An Efficient and Flexible Implementation of Aspect-Oriented Languages

Dr.-Ing. Christoph Bockisch

Software Engineering Group
Universiteit Twente
P.O. Box 217, 7500 AE Enschede, Niederlande
c.m.bockisch@cs.utwente.nl

Abstract: Aspekt-orientierte Programmiersprachen werden zunehmend in der Industrie eingesetzt, da sie die Strukturierung des Quellcodes und damit dessen Wartbarkeit verbessern. Implementierungen dieser Sprachen compilieren den Quelltext allerdings zu einer Intermediate-Repräsentation, die nicht auf Aspekt-orientierte Sprach-Konzepte ausgerichtet ist, wodurch effiziente Laufzeit-Optimierungen für diese verhindert werden, wie sie für Objekt-orientierte Sprache-Konzepte üblich sind. In dieser Arbeit wird eine Architektur für die Implementierung Aspekt-orientierter Sprachen vorgeschlagen, die deren Konzepte in der Intermediate-Repräsentation erhält. Darauf aufbauend wurden spezielle Laufzeit-Optimierungen für Aspekt-orientierte Konzepte entwickelt, die deren Effizienz bis zu 1000-fach gegenüber existierenden Implementierungen Aspekt-orientierter Sprachen steigern.

1 Einführung

Compiler für moderne *Objekt-orientierte Programmiersprachen* generieren Code in einer Plattform-unabhängigen Intermediate-Sprache [LY99, CLI06], die die Konzepte der Quell-Sprache erhält; zum Beispiel können Klassen, Felder, Methoden und virtueller oder statischer Methodendispatch[1] direkt im Intermediate-Code identifiziert werden. Diese deklarative Repräsentation der Konzepte aus der Quell-Sprache in der Intermediate-Sprache ermöglicht höchst effiziente adaptive und spekulative Optimierungen des laufenden Programms, die sonst nicht möglich wären.

Im Gegensatz hierzu werden die Konstrukte von *Aspekt-orientierten Programmiersprachen* – die zu einer besseren Struktur des Quellcodes („Separation of Concerns“) führen können – üblicherweise dadurch realisiert, dass sie zu herkömmlichen Intermediate-Instruktionen compiliert werden oder dass sie Transformationen im Intermediate-Code bewirken. Auf diese Weise bleibt die Semantik Aspekt-orientierter Konstrukte auf der Ebene der Intermediate-Sprache nicht deklarativ erhalten und hoch-performante Optimierungen werden verhindert.

[1] Mit „Dispatch“ bezeichnet man die Logik, aus einer Menge von möglichen Zielmethoden die für einen Aufruf richtige auszuwählen.

Das Aspekt-orientierte Programmierparadigma spielt in zunehmendem Maß auch in der Industrie eine wichtige Rolle. Große Unternehmen wie beispielsweise IBM, Sun Microsystems und Oracle verwenden Aspekt-orientierte Sprachen in kommerzieller Software [oE], und sogar im Embedded-Sektor wird Aspekt-Orientierung eingesetzt wie das Beispiel des WEAVR Tools von Motorola zeigt [CvdBE07]. Daher ist es von allgemeinem Interesse, die Effizienz von Aspekt-orientierten Programmiersprachen zu verbessern. Das wird in dieser Arbeit dadurch erreicht, dass eine deklarative Intermediate-Repräsentation für Aspekt-orientierte Sprachkonstrukte definiert wird. Darauf aufsetzend werden Optimierungen für diese Konstrukte entwickelt, wie sie bisher nur Objekt-orientierten Sprachen zur Verfügung standen.

2 Hintergrund

Aspekt-orientierte (AO) Programmiersprachen, die mit dem hier vorgeschlagenen Ansatz unterstützt werden sollen, erweitern Objekt-orientierte Sprachen um zwei wesentlichen Mechanismen. Erstens kann über verschiedene Ereignisse während der Programmausführung wie beispielsweise Methodenaufrufe quantifiziert werden, und zweitens kann das Programmverhalten an solchen Ereignissen verändert werden. Die Quantifizierung ermöglicht es, Ereignisse aufgrund sowohl lexikalischer Eigenschaften (beispielsweise des Namens der aufgerufenen Methode) als auch dynamischer Eigenschaften (beispielsweise des aktuellen Kontrollflusses) auszuwählen. Das Programmverhalten kann dadurch verändert werden, dass vor oder nach quantifizierten Ereignissen zusätzliche Operationen durchgeführt werden. Es ist sogar möglich, die eigentliche Operation des Ereignisses zu unterdrücken; zum Beispiel kann so ein Methodenaufruf zu einer anderen Methode umgeleitet werden.

