.2: Benchmark Results for the Native Activation Infrastructure with one Bound Base Class per Teams

7.2.1 VM-internal Activation Infrastructure

We measured the benefit of implementing a VM-internal activation infrastructure (cf. Section 6.2.1) with the following benchmark. A team-level method is executed 3000 times, causing 3000 implicit team activations and subsequent deactivations. The corresponding team class contained one role class bound to a base class. Thus, activating an instance of this team led to registration with one base class. We varied the number of previously activated team instances from 0 to 4000. The results of this benchmark are illustrated in Figure 7.2.

The number of active team instances is shown along the x-axis of the diagram. The y-axis represents the consumed time in milliseconds and the table below the chart illustrates the exact execution time values for the different configurations. We compared the performance of the original JamVM (*Original*) with the optimization that moves the activation infrastructure to the VM (*TAI*) and with the version with the additional caching mechanism (*Caching mechanism*).

Our experimental results show that TAI yields a considerable performance gain compared to the original JamVM. The execution time is reduced by a factor of 1.54 for 50 active team instances, by a factor of 2 for 100 instances, and it further improves the more team instances are activated. Compared to that, the gain of the caching mechanism is not as impressive for this benchmark. This result was unexpected because with the cache, we eliminate the need of team instances being moved around in memory. However, it can be explained by the fact that array copying is implemented very efficiently inside the JamVM and leads to a notable overhead only for a very high number of entries. So, the benefit of the cache becomes more significant when a high number of team instances is active. While with the cache the execution time stays nearly constant, for 4000 active team instances, the caching mechanism outperforms TAI by a factor of 1.7.



Active team instances	0	1	10	50	100	500	1000	4000
Original	39	41	48	85	126	541	1153	4200
TAI	42	41	41	43	43	51	58	103
Caching mechanism	41	41	41	40	41	41	40	42

Figure 7.3: Benchmark Results for the Native Activation Infrastructure with three Bound Base Classes per Team

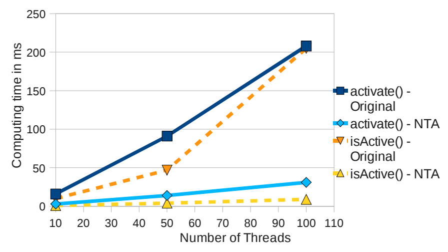
In addition, we repeated this benchmark with three base classes bound by the team class. The results are shown in Figure 7.3. Here, the improvement factor of TAI compared to the original version is even higher: e.g., almost 2 already for 50 active team instances. Note that compared to the activation cache we presented in [HSG10], the caching mechanism we developed in this work scales to an arbitrary number of adapted base classes per team.

In addition to the runtime improvement, we significantly reduced the byte-code size of adapted base classes. Now, the methods for adding and removing teams are omitted and only once implemented in the VM code. Including code blocks and constant pool entries, this implies a code reduction by about 800 bytes per adapted base class. Moreover, the wrapper code for every adapted base method could be reduced by about 90 bytes.

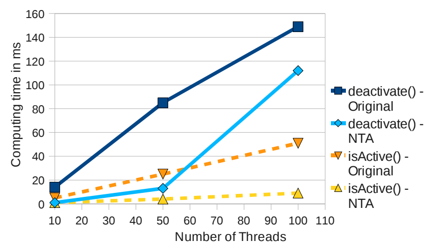
7.2.2 Native Activation Mechanism

To assess the effect of the native activation methods (cf. Section 6.2.2), we implemented a benchmark that creates a variable number of threads and that measures the computing time for activating or deactivating 1000 team instances for every single thread. As we are only interested in the execution time of the activation methods itself here, the corresponding team class does not define any aspect bindings. In addition, we also measured the time for the `isActive` method while all teams are activated or deactivated.

The left chart in Figure 7.4 shows the results of the native (*NTA*) and original (*Original*) implementation for an `activate` and for a subsequent `isActive` call, while the right chart shows the same for a `deactivate` call. The benchmark revealed a significant improvement of the execution time for all native activation methods. As can be seen, the computing time for team activation has been reduced by more than 80% (factor 5 to 7), and checking active teams



Number of Threads	10	50	100
activate() - Original	16	91	208
activate() - NTA	3	14	31
isActive() - Original	10	47	205
isActive() - NTA	1	4	9



Number of Threads	10	50	100
deactivate() - Original	14	85	149
deactivate() - NTA	1	13	112
isActive() - Original	5	25	51
isActive() - NTA	1	4	9

Figure 7.4: Benchmark Results for Native Team Activation

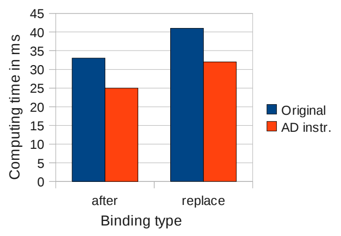
for activation costs at least 90% less time than before. Deactivating the teams for all threads is reduced by 93% for 10 threads and by 85% for 50 threads. For 100 threads, the improvement is only 25%. However, this is still a significant gain and even if the improvement decreases for even more threads, this is not very problematical, as only a few programs use such high numbers of threads. Finally, to call the `isActive` method for deactivated teams takes about 80% less computing time than before. Corresponding benchmarks for global and implicit team activation showed that their execution time is improved similarly.

Since this benchmark only measured the performance of isolated team activation methods, we must still clarify how strong the effect will be on real-world applications. But even if the `activate` and `deactivate` methods are rarely used, an improvement should be present in every program that uses AOP because the `isActive` method is executed once per activated team instance whenever a base method is called.

7.2.3 Bytecode Instructions for Aspect Dispatch

To measure the optimization gain of the JVM-internal aspect dispatch mechanism (cf. Section 6.3.1), we implemented the following benchmarks: The first one consists of a base class, holding a base method with empty body, and of a team class, adding an empty aspect method as callin to the base method. Thus, most of the base method's computing time is used to perform the newly implemented bytecode. Initially, 100 team instances are created and activated. After that, the base method is called 100 times and the computing time from the first to the last base method call is measured. We compare the execution of an after-callin with the execution of a replace-callin. The chart shown in Figure 7.5 compares the results from the original implementation (*Original* with the results from the aspect dispatch instructions (*AD instr.*). With our optimization, we saved 24% of computing time for the after-callin, and 22% for the replace-callin.

As we changed the handling of thread-local inactive team instances in this optimization, we also measure the execution time if the team instances are (partly) only activated for another than the current thread. The team and base



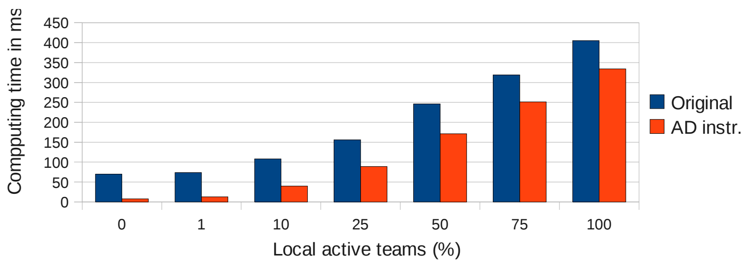
Binding type	after	replace
Original	33	41
AD instr.	25	32

Figure 7.5: Benchmark Results for the Aspect Dispatch Instructions

class are defined as above. Again, we define a single replace-callin. 100 team instances are activated for another thread. In addition, n=[0,1,10,25,50,75,100] team instances are activated for the current (main) thread. Thus, the values of n represent the percentage of registered team instances that are actually active for the thread executing the benchmark. This way, the base class' array of active teams is filled with all 100 teams, but when looking for active aspects, (100-n)% of the teams have to be skipped. The results of this benchmark are shown in Figure 7.6.

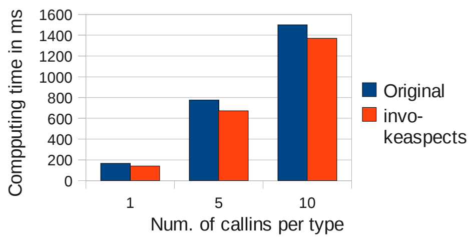
The improvement gets most significant if all team instances are only active for another thread (n=0). Here, we could reduce the execution time by 89%. For more thread-local active team instances, this high percentage gradually degrades to 18% if all teams are also locally active, which conforms to the results in Figure 7.5. The high performance gain, especially for less thread-local activated teams, is explained by the new handling of the array for active teams. In the original implementation, the array is traversed with a recursion over many Java method calls until the next thread-local active team is found. In contrast, the `nextaspect` instruction immediately returns the next team instance that is active for the current thread.

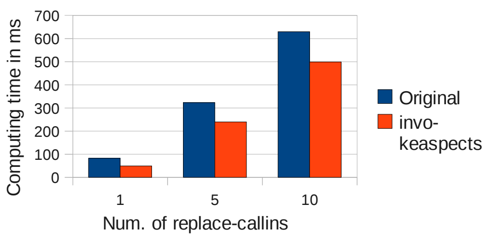
Note that the latter benchmark represents a kind of distributed fat, as described in Section 7.1.2 and further benchmarked in Section 7.2.7.



Local active teams (%)	0	1	10	25	50	75	100
Original	70	74	108	156	246	319	405
AD instr.	8	13	40	89	171	251	334

Figure 7.6: Benchmark results for local inactive teams





Callins per Type	1	5	10
Original	167	776	1500
invokeaspects	140	673	1370

replace-callins	1	5	10
Original	82	323	630
invokeaspects	49	240	499

Figure 7.7: Benchmark Results for the 'invokeaspects' Instruction

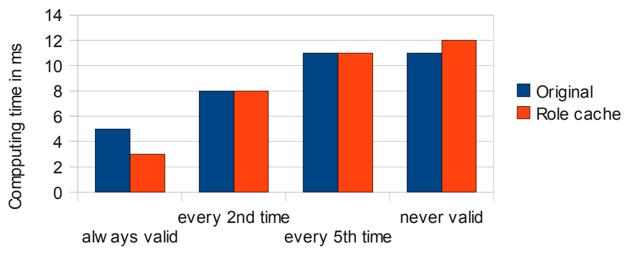
With this optimization we reduced the bytecode size of adapted base classes by about 290 bytes due to redundant constant pool entries. Additionally, the wrapper code for every adapted base method is reduced by about 100 bytes.

7.2.4 Callin Execution inside the JVM

To evaluate the performance benefit of the new `invokeaspects` instruction (cf. Section 6.3.2), we implemented a benchmark measuring the execution time of a base method that is adapted by different kinds and numbers of callins. For the first part of the benchmark, we bind the same number of callins of each type (before, replace, after) to a base method. The callins are all defined in a single role that adapts the class of the base method. For the benchmark, we create and activate 20 instances of the role's surrounding team class. The base method is called 1000 times. Again, the results are arithmetical means of 10 runs.