Anhand des folgenden Beispiels wird die Funktionsweise und gebräuchliche Terminologie von AO Sprachen erläutert. Das Beispiel ist [KHH+01] entnommen und zeigt ein Programmstück in der Sprache AspectJ [KHH+01]. Der Programmauszug sorgt dafür, dass in einem graphischen Editor die Ansicht (Display) aktualisiert wird, sobald sich etwas am Modell (Figure) ändert. Der Ausdruck in den Zeilen 3–4 wird *Pointcut* genannt und hat mehrere Aufgaben. Der Teilausdruck in Zeile 3 quantifiziert über Ereignisse durch deren lexikalische Eigenschaften; die ausgewählten Instruktionen werden *Join-Point-Schatten* genannt. Im Beispiel werden alle Methodenaufrufe ausgewählt, die sich an eine Methode richten, die in der Klasse Figure definiert ist und deren Namen mit set beginnt. Zeile 4 definiert einen Ausdruck, der *wahr* ergibt, wenn der aktuelle Kontrollfluss nicht rekursiv in der Ausführung der gleichen Methoden ist. Die Ereignisse, die von den Teilausdrücken in Zeile 3–4 ausgewählt werden, heißen *Join-Point*. Nachdem (das wird durch das Schlüsselwort **after** spezifiziert) ein Join-Point stattfindet, der vom Pointcut ausgewählt wird, wird der Block (*Advice* genannt) ausgeführt, der in Zeile 5 definiert ist. Die Ereignisse, über die quantifiziert wird, werden von dem so genannten *Basis-Programm* erzeugt, das in einer nicht Aspekt-orientierten *Basis-Sprache* implementiert ist. Advices werden meistens ebenfalls in der Basis-Sprache implementiert.

```
1 aspect DisplayUpdate {
2   after(Figure f) :
3     call(* Figure.set*(..)) &&
4     !cflow(call(* Figure.set*(..))) &&
5     { Display.notify(f); }
6 }
```

Listing 1: Implementierung in der Aspekt-orientierten Sprache AspectJ, die eine Anzeige benachrichtigt, wenn sich ein graphisches Objekt ändert.

3 Architektur

Um die Sprach-Implementierungstechniken für Aspekt-orientierte Programmiersprachen zu verbessern, wird in dieser Arbeit die *Architektur für Aspekt-Sprachen-Implementierungen* (englisch *Aspect-Language Implementation Architecture*, ALIA) vorgeschlagen, welche unter anderem vorschreibt, dass eine Intermediate-Repräsentation existiert, die die Aspekt-orientierten Konstrukte aus dem Quell-Programm erhält. Eine zentrale Komponente dieser Architektur ist ein erweiterbares und flexibles Meta-Model von Aspekt-orientierten Konzepten, das als Schnittstelle zwischen Compilern und Ausführungsumgebungen fungiert. Das Meta-Modell wird anstelle einer dedizierten Intermediate-Sprache verwendet, da es flexibel für beliebige AO Sprachen erweiterbar ist. Das ist wichtig, da der vorgeschlagene Ansatz nicht nur dazu dienen soll, existierende AO Sprachen besser zu implementieren, sondern auch zukünftige Sprachen zu unterstützen.

Die ALIA Architektur und das Meta-Model sind für Java-basierte Sprachen in dem *Framework zum Implementieren von Aspekt-Sprachen* (englisch *Framework for Implementing Aspect Languages*, FIAL) beziehungsweise dem *Sprachunabhängigen Aspekt-Meta-Model* (englisch *Language-Independent Aspect Meta-model*, LIAM), das Teil des Frameworks ist, umgesetzt. FIAL implementiert generisch Arbeitsabläufe, die von einer Ausführungsumgebung benötigt werden, welche Aspekte ausführt, die mit LIAM definiert sind. Zusätzlich zu der deklarativen Intermediate-Repräsentation von Aspekt-orientierten Konzepten behandelt ALIA – und FIAL als dessen Verkörperung – Join-Points so, dass sie spät an einen Join-Point-Schatten gebunden werden. Analog zu der Objekt-orientierten Terminologie für spät gebundene Methoden werden Join-Points in ALIA als *virtuell* bezeichnet. Die deklarative Repräsentation von Aspekt-orientierten Konzepten in der Intermediate-Repräsentation und das Behandeln von Join-Points als virtuell ermöglichen es, neue und effektive Optimierungen für Aspekt-orientierte Programme zu entwickeln.