As shown by the left chart of Figure 7.7, we compare the execution time of the original JamVM with the version enhanced with the `invokeaspects` instruction. As this optimization depends on optimizations implemented before, they are also involved in this benchmark. Only the role cache, which is independent of the other optimizations, is not enabled here. The number of callins of each type takes the values 1, 5, and 10. We see that this optimization reduces the execution time by 16% for one callin of each type. For 5 callins each, the reduction is 13%, and for 10 callins it is 9%. The decreasing performance gain for more callins of each type can be explained as follows: The actual calls to the aspect methods are not subject to our optimizations and are thus identical in both versions. The more callins a team is declaring for a base method, the less optimization potential exists in-between. This in particular applies to before- or after-callins for the same team, as they are simply executed one after another.

In the second part of the benchmark, we only use replace-callins. The other parameters are identical to the first part. The results are illustrated in the right chart of Figure 7.7. Here, the reduction of the execution time is even higher: 40% for 1 callin, 26% for 5 callins, and 21% for 10 callins. The especially high



	always valid	every 2nd time	every 5th time	never valid
Original	5	8	11	11
Role cache	3	8	11	12

Figure 7.8: Benchmark Results for the Role Cache

performance gain for replace-callins is a result of the following facts: On the one hand, we replaced the expensive recursive calls of the base method wrappers in the original version by more efficient VM-internal (C-level) method calls. On the other hand, we simplified the passing of arguments during aspect and base method execution by storing them as part of the aspect iterator (cf. Sec. 6.3.2). Additional performance evaluations of the introduction of the `invokeaspects` instruction can be found in the diploma thesis of Daniel Stöhr [Stö10].

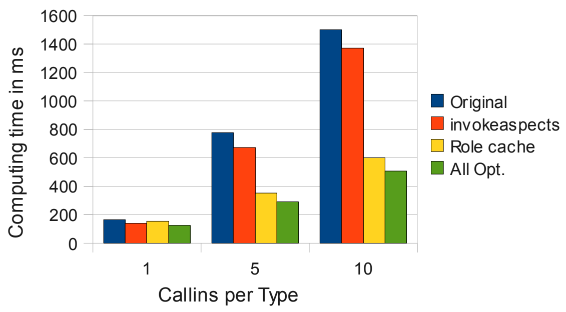
The code size of adapted base classes is reduced because the main part of the wrapper method code is now done by the VM-internal implementation of `invokeaspects`. The actual size reduction depends on the number of adapting teams and role methods. For a single base method that is adapted by one team class with one (after-) bound role method we save about 600 bytes (including constant pool entries).

7.2.5 Role Object Caching

Solely measuring the effects of the role cache (cf. Section 6.3.3) on the lifting mechanism resembles the results of the simulation described in Section 5.3.2. To evaluate the effect on real aspect execution runtime, we also benchmark the effect of the cached lifting during a callin execution. A base method is adapted by an after-callin. It is repetitively called 1000 times, causing a lifting of the corresponding base object to the involved role object each time. Here, the cache is always valid. One instance of the adapting team is activated. To provoke cache misses, another base method that is adapted by a role of another team class is called in between.

As shown in Figure 7.8, we get an improvement of 40% if the cache is *always valid*. If, the cache is valid *every second time* and *every fifth time*, respectively, the execution time does not change. In this cases, the benefit of the role cache and its overhead when missed seem to cancel each other. In the worst case, that is if the cache is *never valid*, the performance decreased by 9%.

In addition, we also repeated the benchmark illustrated in the left side of Figure 7.7 for the role cache optimization. The results shown in Figure 7.9,



Callins per Type	1	5	10
Original	167	776	1500
invokeaspects	140	673	1370
Role cache	154	351	601
All Opt.	126	293	507

Figure 7.9: Benchmark Results for Multiple Callins

furthermore comprise the execution with all our optimizations enabled. We observe that the role cache (*Role cache*) significantly reduces the execution time for multiple callins per team. For 5 callins of each type the improvement is 55% and for 10 callins it is 60%. This can be explained by the fact that here, many callins to the same role object are executed subsequently. Thus, the role cache frequently contains a valid role object. Finally, the results also show that we achieve an even higher performance improvement if we enable all our optimizations together. In Section 7.2.6 we discuss more benchmark results for the combined optimizations.

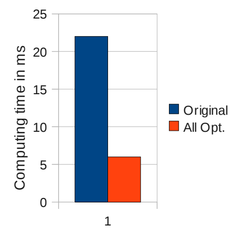
The influence of the role cache optimization on the execution time highly depends on the concrete design of the executed program. Thus, to avoid the overhead in case of frequent cache misses, we think it would be worthwhile to only include the role cache for base objects which are (statically) expected to be frequently lifted to the same role object.

In contrast to the preceding optimizations, with the role cache the code size is not reduced. Due to the additional cache access code, this optimization actually increased the bytecode size by about 56 bytes for every lifting method.

7.2.6 Combining all Optimizations

To evaluate the overall performance gain of our optimizations, we combined the optimizations presented in the previous sections. One result is already illustrated by Figure 7.9. With all optimizations (*All Opt.*) we could reduce the execution time by 25% for one callin per type and even by 66% for 10 callins each type. As stated in the description of this benchmark (see Sec. 7.2.5), it is especially beneficial for a profitable use of the role cache.

In addition, we constructed a benchmark comprising the *activation* and *execution* mechanisms considered by our approach. A team level method is



Original	22
All Opt.	6

Figure 7.10: Benchmark Results for Implicit Team Activation and Callin Execution

executed 1000 times, causing implicit activations. Furthermore, the method calls a base method, which is adapted by an after-callin of the afore activated team. Thus, also aspect execution including lifting is considered. The results, shown in Figure 7.10, are the arithmetic means of 10 benchmark runs. Executed by the original JamVM, the benchmark took 22 ms. With all optimizations enabled, we could reduce the execution time to 6 ms. This means that with our optimizations, the overhead for this benchmark could be reduced by 73%. Thus, it could be executed more than three times faster.

The effect of our combined optimizations on methods which are not subject to actual aspect execution is evaluated in the next section.

7.2.7 Inherent Overhead (Distributed Fat)

As announced in Section 7.1.2, we also analyze the inherent overhead of our approach for execution of unadapted methods. We start by classifying the possible cases of executing a base method `BC.bm()`, as illustrated by the flowchart in Figure 7.11. The conditional symbols gradually check the degree of adaption of the base class `BC` and the base method `bm()`, respectively. We differentiate between aspect (callin) binding (*Bind. for ...*) and aspect activation (*Act. for ...*). The end symbols of the flowchart represent the eight possible cases for executing a base method. Note that only in the bottommost case (C8) aspect functionality is actually executed. In the following, we shortly describe each of the possible cases (C1-C8).

[**C1**] Normal execution of method BC.bm(), no aspects involved

[**C2**] Aspect bound to other method BC.bm′, no activation

[**C3**] Aspect bound to other method BC.bm′, with activation

[**C4**] Aspect bound to the method BC.bm(), no activations

[**C5**] In-active aspect bound to BC.bm();
Other aspect bound to other method BC.bm′, activation for other thread

[**C6**] Aspect bound to BC.bm(), activation for other thread

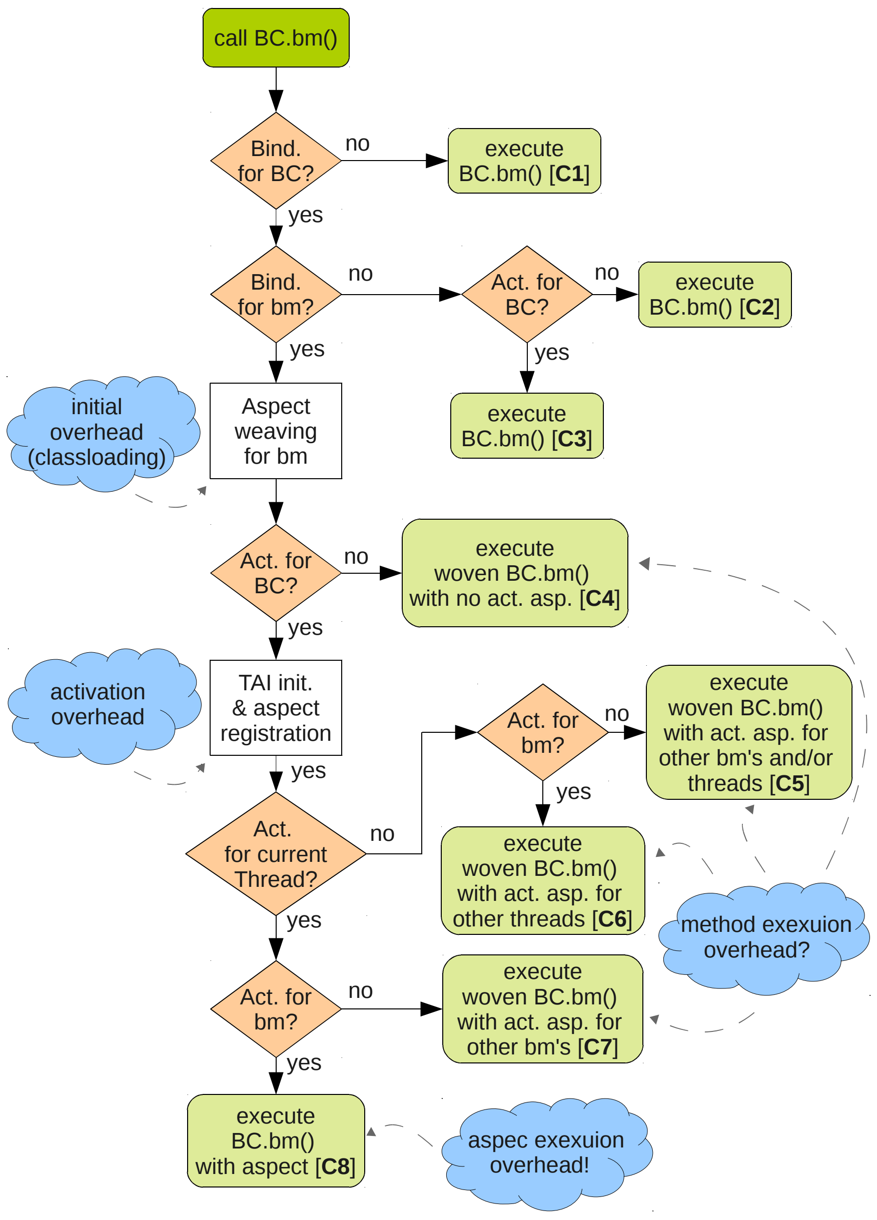
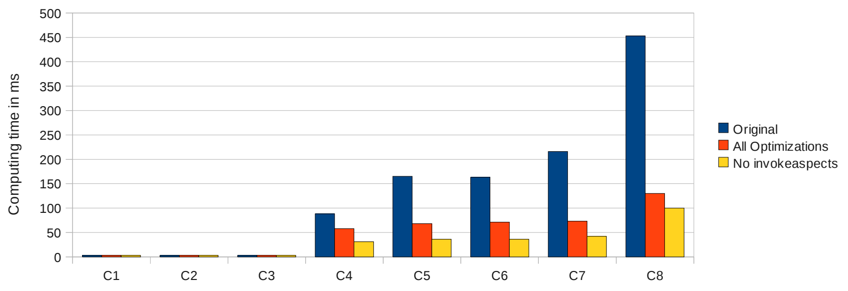

call BC.bm()
Bind. for BC?
no
execute BC.bm() [C1]
yes
Bind. for bm?
no
Act. for BC?
no
execute BC.bm() [C2]
yes
execute BC.bm() [C3]
yes
initial overhead (classloading)
Aspect weaving for bm
Act. for BC?
no
execute woven BC.bm() with no act. asp. [C4]
yes
activation overhead
TAI init. & aspect registration
yes
Act. for current Thread?
no
Act. for bm?
no
execute woven BC.bm() with act. asp. for other bm's and/or threads [C5]
yes
execute woven BC.bm() with act. asp. for other threads [C6]
method exexuion overhead?
yes
Act. for bm?
no
execute woven BC.bm() with act. asp. for other bm's [C7]
yes
execute BC.bm() with aspect [C8]
aspec exexuion overhead!