Abbildung 1 zeigt eine übersicht über den Ablauf des Compilierens und Ausführens von Programmen in Sprach-Implementierungen, die der hier vorgeschlagenen ALIA-Architektur folgen: die Teile des Quelltextes, die in der Basis-Sprache ausdrückbar sind, werden in deren Intermediate-Code übersetzt; alle Programmteile, die Aspekt-orientierte Konzepte nutzen, werden in einem Modell dargestellt dessen Struktur von einem Meta-Modell definiert wird, das aus den Kern-Konzepten von AO Sprachen besteht.

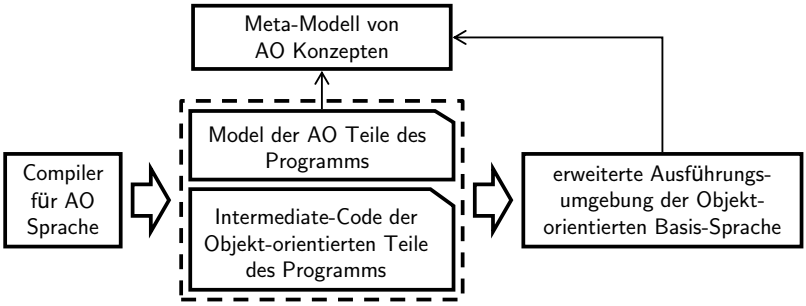

Abbildung 1: Lebenszyklus eines Programms in der ALIA-Architektur.

Durch die Verwendung einer Intermediate-Repräsentation für Aspekt-orientierte Konzepte, wird die Ausführungsumgebung von der Programmiersprache entkoppelt, was es ermöglicht, nur eine Ausführungsumgebung für alle Sprachen gemeinsam zu implementieren. Trotzdem wurden verschiedene auf Java basierende Ausführungsumgebungen entwickelt, um zu demonstrieren, dass der vorgeschlagene Ansatz für Umgebungen mit verschiedenen Charakteristiken anwendbar ist. Gemeinsame Arbeitsabläufe und allgemeine Aufgaben sind generisch in FIAL implementiert wobei Umgebungs-spezifische Funktionalität abstrakt gelassen ist. Die verschiedenen Ausführungsumgebungen sind als Instantiierung von FIAL realisiert.

3.1 Ein Meta-Modell für Aspekte-Orientierte Konzepte

LIAM ist als ein Paket von Java Klassen realisiert, wobei für jede Kategorie von Aspekt-orientierten Sprachkonzepten (zum Beispiel primitive Ausdrücke zur Quantifizierung von Join-Points) von LIAM eine Meta-Entität als abstrakte Klasse zur Verfügung gestellt wird. Für jedes konkrete Konzept einer Sprache (beispielsweise das Quantifizierungskriterium Kontrollfluss), das auf LIAM abgebildet wird, wird eine Subklasse der entsprechenden abstrakten Klasse erstellt. Aus Instanzen dieser Klassen wird das so genannte LIAM-Modell aufgebaut, das eine deklarative Definition der Aspekt-orientierten Konzepte im Quelltext darstellt.

3.2 Ein Framework für Aspekt-Orientierte Ausführungsumgebungen

Die Intermediate-Repräsentation in Form eines LIAM-Modells wird an die FIAL-Instantiierung übergeben. Teil von FIALs generischer Funktionalität ist es, die LIAM-Modelle partiell auszuwerten und daraus so genannte *Dispatch-Modelle* für jeden Join-Point-Schatten zu erstellen, die sich aus LIAM-Entitäten zusammensetzen.

Da die in dieser Arbeit behandelten Aspekt-orientierten Sprachen die Java Sprache erweitern, erweitert eine FIAL-Instantiierung eine Java Ausführungsumgebung (virtuelle Maschine) und realisiert FIALs abstrakte Funktionalität durch Interaktion mit dieser. Am wichtigsten ist hierbei die Funktionalität, Code für einen Join-Point-Schatten aus dem deklarativen Dispatch-Modell zu erstellen, das FIAL bereitstellt. Weiterhin muss die FIAL-Instantiierung in der Lage sein, den Code für einen Join-Point-Schatten neu zu generieren, wenn sich Aspekt-Deklarationen zur Laufzeit ändern, was zum Beispiel geschieht, wenn ein Aspekt dynamisch aktiviert oder deaktiviert wird.