Figure 7.11: Distributed Fat for Executing a Base Method



	C1	C2	C3	C4	C5	C6	C7	C8
Original	3	3	3	88	165	163	216	453
All Optimizations	3	3	3	58	68	71	73	130
No `invokeaspects`	3	3	3	31	36	36	42	100

Figure 7.12: Benchmark Results for the Cases C1-C8

[**C7**] In-active aspect bound to BC.bm();
Other aspect bound to other method BC.bm′, activation for current thread

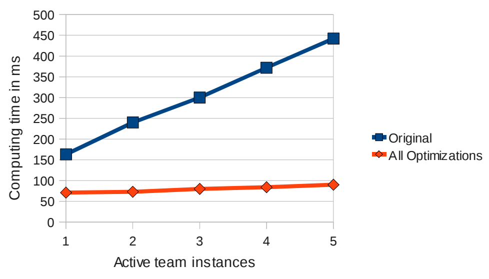
[**C8**] Aspect bound to BC.bm(), activation for current thread:
Actual aspect execution!

The cloud symbols in Figure 7.11 indicate where we expect some kind of overhead. As for the cases C1 to C3, the executed base method itself is not concerned by an aspect binding and thus not woven, we do not expect any overhead here. Due to the actual aspect execution, for C8 an overhead is essential. The results of the cases C4 to C7 are most interesting. Here, the base method has been woven but the activation state does not allow aspect execution.

We executed benchmarks for each of the cases C1 to C8. As before, the bodies of the involved methods are empty. In each case the base method is called 100,000 times. Figure 7.12 shows the results of our benchmarks. Here, only *one* team instance is activated in each case (if activation is involved at all). We compare the execution time of the original JamVM (*Original*) with a version with all our optimizations enabled (*All Optimizations*). In addition to this, we also include the results of measurements with the invokeapects optimization disabled (*No INVOKEASPECTS*).

In the following discussion of the results, multiple cases are summarized if appropriate. Note that the even better results without `invokeaspects` are subsequently discussed separately.

C1-C3: In these cases, no aspects are bound to `BC.bm()`, thus the method is not woven. As expected, the benchmark shows a zero-overhead. The results are the same for an arbitrary number of active team instances.



Active team instances	1	2	3	4	5
Original	163	240	300	372	442
All Optimizations	71	73	80	84	90

Figure 7.13: Varying the Number of Teams for C6

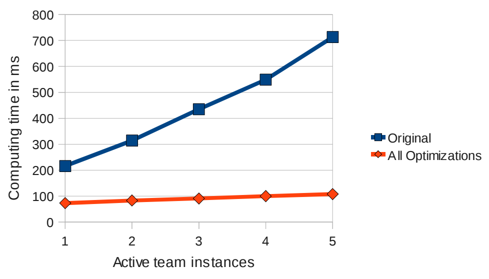
C4: Here, we have no active aspects but the base method has been woven. This causes a significant overhead. With our optimizations (*All Optimizations*), we could reduce this overhead by a factor of 1.52 (34%). Again, the results do not change if more team instances are activated.

C5 & C6: These two cases are similar, as with both the registered aspects are only activated for another than the current thread. Here, the original overhead is nearly twice as high as for C4. This can be explained by the fact that the activated aspects have to be considered, even if they are only active for another thread. With our optimizations, we could reduce this overhead to less than 50% (by factor 2.43 for C5; by factor 2.3 for C6). In this case, the benefit of our optimizations is even higher if more team instances are active. Figure 7.13 shows this effect for C6.

C7: This case shows an even higher overhead than C5 and C6. Here, the activated aspects are processed and none can be excluded because they are activated for another thread. We could reduce the overhead by nearly two thirds (factor 2.96).

Nonetheless, these results indicate that it would be reasonable to use a more sophisticated aspect registration strategy. This would facilitate to only process aspects that are relevant for the currently executed method. Again, our optimizations are even more beneficial for more active aspects, as shown in Figure 7.14.

C8: Actually, the case C8 does not measure distributed fat but real aspect execution, which is already covered by the benchmarks in Section 7.2.4 and 7.2.6. For multiple activated team instances, enabling the role cache slightly degrades the performance, as the cache is always invalid because of the individual callins from different team instances. For a single active team instance, the role cache is an advantage because the base object is always lifted to the same role instance of the only existing team instance.



Active team instances	1	2	3	4	5
Original	216	314	435	549	713
All Optimizations	73	83	91	100	108

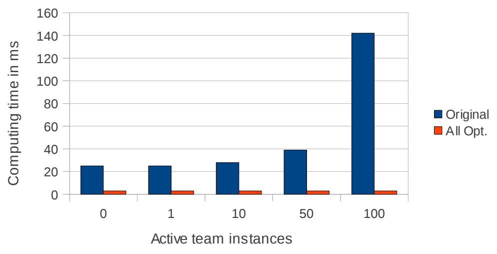
Figure 7.14: Varying the Number of Teams for C7

Overhead of invokeaspects As shown by Figure 7.12, for C4 to C8 the results for a single active team instance are even better if the `invokeaspects` optimization is disabled. This means that this optimization increases the distributed fat and even for C8 the results are better. This is due to the fact that the realization of `invokeaspects` includes a lot of initializations (e.g., for the aspect iterator) which are overkill if no (C4) or only one team instance is active. For C5 and C6 for 100 or more active team instances the results for `invokeaspects` are no longer worse and for C7 the optimization is profitable for more than 20 active team instances. However, for C8, the result is better already for two active team instances. For actual aspect execution the advantage of the `invokeaspects` optimization is also demonstrated by the benchmark in Section 7.2.4.

These results indicate that it might be reasonable to disable some of the optimizations under certain circumstances. If an application frequently executes methods that are adapted by locally deactivated team instances, the bytecode instruction `invokeaspects` could be disabled. The same applies to the role cache, which could be disabled if cache misses are commonly occurring. However, the overall effect of our optimizations on the distributed fat is very promising.

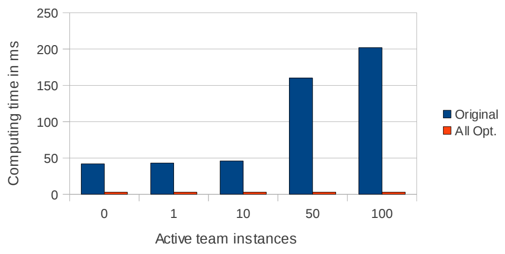
7.2.8 Benchmarks on the ARM Processor

To demonstrate the success of our optimizations for real embedded mobile devices, we validated our results on Openmoko's ARM-based open source smart phone FreeRunner (see Sec. 7.1.3). To this end, we repeated the benchmarks that we presented in Sections 7.2.1 – 7.2.7. In this section, we outline the relevant results for the aspect activation and the aspect execution. We compare the execution time of the original JamVM (*Original*) with a version with all our optimizations enabled (*allOpt*). If notable, we also present the results for other combinations of our optimizations. As the details of the benchmarks are already discussed in previous sections, we only refer to them and discuss the results of executing them on the ARM.



Active team instances	0	1	10	50	100
Original	25	25	28	39	142
All Opt.	3	3	3	3	3

(a) One adapted base class



Active team instances	0	1	10	50	100
Original	42	43	46	160	202
All Opt.	3	3	3	3	3

(b) Three adapted base classes

Figure 7.15: Benchmark Results for Implicit Team Activation

Due to the restricted hardware, the benchmarks are generally executed notably slower on the ARM. On that account, we reduced the number of call iterations for the benchmarks of this section. For some benchmarks, we further reduced the number of active team instances or the number of existing threads. The actual numbers are specified together with the particular benchmark results.

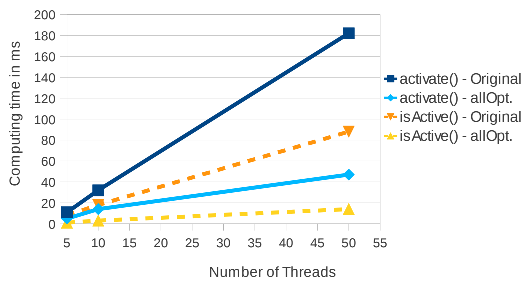
Aspect Activation

To evaluate the optimized aspect activation on the ARM, we executed the benchmark that was presented in Section 7.2.1 (cf. Fig. 7.2 and 7.3). This benchmark measures the execution time of implicit team activation for different numbers of active teams. Note that we reduced the number of calls to the team method from 3000 to 100 and that at most 100 team instances are activated. The corresponding results for the ARM are shown in Figure 7.15. They show that also on the ARM, the overhead of the implicit team activation could be significantly reduced. It is now constant and small (3ms), independent of the number of previously activated team instances.

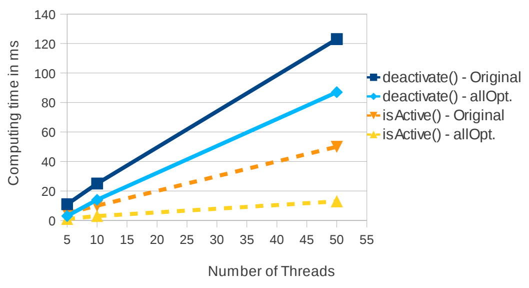
To measure the effect of our optimizations on multi-threaded team activation, we used the benchmark presented in Section 7.2.2 (cf. Fig. 7.4). This time, we reduced the number of call iterations from 1000 to 100. Furthermore, we executed the benchmark with 5, 10, and 50 threads respectively. The results for the ARM are illustrated in Figure 7.16. With our optimizations, reduced the execution time of the `activate` method by 55–74%. Calling the `isActive` method costs 74–87 % less time, and `deactivate` consumes 29–73% less time. For `deactivate`, the improvement decreases if more threads are involved.

Aspect Execution

To evaluate the optimized aspect execution on the ARM, we repeated three benchmarks from the preceding sections. All of them repeatedly execute a



Number of Threads	5	10	50
activate() - Original	11	32	182
activate() - allOpt.	5	14	47
isActive() - Original	8	18	88
isActive() - allOpt.	1	3	14



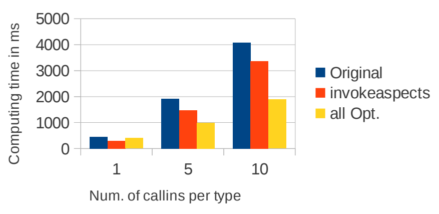
Number of Threads	5	10	50
deactivate() - Original	11	25	123
deactivate() - allOpt.	3	14	87
isActive() - Original	5	10	50
isActive() - allOpt.	1	3	13

Figure 7.16: Benchmark Results for Multi-threaded Activation

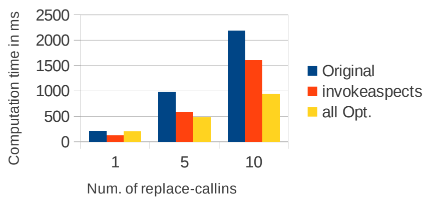
base method that is adapted by aspect functionality in different scenarios. Note that compared to the corresponding benchmarks on the desktop PC, we changed the number of call iterations from 1000 to 100.

In the first benchmark, the base method is executed with different numbers of adapting callins. We described this benchmark in Section 7.2.4 (cf. Fig. 7.9). The results for the execution on the ARM are shown in Figure 7.17. On the left side of the figure, we see the results for callins of all types (after, before, and replace). For one callin of each type, the execution time reduction of *allOpt* is 9%, for 5 callins each, it is 48%, and for 10 callins each, it is 54%. The results also show that for 1 callin of each type, the result of the *invokeaspects* version (without the role cache) is even better, namely already 35%. In this case, the advantage of the role cache seems to be not yet big enough to redeem the overhead of its introduction. However, in the other two cases, *allOpt* has the best results. The results for solely replace-callins are comparable (6–57%). Again, *allOpt* is outperformed by *invokeaspects* for one replace-callin.

For the second benchmark on aspect execution, we varied the percentage of team instances that are not only active for another, but also for he current thread. This benchmark was described in Section 7.2.3 (cf. Fig. 7.6). The results for the ARM are illustrated in Figure 7.18. As on the desktop PC, the

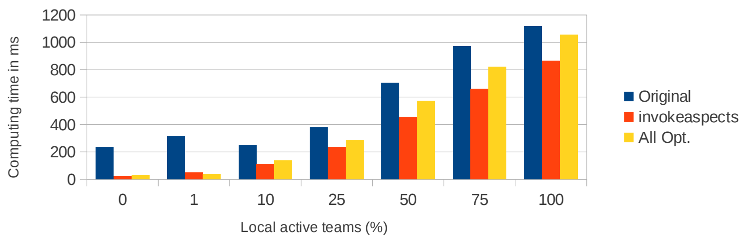


Callins per Type	1	5	10
Original	446	1926	4078
invokeaspects	290	1474	3370
all Opt.	404	994	1894



replace-callins	1	5	10
Original	213	979	2184
invokeaspects	127	588	1604
all Opt.	201	484	937

Figure 7.17: Benchmark Results for Multiple Callins



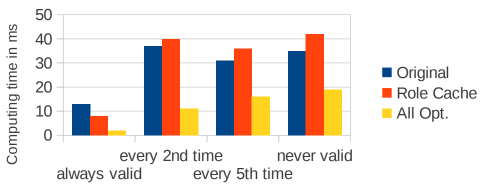
Local active teams (%)	0	1	10	25	50	75	100
Original	237	316	251	379	706	969	1115
invokeaspects	24	49	113	234	454	660	866
All Opt.	33	40	137	287	574	819	1055

Figure 7.18: Benchmark Results for Partly Deactivated Teams

improvement is very high for few local active team instances (about 86%) and gets less if the percentage of local active team instances converges to 100%. For this benchmark, the improvement of the *invokeaspects* version is generally higher than those of *allOpt*. The difference gets more significant for higher numbers of local active teams instances. This contradicts the results we got on the desktop PC.

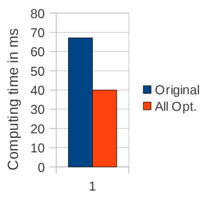
For the third benchmark on aspect execution, we varied the number of role cache hits. This benchmark was described in Section 7.2.5 (cf. Fig. 7.8). The ARM results are shown in Figure 7.19. Solely using the *Role Cache* is only profitable if the cache is always valid. However, when using *allOpt*, the performance is considerably better. Even for permanent cache misses (*never valid*), the drawback of cache misses is compensated by the optimizations that improve the other parts of the aspect execution.

Moreover, we repeated the benchmark that we used to measure the performance improvement for our combined optimizations in Section 7.2.6 (cf. Fig. 7.10). This benchmark measures the execution time for implicit team activation in combination with the execution of an adapted base method. The results for executing this benchmark on the ARM are illustrated in Figure 7.20. The reduction is not as high as for the desktop PC (73%) but it is still considerable, namely 40%.



	always valid	every 2nd time	every 5th time	never valid
Original	13	37	31	35
Role Cache	8	40	36	42
All Opt.	2	11	16	19

Figure 7.19: Benchmark Results for Lifting



Original	67
All Opt.	40

Figure 7.20: Benchmark Results for Implicit Team Activation and Callin Execution

Finally, we validated that our optimizations do not negatively affect the execution of regular (unadapted) Java methods.

7.2.9 Performance Evaluation

In this work, we significantly improved the *aspect activation* of OT/J by developing a VM-internal activation infrastructure as well as a native team activation mechanism. The VM-internal activation infrastructure significantly reduces the runtime overhead of team activation, especially for larger numbers of active teams. Thus, the implicit activation of teams is now fast and its effort is nearly constant, independent from the number of previously activated teams. Based on this adaptation, we achieved a high performance gain by implementing a native team activation mechanism. Of particular importance is the 80 to 90% faster team activation check because this functionality is also involved in aspect execution.

We further optimized the *aspect execution* of OT/J by additional bytecode instructions and a role cache. Our newly introduced bytecode instructions provide an optimized usage of the VM-internal activation infrastructure. This optimization guarantees an efficient team lookup, especially for thread-local team activation. Finally, by the introduction of the `invokeaspect` instruction that is responsible for the actual aspect execution we could reduce the execution time of adapted base methods by up to 16%. By the introduction of a role cache, we could improve the pure lifting mechanism by up to 93%. This leads to an improvement of the general aspect execution by up to 40%.

Eventually, the combination of our optimizations shows an even higher improvement of the execution performance. For example, on the Intel PC, the execution time of adapted base methods is 66% less for 10 callins of each type from one team instance (see Fig. 7.9). On the ARM processor, execution time for the same benchmark is reduced by 54% (see Fig. 7.17). Furthermore, for a combined execution of (implicit) team activation and aspect execution the improvement is 73% on the Intel PC (see Fig. 7.10) and 40% on the ARM (see Fig. 7.20).

The effect on the distributed fat (see Sec.7.2.7) is also very promising. For completely unbound base classes, no overhead is imposed at all. For the other cases, where a base method is bound but no aspect functionality has to be executed due to the activation state, we could reduce the distributed fat to by up to one third.

The results of our micro benchmarks demonstrate that our optimizations are very successful on the Intel PC as well as on the ARM processor. In addition to the micro benchmarks, we also want to estimate the benefit for real-world application. For this reason, we performed additional experiments on a game application. We discuss the results of these experiments in Section 7.3.2.

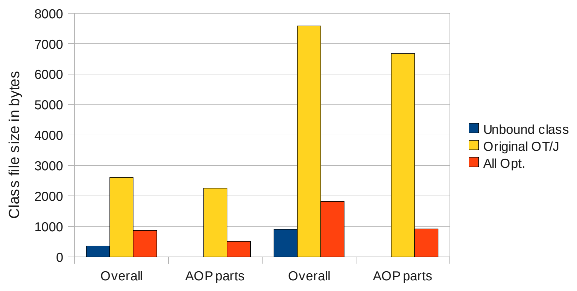
7.2.10 Code Size Effects

In the preceding sections, we briefly referred to the influence of our optimizations on the code size. Here, we investigate the overall effect of the combined optimizations.

As base classes are the main subject of aspect weaving, the major part of changes affect the code size of adapted base classes. The methods for registering (and removing) team instances with base classes are no longer part of the base class. Moreover, the wrapper code of adapted base classes could be significantly reduced. Actually, the only thing the remaining wrapper is doing is to invoke the `invokeaspects` instruction. The actual size reduction of this adaptations depends on the number of adapting teams and role methods. In contrast, the size of team classes is slightly increased by the additional code for role cache access inside the lifting methods.

We measured the size of woven base class files for two examples. In the first one, the only method of a base class is adapted by an after-callin of a single team class. In the second example, the base class has 10 methods which are each adapted by a different role method of the same team. The results of this measurement are shown in Figure 7.21. For a single adapted base method, we could reduce the overhead of the AOP parts by 77% and for 10 adapted base methods it could be reduced by 86%. Note that this measurement only takes into account the size of the class files and neglects the additional runtime data we introduced by our optimizations.

To finally evaluate the code size effect, also the runtime memory footprint has to be examined. An important fact is the effect for parts of the application that are not subject to AOP mechanism. By our optimizations, we enhanced different runtime data structures to hold additional aspect-specific information. Although we payed attention to only initialize this data if actually needed, a minimal overhead is generally imposed by the additional pointers in the corresponding structures. Table 7.1 summarizes this initial overhead for the affected runtime data structures.



Number of base methods	1		10	
	Overall	AOP parts	Overall	AOP parts
Unbound class	355	0	906	0
Original OT/J	2606	2251	7583	6677
All Opt.	864	509	1822	916
Size reduction in %	67	**77**	76	**86**

Figure 7.21: Code Size Measurement for adapted Base Class Files

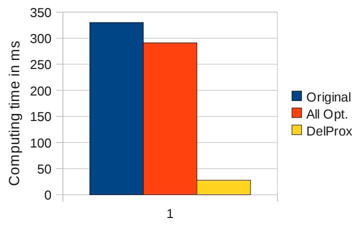
In a first experiment, we used the Java API of `java.lang.Runtime` to measure the additional memory consumption for loading a base class, creating an instance, and executing methods on it. We used the example of a base class with 10 base methods from above. First, we examined the used memory if no aspect adaption is used. As indicated by Table 7.1, the amount of memory increases by 60 bytes (from 284 to to 344 bytes). This overhead is composed of 12 bytes for the base class plus 8 bytes for the base object plus 10*4=40 bytes for the 10 base methods.