Die Dispatch-Modelle und LIAM-Entitäten sind so definiert, dass eine Standard-Strategie ausführbaren Java Bytecode daraus generieren kann. Dazu muss jede LIAM-Entität eine normale Java Methode bereitstellen, die die Semantik der Entität implementiert, beispielsweise durch Verwendung von Reflection [Jav, JVM]. Der generierte Code ruft dann diese Methode auf. Dadurch können neue LIAM-Entitäten hinzugefügt werden, ohne die Ausführungsumgebung zu verändern.

Für AO Sprach-Konzepte, die einer FIAL-Instantiierung bekannt sind, können optimierte Code-Generierungsstrategien angewendet werden. Diese speziellen Strategien können dabei die Mächtigkeit der erweiterten Basis-Umgebung voll ausnutzen, was erst durch die Deklarativität der Dispatch-Modelle ermöglicht wird. Im Rahmen dieser Arbeit wurde eine FIAL-Instantiierung (STEAMLOOM^{ALIA}) in eine virtuelle Maschine für Java (die von IBM entwickelte Jikes Research Virtual Machine) integriert, wodurch der generierte Code weit über das Maß hinaus optimiert werden kann, das durch Bytecode-Generierung erreicht werden kann, die von aktuellen Sprach-Implementierungen praktiziert wird.

4 Integration von AO Konzepten in die Virtuelle Maschine

Die Ausführung von Bytecode-Instruktionen erfolgt in modernen virtuellen Maschinen, indem die Instruktionen einer Methode zu nativem Maschinencode compiliert werden bevor die Methode zum ersten Mal ausgeführt wird [DS84] (dies wird just-in-time Compilierung genannt). Während dieses Schritts, kann die virtuelle Maschine Optimierungen durchführen.

In dieser Arbeit wurden insbesondere Code-Generierungsstrategien für die effiziente Unterstützung von zwei Aspekt-orientierten Konzepten implementiert, die durch die tiefe Integration in die virtuelle Maschine ermöglicht wurden. Diese Konzepte sind das dynamische Aktivieren und Deaktivieren von Aspekten (so genanntes dynamisches Deployment) und der primitive Quantifizierungsmechanismus **cflow**. Die hier entwickelten Code-Generierungsstrategien beschleunigen diese Sprach-Konzepte um den Faktor 1000 beziehungsweise 100 gegenüber vergleichbaren Ansätzen, die durch das Potenzial des erzeugten herkömmlichen Intermediate-Codes begrenzt sind.

4.1 Optimiertes Aspekt-Deployment

Die Optimierung von dynamischem Deployment beruht darauf, dass in STEAMLOOM^{ALIA} Join-Point-Schatten als Aufruf von virtuellen Join-Points behandelt werden und nutzt Gemeinsamkeiten zu virtuellen Methodenaufrufen aus. Etablierte und höchst effiziente Optimierungen für virtuellen Methodendispatch wurden angepasst, die die spekulative Annahme treffen, dass das Ziel des Methodenaufrufs zur Compile-Zeit bekannt ist. Dann kann der Code des Ziels direkt an der Aufrufstelle eingefügt werden [AFG+05], was *Inlining* genannt wird. Gleichzeitig muss dafür Sorge getragen werden, dass diese Optimierung rückgängig gemacht werden kann, wenn die Annahme nicht mehr zutrifft.

Aus der Palette der möglichen Techniken, um spekulative Optimierungen rückgängig zu machen, wurden in dieser Arbeit On-Stack-Replacement und Code-Patching [AFG+05] verwendet, um die Implementierung von Join-Points an Join-Point-Schatten spekulativ zu inlinen.