When optimizing code, there is often a trade-off between code size and execution speed. For example, we could avoid the initial size overhead for unadapted code by introducing VM-global data structures (e.g., hash maps) which store the additional data needed by our optimizations. This way, the memory usage would not be increased for every object, but the lookup of this data would be less efficient. As our main focus in this work is the reduction of execution time, we decided to prioritize faster lookup mechanisms.

We also examined the memory use for the same base class when each of its base methods is adapted by a callin, as described above. This time, the used memory decreases from 8992 to 7248 bytes. This indicates that our optimizations have a positive effect on the runtime memory use of adapted base classes. However, more detailed experiments will have to be carried out, to evaluate the overall memory effect of our approach.

Table 7.1: Initial Code Size Overhead for Runtime Data Structures

Data Structure	**Overhead (in bytes)**
Class	12
Object	8
Method	4



Original	330
All Opt.	291
DelProx	28

Figure 7.22: Callin Execution Time

Another factor for the code size is the size of the JVM binary. By our additions, this size increased from 788 K to 892 K (by 13.2%). However, this overhead is acceptable as the JVM is installed only once at each system.

7.2.11 Delegation Proxy Approach

To measure how the delegation proxy approach (cf. Section 6.4), affects the *execution time*, we used a micro benchmark that we already presented in [Bis10, p.103]. This benchmark measures the execution time of a base method with 10 before-callins from 10 different team instances. Figure 7.22 shows that we significantly reduced the overhead of executing adapted base methods with the delegation proxy approach. The execution time is reduced by 92%. Note that with all our optimizations from the approach described above, the improvement is only 12%. This result shows that, at least for this benchmark, the delegation proxy approach is extremely performant. Thus, it is worthwhile to integrate both of the approaches. Unfortunately, we could not repeat the benchmarks from Sections 7.2.1 – 7.2.7 for the delegation proxy approach because the implementation prototype does not support all of the used OT features.

The *memory usage* of base classes and team classes is also affected by the delegation proxy approach. On the one hand, the code size has been reduced. Looking at the classes in Figure 4.8, we can see that there are no more wrapper methods, neither in team classes nor in base classes. Moreover, the role specific lift methods in teams are no longer needed. On the other hand, the run-time memory usage is slightly increased by introducing additional VM-internal data. The delegation proxy adds 24 bytes to the code block of every executed method. Besides, additional fields have been added to the stack frame structure. The overall effect for classes with high adaptation by aspects, as well as for ones which are adapted less or not at all, will have to be weighed up against each other in future work.

Table 7.2: Mobile Device Application Types

Application Type	**Examples**
Communications	E-mail / IM clients
Games	Puzzle, Sports
Multimedia	Music / Video Players
Productivity	Calendars, Notepads
Travel	City Guides, GPS / Maps
Utilities	Address Book, Profile Manager, Browser

7.3 Case Study

In this section, we briefly discuss what kind of mobile device application scenarios generally benefit from the use of (dynamic) AOP. Then, we present our case study, a game application together with experimental results of a benchmark that measures the execution time of updating the game's view.

7.3.1 (Dynamic) AOP-Scenarios for Mobile Devices

AOP can be used to adapt mobile applications *statically* or *dynamically*. Statical adaptations can be used to encapsulate device specific parts of an application in aspect modules. Target devices can, for example, vary in the *screen size* or *resolution*, the *sound* support, or the available *control keys*, as illustrated by the feature diagram in Figure 7.23. Before an application is deployed to a specific device, the application can be configured in a product-line-like manner [HMPS07] [AMC$^+$07] to incorporate the matching aspects. This can be done manually or based upon information about the device configuration. Thus, the same core application can be deployed with different kinds of user interfaces, for example. Commonly used mobile applications can be classified according to [Ass08] by the types specified in Table 7.2.

Typical dynamic adaptation scenarios for mobile applications are often related to the changing context (e.g., the location). If they are anticipated, they

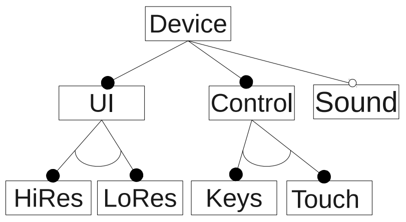


Figure 7.23: Device-specific, *statical* Adaptations

can be pre-installed by the user and triggered by the environment. Unanticipated dynamic adaptations can be proposed and installed by the environment. In the following scenarios, the change of contexts involves a modified application behavior. This can be realized by implementing the context-sensitive parts as aspects and dynamically deploying them when required by the context.

Sight Seeing In [Sch07], a sight seeing application is used to demonstrate context sensitive integration of GPS (Global Positioning System) coordinates. The base application features a list of sights together with a brief description for each sight. This application is then extended by context-sensitive features. A GPS sensor is integrated and used to enable the sorting of the sights according to their distance from the current user position.

In-Building Navigation Several dynamic extensions to common outdoor navigation applications can be considered. The following examples use building-specific point-of-interest navigation to assist users while inside these buildings.

- Shops: extra information like prices and location of specified products, routing to items [FN07, ACK94]
- Museum Guide: in-building map and explanations [RTA05]
- University/School: class schedule, room allocation schedule

Context-sensitive Games Mogi [LI09] is a virtual collecting game for mobile devices. To collect items, the player has to move around in a real city. The application has a map which shows items as well as other players. It is possible to trade with other players. GPS information is used to track the current position and to ensure that only items in a specified radius are visible.

On-demand Appointment Optimization In this scenario [RSC07], appointments can be rescheduled when the participants are unsuspectingly near to each other. This can be the case if, for example, both are visiting the same exhibition and both have fitting earlier time slots in their calendars.

In the future, more and more context-sensitive applications will be used on mobile devices. In particular, dynamic loading of location-related modules has to be expected.

7.3.2 OTPong - Scenario for this work

The presented scenarios show that context-sensitive adaptation is highly relevant in the context of mobile applications. To illustrate the benefits of our work, we selected a game scenario. After describing the example application, we evaluate our approach by benchmarking it. Finally, we present dynamic

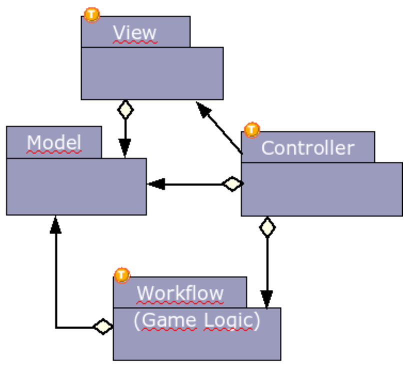


Figure 7.24: Overview of the OTPong Architecture

adaptation scenarios for the game that can be realized with dynamic AOP support.

Our example application *OTPong* is a variant of the classical computer game Pong [Pon] that we implemented in ObjectTeams as an exercise of an AOP class. The architecture consists of four modules, as illustrated by Figure 7.24. The *Model*, the *View*, the *Controller* and the *Workflow (Game Logic)* are each implemented in separate modules. While the model is implemented in pure Java, the other components make use of AOP features to interact with the rest of the application. Thus, for example, the *view* can easily be exchanged to fit the screen size and resolution of the target device. Such static adaptations can generally be used to configure the application according to the configuration of the target device before installation.

Benchmarking OTPong

To evaluate the effect of our optimizations on the OTPong application, we instrumented the code to measure the execution time of updating the view. This has to be done whenever game objects are moved, and thus happens frequently during the whole game time. In the workflow component (see Fig. 7.24), the class `Game` controls the overall game logic. As illustrated by Figure 7.25, the method `play()` executes a loop as long as the game is still in progress (`isPlaying()`). The game logic consists of the following steps: First, all game objects (e.g., paddles and balls) are moved by changing their geometry. Note that this only concerns the model objects. In the second step, the view is updated. Finally, in step three, potential collisions are detected and handled (e.g., by bouncing off the wall).

As indicated by the label *BM* in Figure 7.25, our benchmark measures the execution time of the second step. This part is especially relevant to us because the update of the view is realized by AOP mechanisms. The simplified control flow is also shown in the figure. The game logic loop calls the method `Board.update()`. This method is bound by an after-callin to the `repaint()` method of the `BoardView`. It triggers the invocation of the `paint()` methods

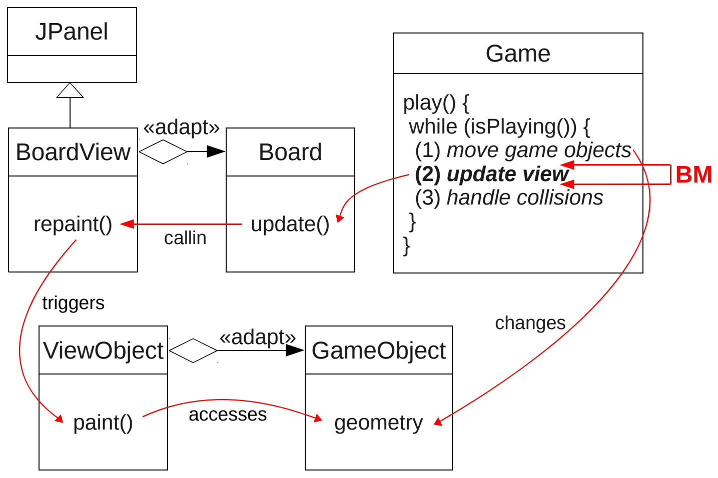


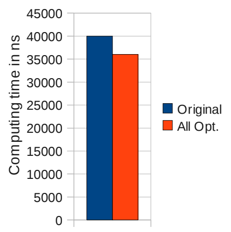
Figure 7.25: Updating the View of OTPong

of the view objects that are used to display the game objects. The view objects are modeled as role objects that are bound to the corresponding game objects. In order to update its position, each view objects accesses the geometry of its base game object.

Again, we executed this benchmark both on the desktop PC and on the FreeRunner smart phone, whose specifications are described in Section 7.1.3. Figure 7.26 shows the results of benchmarking the view update process in nanoseconds. Figure 7.26a shows that on the desktop PC our optimizations have improved the updating of the view by 10%. In Figure 7.26b, we see that on the FreeRunner the improvement is similar, namely 11%. As updating the view constitutes an important part of the OTPong application, this improvement is highly relevant for the overall performance of the application.

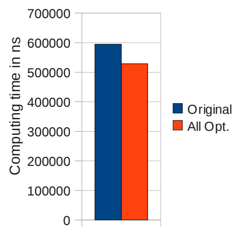
Extension Scenarios for OTPong

Possible additions to the basic game are *monsters*, a *shooting ability*, or *extra items* which can be collected with the ball. These extra items can, for example,



Original	40000
All Opt.	36000

(a) On Desktop PC



Original	594953
All Opt.	528617

(b) On FreeRunner

Figure 7.26: Benchmark Results for Updating the View in OTPong

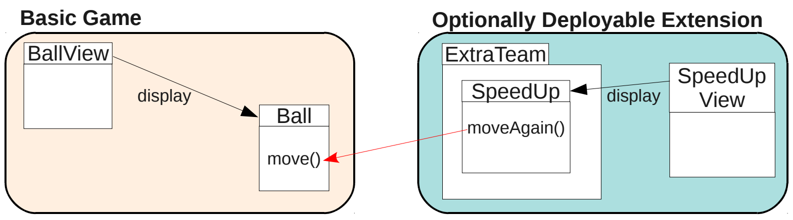


Figure 7.27: OTPong Extra Addition

increase or decrease the *size* of the paddles or change the *speed* of the ball. The *extra addition* has been implemented entirely encapsulated in aspect modules, which can optionally be deployed to the basic game. Figure 7.27 shows an `ExtraTeam` aspect which adapts the `move()` method of the `Ball` class. The corresponding aspect method `moveAgain` causes the ball to instantly move again, resulting in a faster movement. Such optional additions could also be developed and bought at a later date.

If the game had also a multi-player modus which allows playing against other, nearby people, we can imagine the following *dynamic* adaptation scenario. Different players can own different game extensions. If, for example, one player uses extra items, but the other one does not own these extensions, this could be a problem when playing together. Using dynamic AOP, the extension can be temporarily deployed also on the other device, as illustrated by Figure 7.28. It is, however, only available during the multi-player game and will not be permanently installed until the extension is actually bought.

The described extensions of the OTPong game can easily be realized with the AOP mechanisms of ObjectTeams. In fact, we already implemented the addition of extra items described above. To support real dynamic adaptation with additional features for this complex application, our prototypical implementation has to be completed in some points. However, the experimental setup we presented in Section 6.4 shows that our approach supports the dy-

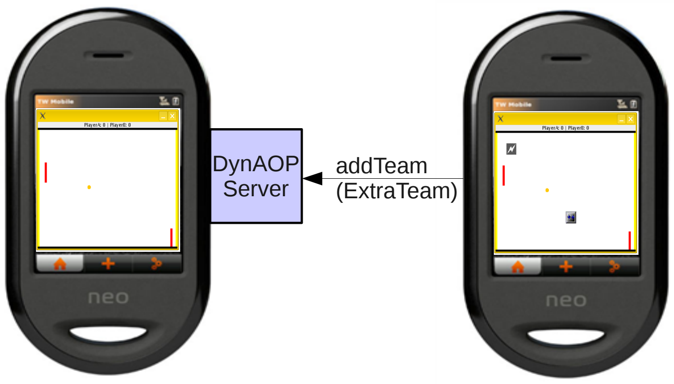


Figure 7.28: OTPong Dynamic Aspect Deployment

namical addition, activation and deactivation of new (unknown) aspects at runtime.

7.4 Transferability to other Approaches

In this thesis, we implemented our optimizations by extending the Java virtual machine JamVM for the aspect-oriented mechanisms of the programming language ObjectTeams. In this section, we investigate the transferability of our approach to other VM implementations on the one hand, and to other AOP languages on the other hand.

7.4.1 Other VM Implementations

By integrating our optimizations into the virtual machine, we mainly adapted the following components:

1. The **native interface**: to realize the native API for the VM-internal aspect activation
2. VM-internal **data structures** of classes, objects, methods and stack frames: to store additional AOP-specific data
3. The **interpreter**: to integrate the aspect-specific bytecode instructions (`getaspects`, `nextaspect`, `invokeaspects`)
4. The **class loading** mechanism: to lookup aspect specific binding information from bytecode attributes

As all of these are integral components of Java virtual machines, we expect that it is possible to transfer our approach to most of the existing JVM implementations. For more evidence, we investigated the implementation of Android's Dalvik VM (DVM) [Goob], which we introduced in Section 2.2.4. As expected, we could identify all the components affected by our optimizations. For example, the realization of the native interface for VM-internally implemented methods can be easily extended to support additional functionality. Furthermore, the relevant VM-internal data structures seem to be realized similarly to those in the JamVM. However, in the handling of bytecode attributes the DVM differs from the JamVM. The JamVM reads out and internally stores the bytecode attributes when a class is defined during class loading. By contrast, in the DVM, the attributes are kept in the *Dalvik executable* (dex) files and accessed there, when required. The AOP-specific attributes could be maintained in the same way. Note that it might be useful to adapt the dex-file format so that these additional attributes can be stored explicitly and more efficiently.

We assume that the main challenge of transferring our approach to the DVM is the integration of additional bytecode instructions in the interpreter. The DVM involves three different interpreters: a portable interpreter written in C, a fast interpreter with hand-coded assembler fragments for each architecture, and an interpreter with debugging support. All three interpreters have to be extended in order to integrate new instructions. Finally, also the generation of the dex-files needs to support them. However, we expect that altogether, all of our optimizations can be transfered to the DVM with comparatively small effort.

7.4.2 Other Programming Languages

In this section, we briefly investigate the applicability of our approach to other programming languages. Therefore, we specify the language concepts that are affected by our optimizations in the following list:

- VM-internal activation infrastructure: dynamic aspect activation at class-level, implicit activation
- Native activation mechanism: dynamic aspect activation (global and thread-local)
- Aspect dispatch instructions: dispatch/lookup of active adapting aspect instances
- Aspect execution instructions: method-level aspect execution, considers dynamic activation
- Role object caching: role-based concepts
- Delegation proxy: method-level aspect execution; dynamic aspect deployment; aspect activation

A programming language that provides one or more of the listed concepts can potentially benefit from the corresponding optimization. Thus, programming languages that support the dynamic activation of partial program definitions, like CaesarJ [Cae] and ContextJ [CHM06] could benefit from our optimizations for aspect activation.

8 Conclusion & Future Work

8.1 Results

Advanced modularization mechanisms like aspect-oriented programming are qualified to meet the increasing demands on applications for embedded mobile devices, such as *reusability*, *adaptability*, *extensibility*, and *portability*. However, without adequate optimizations their practicality to small devices with limited resources is restricted by overheads in execution time and code size. In this work, we presented optimizations for common AOP-mechanisms on the level of the virtual machine aiming for an applicability of AOP for embedded mobile devices. The key idea is a deep integration of AOP mechanisms into the virtual machine.

Our main contribution is a significant reduction of the overhead of high-level AOP constructs, which is also demonstrated by the results of our experiments. The success of the optimizations provides evidence that advanced high-level abstraction techniques like AOP can be efficiently used in embedded mobile devices. Furthermore, our work shows that efficient dynamic aspect deployment can be supported on the level of the JVM.

Overview Our overall approach can be summarized as follows: We started by investigating the overhead that is typically generated by the realization of AOP mechanisms. After that, we presented various optimizations to overcome this overhead for the aspect-oriented programming language ObjectTeams. We implemented these optimizations by extending the JamVM Java virtual machine. In doing so, we shifted mechanisms like the registration of activated aspects to the level of the JVM. Furthermore, we optimized the execution of AOP mechanisms, e.g., by introducing a caching mechanism for implicit activation and by efficient hash maps for thread-local team activation. Moreover, we analyzed AOP-specific semantic code properties in order to develop optimizations that utilize these AOP-specific semantic information. To this end, we developed a cache for role objects. In addition to the AOP optimizations, we presented the delegation proxy approach. With that, we have realized a deeper integration of the aspect execution mechanism into the virtual machine, and we substantially enhanced the dynamic capabilities of ObjectTeams by bringing in the capability of dynamically adding and activating new aspects at runtime.

Detailed Contributions As foundation of our approach, we identified the sources of overhead that are caused by the implementation of aspect activation and aspect execution at the level of the base programming language, namely Java. The main sources of overhead are the realization of aspect activation mechanisms at the level of Java bytecode of the adapted classes, the additional dynamic dispatch for aspect execution, and the missing native support of advanced modularization mechanisms, like the implicit inheritance of roles to subclasses of teams. To overcome this overhead, we developed optimizations for common AOP mechanisms by extending the data structures and the execution mechanisms of the JVM. In the following, we review the main achievements of our optimizations.

To optimize aspect activation, we introduced a VM-internal team activation infrastructure that also maintains the correct inheritance of active aspects to subclasses of adapted base classes. By our caching mechanism for subsequently activated and deactivated team instances, we further reduced the overhead of implicit team activation. In addition, we proposed a native team activation mechanism that enables the efficient activation and deactivation of aspects by avoiding the use of inefficient Java constructs. Moreover, we improved the aspect execution by the introduction of specific bytecode instructions for fast aspect dispatch and efficient execution of aspect functionality. Hereby, we were able to make the originally used method wrapper of base methods obsolete.

Furthermore, to support dynamic AOP for mobile device applications, we introduced a dynamically deployable method header (delegation proxy) that is responsible for the complete aspect dispatch and establishes the foundation for dynamic AOP. To evaluate the delegation proxy's capability for dynamically adding (and removing) new aspects at runtime, we implemented a prototypical dynamic AOP component. By this successful experiment, we showed that our approach supports the dynamic deployment of aspects at run-time

In Chapter 5, we furthermore investigated how the overhead provoked by the additional abstraction mechanisms of AOP can be reduced by taking aspect-specific semantic information into account. We analyzed the characteristics of abstraction mechanisms in high-level programming languages and identified aspect-specific semantic conditions of AOP in general and of ObjectTeams in particular. Based on this analysis, we developed a number of optimizations for ObjectTeams that utilize these semantic conditions. We proposed to use a cache to avoid the overhead of dynamically looking up the base method during a base-call of a replace-callin. Furthermore, we introduced a caching mechanism for role objects in order to optimize the lifting of base objects during aspect execution. We analyzed possible lifting scenarios in order to optimally design the structure and the validation method of the role cache. Finally, we discussed how existing virtual machine optimizations, like quick instructions, can profit from the aspect-specific semantic information.

Evaluation Results In Chapters 6 and 7, we presented the implementation and a detailed evaluation of our approach. We executed micro benchmarks and investigated the effect of our optimizations on a real-world application, both on an Intel PC and on the ARM-based FreeRunner smart phone. Finally, we discussed the transferability of our approach to other VM implementations as well as to other programming languages.

The micro benchmark experiments show that we significantly improved the *aspect activation* of OT/J by developing a VM-internal activation infrastructure as well as a native team activation mechanism. The VM-internal activation infrastructure significantly reduces the runtime overhead of team activation, especially for larger numbers of active teams. Thus, the implicit activation of teams is now fast and its effort is nearly constant, independent from the number of previously activated teams. Based on this adaptation, we achieved a high performance gain by implementing a native team activation mechanism. Of particular importance is the 80 to 90% faster team activation check because this functionality is also involved in aspect execution.

We further optimized the *aspect execution* of OT/J by additional bytecode instructions and a role cache. Our newly introduced bytecode instructions provide an optimized usage of the VM-internal activation infrastructure. This optimization guarantees an efficient team lookup, especially for thread-local team activation. Finally, by the introduction of the `invokeaspect` instruction, which is responsible for the actual aspect execution, we reduced the execution time of adapted base methods by up to 16%. By the introduction of a role cache, we improved the pure lifting mechanism by up to 93%. This leads to an improvement of the general aspect execution by up to 40%.

Eventually, the combination of our optimizations shows an even higher improvement of the execution performance. For example, the execution time of a base method that is adapted by a team with 10 callins of each type could be reduced by 66% on the Intel PC and by 54% on the ARM processor. Furthermore, for a combined execution of (implicit) team activation and aspect execution the improvement is even 73% on the Intel PC and 40% on the ARM.

The effect of our optimizations on the distributed fat is also very promising. For completely unbound base classes no overhead is imposed at all. For the other cases, where a base method is bound, but no aspect functionality has to be executed due to the activation state, we could reduce the distributed fat to up to one third.

To evaluate the overall effect of our optimizations on a real-world application, we used the game application OTPong. OTPong is an interactive GUI application with a modular model-view-controller architecture. Updating the view constitutes an important part of the OTPong application. Our experimental results show that with our approach, we improved the updating of the view by 10% on the Intel PC and by 11% on the smart phone (ARM). As the view is updated very frequently, this improvement is highly relevant for the overall performance of the application.

The evaluation of our approach shows a considerable performance gain for the aspect activation and the aspect execution of ObjectTeams. With our experiments, we demonstrated that our optimizations yield a significant performance gain of up to 90% for commonly used aspect-oriented mechanisms. At the same time, we were able to reduce the code size of the adapted classes, which is also important for small devices. Finally, with our case study on the OTPong game application, we showed that our approach is capable to significantly optimize the execution time of real-word applications.

8.2 Outlook

In this section, we outline future work. There are basically two directions: the completion of our implementation prototype together with ideas for further evaluation as well as the exploration of additional optimization potential.

To provide an implementation that covers the complete range of the ObjectTeams language features and, at the same time, supports the dynamic deployment of new aspects, we need to integrate the implementation of the delegation proxy approach with the rest of our optimizations. Furthermore, we could implement the base-call caching mechanism, we have proposed in Section 5.3.1. Note that in our realization of the `invokeaspects` bytecode, we already cache the method block of base methods for subsequent invocations. However, this does not yet prevent the regular dynamic method lookup. Finally, we could completely integrate the weaving process into the JVM. This would make the load-time weaving of the OTRE obsolete and thus, we could avoid the use of OTEquinox wrappers to execute OT programs.

To further evaluate our approach, more experiments and case studies should be accomplished. Thus, it would be interesting to port existing AOP benchmarks (e.g., from [Asp] and [AOP]) to ObjectTeams. In addition, regular Java benchmark suites, like SPECjvm and JavaGrande could be applied to but they can only measure the distributed fat as they do not contain aspects. Next, our results could be further validated on other embedded devices. Finally, the OTPong case study could be further accomplished to also evaluate the dynamic extension scenarios, which we developed in Section 7.3.2.

As we referred to in Section 2.1.3, roles of a team implicitly inherit from roles with the same name in a super team. This inheritance relationship is no normal type inheritance and is only valid in the context of a surrounding team instance. In Section 4.2.4, we discussed the overhead of the current realization of implicit role inheritance in ObjectTeams. To reduce this overhead, we could add a reference to the implicit super role for role classes in the VM. This could be realized, analog to the super class reference stored in class structures. Thus, we could avoid code duplication and inefficient interfaces method invocation. Furthermore, it could be valuable to investigate the benefit of more sophisticated role caching strategies. Multiple role objects could be cached for one base object, e.g., one for every participating team class. As witnessed by the benchmark results in Section 7.2.5, the effect of the role cache depends on the concrete design of the executed program. To minimize the overhead of frequent cache misses, we could *statically* analyze, whether a base object is often lifted to the same role object. It could also be profitable to dynamically turn the role cache on and off according to the dynamic lifting behavior of a program.

With our approach, we significantly reduce the overhead of common AOP mechanisms. This sets the stage for taking advantage of the advanced modularization capabilities of AOP for developing and dynamically reconfiguring applications for the constantly growing mobile market.

Bibliography

[ACK94] Abhaya Asthana, Mark Cravatts, and Paul Krzyzanowski. An Indoor Wireless System for Personalized Shopping Assistance. In *Proceedings of the Mobile Computing Systems and Applications Workshop 1994*, pages 69–74. IEEE, 1994.

[AFG$^+$05] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, pages 449 –466, 2005.

[AH03] Swen Aussmann and Michael Haupt. Axon - Dynamic AOP through Runtime Inspection and Monitoring. In *Proceedings of ECOOP Workshop ASARTI*, Darmstadt, Germany, July 2003.

[ajH] AspectJ Homepage. `http://www.eclipse.org/aspectj`.

[AMC$^+$07] Vander Alves, Pedro Matos, Jr., Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Transactions on Aspect-Oriented Software Development IV. chapter Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming, pages 117–142. Springer-Verlag, Berlin, Heidelberg, 2007.

[AOP] Codehaus AOP Benchmark Homepage. `http://docs.codehaus.org/display/AW/AOP+Benchmark`.

[Asp] Sable AspectJ Benchmarks. `http://www.sable.mcgill.ca/benchmarks/`.

[Ass08] Mobile Marketing Association. Mobile Applications. `http://www.mmaglobal.com/mobileapplications.pdf`, 2008.

[Ayc03] John Aycock. A Brief History of Just-In-Time. *ACM Comput.*

Surv., 35:97–113, June 2003.

[BADM06] Christoph Bockisch, Mathew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 109–124. ACM, 2006.

[Bel73] James R. Bell. Threaded Code. *cacm*, 16(6):370–372, 1973.

[BHMO04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In Karl Lieberherr, editor, *AOSD 2004 Proceedings*, Lancaster, UK, March 2004. ACM Press.

[Bis10] Julian Bischof. Erweiterung einer Java Virtual Machine um delegationsbasierte Aspektausfuhrungsmechanismen zur Optimierung von ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, 2010.

[Boy08] Brent Boyer. Robust Java benchmarking. `http://www.ibm.com/developerworks/java/library/j-benchmark1.html`, June 2008.

[bug08] buglabs.net. Java VMs Compared. `http://bugblogger.com/java-vms-compared-160/`, 2008.

[Cae] CaesarJ Project. `http://caesarj.org`.

[CFM$^+$97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java Just in Time. *IEEE Micro*, 17:36–43, May 1997.

[CGEN03] Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. Towards Superinstructions for Java Interpreters. In Andreas Krall, editor, *SCOPES: Software and Compilers for Embedded Systems*, Lecture Notes in Computer Science, pages 329–343. Springer, 2003.

[CHM06] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In *JMLC*, volume 4228. Springer, 2006.

[Dmi02] Mikhail Dmitriev. Application of the HotSwap Technology to Advanced Profiling. In *Proceedings of the Workshop on Unanticipated*

Software Evolution, held at ECOOP, 2002.

[DW03] Brian Davis and John Waldron. A Survey of Optimisations for the Java Virtual Machine. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, PPPJ '03, pages 181–183, New York, NY, USA, 2003. Computer Science Press, Inc.

[EG04] M. Anton Ertl and David Gregg. Combining Stack Caching with Dynamic Superinstructions. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 7–14. ACM, 2004.

[EL03] Erik Ernst and David H. Lorenz. Aspects and polymorphism in AspectJ. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 150–157, New York, NY, USA, 2003. ACM.

[Ert95] M. Anton Ertl. Stack Caching for Interpreters. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 315–327, New York, NY, USA, 1995. ACM.

[Esp03] Miklós Espák. Improving Efficiency by Weaving at Run-time. In *Proceedings of GPCE Young Researchers Workshop, held at Net.ObjectDays*, 2003.

[ETK06] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and Replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006.

[FN07] J. Fabry and C. Noguera. AmI: The future is now - a position paper, 2007.

[Fre] Openmoko Wiki: Neo FreeRunner. `http://wiki.openmoko.org/wiki/Neo_FreeRunner`.

[GB04] Philip Greenwood and Lynne Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. Technical report, Proceedings of the 2004 Dynamic Aspect Workshop (DAW04 2004), RIACS, 2004.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, 2005.

[GK04] Michael Austermann Günter Kniesel, Pascal Costanza. JMangler - A Powerful Back-End for Aspect-Oriented Programming. In T. Elrad R. Filman and M. Aksit S. Clarke, editors, *Aspect-oriented Software Development*. Prentice Hall, 2004.

[GK07] Ryan M. Golbeck and Gregor Kiczales. A Machine Code Model for Efficient Advice Dispatch. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*. ACM Press, 2007.

[Gooa] Google. Android. `http://www.android.com/`.

[Goob] Google. Dalvik VM Homepage. `http://code.google.com/p/dalvik/`.

[H0̈6] Paul Häder. Benchmarking und Optimierung der Aspektwebestrategie von ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, 2006.

[HA00] Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java Virtual Machine Interpreters. In *In Proceedings of the 9th International Conference on Compiler Construction (CC' 00). Springer LNCS*, 2000.

[Hau06] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, 1991. Springer-Verlag.

[Her02a] Stephan Herrmann. Composable Designs with UFA. In *In Workshop on Aspect-Oriented Modeling with UML at 1st Intl. Conference on Aspect Oriented Software Development*, 2002.

[Her02b] Stephan Herrmann. Object Teams: Improving Modularity for

Crosscutting Collaborations. In *Proceedings of Net.ObjectDays*, pages 248–264. Springer, 2002.

[Her10] Stephan Herrmann. OTEquinox. `http://wiki.eclipse.org/Category:OTEquinox`, February 2010.

[HG09] Christine Hundt and Sabine Glesner. Optimizing Aspectual Execution Mechanisms for Embedded Applications. In *Proceedings of the First Workshop on Generative Technologies (WGT) 2008*, volume 238, pages 35–45. Electronic Notes in Theoretical Computer Science, 2009.

[HH10] Stephan Herrmann and Christine Hundt. OT Bytecode Attributes. `http://wiki.eclipse.org/OT_Bytecode_Attributes`, August 2010.

[HHM07] Stephan Herrmann, Christine Hundt, and Marco Mosconi. ObjectTeams/Java Language Definition — version 1.0. Technical Report 2007/03, Fak. IV, Technische Universität Berlin, 2007.

[HHP06] Stephan Herrmann, Christine Hundt, and Carsten Pfeiffer. Eclipse Plugin Adaptation with Equinox and ObjectTeams/Java. Eclipse Technology eXchange Workshop (eTX), ECOOP 2006, 2006.

[HM04] Michael Haupt and Mira Mezini. Micro-measurements for Dynamic Aspect-Oriented Systems. In Mathias Weske and Peter Liggesmeyer, editors, *Object-Oriented and Internet-Based Technologies*, Lecture Notes in Computer Science, pages 277–305. Springer Berlin / Heidelberg, 2004.

[HM05] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet*, 11(3):9–30, 2005.

[HM07] Stephan Herrmann and Marco Mosconi. Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity. *Journal of Object Technology*, 6(9):105–125, 2007.

[HMPS07] Christine Hundt, Katharina Mehner, Carsten Pfeiffer, and Dehla Sokenou. Improving Alignment of Crosscutting Features with Code in Product Line Engineering. *Journal of Object Technology*, 6(9):417–436, 2007.

[HS07] Michael Haupt and Hans Schippers. A Machine Model for Aspect-Oriented Programming. In *ECOOP 2007*, volume 4609 of *Lecture*

Notes in Computer Science, pages 501–524. Springer, 2007.

[HSG10] Christine Hundt, Daniel Stöhr, and Sabine Glesner. Optimizing Aspect-oriented Mechanisms for Embedded Applications. In J. Vitek, editor, *TOOLS 2010*, number 6141 in LNCS, pages 137–153, Heidelberg, 2010. Springer.

[Hun03] Christine Hundt. Bytecode-Transformation zur Laufzeitunterstützung von Aspekt-Orientierter Modularisierung mit Object-Teams/Java. Diplomarbeit, Technische Universität Berlin, 2003.

[Jav] JavaGrande Benchmark Homepage. `http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/`.

[JBo] JBossAOP Homepage. `http://www.jboss.org/jbossaop`.

[JDT] Eclipse JDT Website. `http://www.eclipse.org/jdt/`.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley (JWS), 1996.

[JPD] Java Platform Debugger Architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[JPL] Package java.lang.instrument (API Documentation). `http://download.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html`.

[JTSJ07] Nico Janssens, Eddy Truyen, Frans Sanen, and Wouter Joosen. Adding Dynamic Reconfiguration Support to JBoss AOP. In *Proceedings of the 1st workshop on Middleware-application interaction: in conjunction with Euro-Sys 2007*, MAI '07, pages 1–8. ACM, 2007.

[KB08] Dong Kwan Kim and Shawn Bohner. Dynamic Reconfiguration for Java Applications using AOP. In *Southeastcon, 2008. IEEE*, pages 210 –215, 2008.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, number 1241 in LNCS, pages 220–243. Springer, 1997.

[KVM00] White Paper on KVM and CLDC. `http://java.sun.com/`

`products/cldc/wp/`, 2000.

[LI09] Christian Licoppe and Yoriko Inada. The Mogi Location-aware Community and its Interaction Order: ”Augmented” Face-to-Face Encounters as Rare, Public Performances. In *Proceedings of MobileHCI09*, Bonn, Germany, 2009. ACM.

[Lou] Robert Lougher. JamVM Homepage. `http://jamvm.sourceforge.net`.

[LR05] Peter Liggesmeyer and Dieter Rombach, editors. *Software Engineering eingebetteter Systeme*. Spektrum Akademischer Verlag, 2005.

[LY96] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., first edition, 1996.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., second edition, 1999.

[Meh03] Katharina Mehner. Performante Überwachung von Methodenaufrufen mit JPDA. *JavaSPEKTRUM*, 2003(6):42–46, 2003.

[MJV$^+$97] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pierre Verbaeten. Aspects should not die. Position paper at the ECOOP ’97 workshop on Asepct-Oriented Programming, 1997.

[MTC$^+$07] H. Mügge, É. Tanter, P. Cherrier, J. Dedecker, C. Lopes, and (eds.) Cebulla, M. Proceedings of the 3rd ECOOP workshop on Object Technology for Ambient Intelligence and Pervasive Computing (OT4AmI 2007). Technical Report 2007-12, Technische Universität Berlin, Germany, 2007.

[Nic09] Carlo U. Nicola. Einblicke in die Dalvik Virtual Machine. *IMVS Fokus Report*, pages 5 – 12, 2009.

[Noka] Nokia. Cross-platform application and UI framework. `http://qt.nokia.com/`.

[Nokb] Nokia. Symbian at Nokia. `http://symbian.nokia.com/`.

[otH] Object Teams Homepage. `http://www.eclipse.org/objectteams`.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In Mehmet Aksit, editor, *AOSD 2003 Proceedings*, Boston Massachusetts, March 2003. ACM Press.

[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In Gregor Kiczales, editor, *AOSD 2002 Proceedings*, Enschede, The Netherlands, March 2002. ACM Press.

[pho] PhoneME Homepage. `http://java.net/projects/phoneme/`.

[Pon] The Pong Computer Game. `http://en.wikipedia.org/wiki/Pong`.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 322–332, New York, NY, USA, 1995. ACM.

[PZ03] E. Kessler Piveta and L. Carlos Zancanella. Aspect Weaving Strategies. *Journal of Universal Computer Science*, 9(8):970–983, 2003.

[QSVY06] Dan Quinlan, Markus Schordan, Richard Vuduc, and Qing Yi. Annotating User-Defined Abstractions for Optimization. In *In Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE, 2006.

[QSYS04] Dan Quinlan, Markus Schordan, Qing Yi, and Andreas Saebjornsen. Classification and Utilization of Abstractions for Optimization. In *In Proc. 1st International Symposium on Leveraging Applications of Formal Methods, Paphos*, pages 86–101, 2004.

[RFSC04] T. Elrad R. Filman and M. Aksit S. Clarke, editors. *Aspect-oriented Software Development*. Prentice Hall, 2004.

[RSC07] Tobias Rho, Mark Schmatz, and Armin B. Cremers. Towards Context-Sensitive Service Aspects. [MTC+07].

[RTA05] Dimitrios Raptis, Nikolaos Tselios, and Nikolaos Avouris. Context-based Design of Mobile Applications for Museums: A Survey of Existing Practices. In *In MobileHCI '05: Proceedings of the 7th international conference on Human computer interaction* *with mobile devices & services*, pages 153–160. ACM Press, 2005.

[Sch04] Klaus-Dieter Schmatz. *Java Micro Edition. Entwicklung mobiler Anwendungen mit CLDC und MIDP.* Dpunkt Verlag, 2004.

[Sch07] Mark Schmatz. Generische kontext-sensitive Aspekte für Service-Orientierte Architekturen. Diplomarbeit, Institut für Angewandte Informatik, Rheinische Friedrich-Wilhelms Universität Bonn, 2007.

[SCT03] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A Selective, Just-In-Time Aspect Weaver. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 189–208, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[SHH09] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1944–1951, New York, NY, USA, 2009. ACM.

[SLU05] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in AOP with AspectC++. In *New Trends in Software Methodologies Tools and Techniques*, Frontiers in Artificial Intelligence and Applications, pages 33–53, 2005.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, June 2005.

[SPE] SPECjvm2008 Homepage. `http://www.spec.org/jvm2008/`.

[Stö10] Daniel Stöhr. Modifizierung einer JVM für effizienten Aspect-Dispatch in ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, 2010.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison Wesley, 1994.

[TM99] Tom Tourwe and Wolfgang De Meuter. Optimizing Object-Oriented Languages through Architectural Transformations. In *In 8th International Conference on Compiler Construction*, pages 244–258. Springer-Verlag, 1999.

[VH96] Jan Vitek and R. Nigel Horspool. Compact Dispatch Tables for

Dynamically Typed Object Oriented Languages. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 309–325, London, UK, 1996. Springer-Verlag.

[WNGG08] Kevin Williams, Albert Noll, Andreas Gal, and David Gregg. Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, New York, NY, USA, 2008. ACM.

List of Figures

List of Tables

Christine Hundt's Publications

[HG09] Christine Hundt and Sabine Glesner. Optimizing Aspectual Execution Mechanisms for Embedded Applications. In *Proceedings of the First Workshop on Generative Technologies (WGT) 2008*, volume 238, pages 35–45. Electronic Notes in Theoretical Computer Science, 2009.

[HHM04a] Stephan Herrmann, Christine Hundt, and Katharina Mehner. Mapping Use Case Level Aspects to ObjectTeams/Java. Workshop on Early Aspects, OOPSLA 2004, 2004.

[HHM04b] Stephan Herrmann, Christine Hundt, and Katharina Mehner. Translation Polymorphism in Object Teams. Technical Report 2004/05, Fak. IV, Technische Universität Berlin, 2004.

[HHM07] Stephan Herrmann, Christine Hundt, and Marco Mosconi. ObjectTeams/Java Language Definition — version 1.0. Technical Report 2007/03, Fak. IV, Technische Universität Berlin, 2007.

[HHMW05] Stephan Herrmann, Christine Hundt, Katharina Mehner, and Jan Wloka. Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation. In *In Dynamic Aspects Workshop (DAW'05), at AOSD 2005*, pages 93–101, 2005.

[HHP06] Stephan Herrmann, Christine Hundt, and Carsten Pfeiffer. Eclipse Plugin Adaptation with Equinox and ObjectTeams/Java. Eclipse Technology eXchange Workshop (eTX), ECOOP 2006, 2006.

[HMPS07] Christine Hundt, Katharina Mehner, Carsten Pfeiffer, and Dehla Sokenou. Improving Alignment of Crosscutting Features with Code in Product Line Engineering. *Journal of Object Technology*, 6(9):417–436, 2007.

[HSG10] Christine Hundt, Daniel Stöhr, and Sabine Glesner. Optimizing Aspect-oriented Mechanisms for Embedded Applications. In J. Vitek, editor, *TOOLS 2010*, number 6141 in LNCS, pages 137–153, Heidelberg, 2010. Springer.

[Hun03] Christine Hundt. Bytecode-Transformation zur Laufzeitunterstützung von Aspekt-Orientierter Modularisierung mit ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, 2003.

Supervised Diploma Theses

[Bis10] Julian Bischof. Erweiterung einer Java Virtual Machine um delegationsbasierte Aspektausfuhrungsmechanismen zur Optimierung von ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, 2010.

[Fei07] Marko Feistkorn. Abbildung von produktlinienorientierten Featurediagrammen auf aspektorientierte Implementierungsmodule. Diplomarbeit, Technische Universität Berlin, 2007.

[Flü06] Michael Flüh. Schemaerhaltende Bytecodetransformationen zum Aspektweben zur Programmlaufzeit. Diplomarbeit, Technische Universität Berlin, 2006.

[H0̈6] Paul Häder. Benchmarking und Optimierung der Aspektwebestrategie von ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, 2006.

[Stö10] Daniel Stöhr. Modifizierung einer JVM für effizienten Aspect-Dispatch in ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, 2010.