4.2 Optimiertes Quantifizieren über den Kontrollfluss

Neben der Unterstützung für effizientes dynamisches Deployment, ist in dieser Arbeit auch eine neuartige Optimierung für den primitiven Quantifizierungsmechanismus **cflow** implementiert worden. Dieser Mechanismus wählt zur Laufzeit einen Join-Point aus, wenn sich die Ausführung gerade im Kontrollfluss beispielsweise einer bestimmten Methode befindet. Man sagt, dass diese Methode den Kontrollfluss *konstituiert*, und dass der Join-Point von diesem Kontrollfluss *abhängt*. Für eine Realisierung dieses Konzepts muss am Anfang und Ende der Ausführung der konstituierenden Methode protokolliert werden, dass der Kontrollfluss betreten beziehungsweise verlassen wurde. An abhängigen Join-Points kann so einfach geprüft werden, ob der Kontrollfluss aktiv ist oder nicht [ACH+05]. Für eine effiziente Implementierung dieser Strategie muss also sowohl das Protokollieren als auch der Zugriff auf das Protokoll optimiert werden.

Die Integration in die virtuelle Maschine hat es in dieser Arbeit ermöglicht, das Stack-Layout, den Multi-Threading-Mechanismus und den just-in-time Compiler anzupassen, um beide für **cflow** relevanten Operationen zu optimieren. Diese Optimierung basiert zum Teil auf dem Konzept von Thin Guards, das in [AR02] beschrieben wird.

5 Evaluation

Mit dem spekulativen Inlinen von Join-Points und der Modifikation des Stack-Layouts und des Multi-Threading-Systems für die schnelle Nachverfolgung des Kontrollflusses sind in dieser Arbeit zwei besonders mächtige Optimierungen entwickelt worden. Um diese Optimierungen zu evaluieren, muss berücksichtigt werden, dass in beiden Fällen eine unterstützende Infrastruktur benötigt wird. Im Fall des dynamischen Deployments ist dies die Möglichkeit, das Inlining rückgängig zu machen, und im **cflow**-Fall ist dies das

Protokollieren des Kontrollflusses. Der Effekt dieser Infrastruktur auf die Laufzeit muss genauso gemessen werden wie die Performance der Operationen Deployment und **cflow**-Test selbst.

Da es keine Standard-Benchmarks für Aspekt-orientierte Ausführungsumgebungen gibt, die die Performance dieser Konzepte in einem realitätsnahen Kontext messen, wurden in dieser Arbeit zwei Arten von Benchmarks entwickelt und durchgeführt. Zum einen sind das Micro-Benchmarks, die die Effizienz einzelner Operationen messen. Zum anderen wurden basierend auf der SPEC JVM98 Benchmark-Suite Macro-Benchmarks entwickelt, die Aspekt-orientierte Operationen in einem realistischen Kontext anwenden.

5.1 Micro-Benchmarks

In dieser Arbeit wurden Micro-Benchmarks entwickelt, welche die Operationen Konstituieren und Testen eines Kontrollflusses, und dynamisches Deployment messen. Die Ergebnisse der Micro-Benchmarks sind in Tabelle 1 dargestellt. Die Ausführungsumgebungen AspectWerkz und JAsCo erlauben es ähnlich wie die hier entwickelte Umgebung STEAMLOOM^{ALIA}, Aspekte dynamisch zu deployen, wobei im Fall von JAsCo immer ein Fehler auftritt, wenn der Aspekt im Benchmark undeployt wird. Die übrigen Laufzeitumgebungen unterstützen kein dynamisches Deployment. Um die benötigte Zeit zum Deployen eines Aspektes zu bestimmen, wurde ein Aspekt, der jeden Methodenaufruf betrifft, in alle Benchmark-Anwendungen der SPEC JVM98 Suite deployt beziehungsweise undeployt. Die in Tabelle 1 in den Zeilen *deploy* und *undeploy* gezeigten Zeiten sind jeweils der Durchschnitt über alle Anwendungen. Die Tabelle zeigt, dass die hier entwickelte Technik zur Optimierung von dynamischem Deployment um drei Größenordnungen schneller ist als in vergleichbaren Aspekt-orientierten Umgebungen.

Die Zeilen *konst.* und *test* in Tabelle 1 zeigen wie sehr ein Methodenaufruf verlangsamt wird, wenn die Methode einen Kontrollfluss konstituiert beziehungsweise zusätzlich einen **cflow**-Test durchführt. `ajc` ist der Standard-Compiler der Sprache AspectJ und wird in vielen kommerziellen Projekten eingesetzt. `abc` ist ebenfalls ein Compiler für die AspectJ Sprache; hierbei handelt es sich um ein Forschungsprojekt, in dem Bytecode-Optimierungen untersucht werden. Der `abc` Compiler implementiert eine spezielle Optimierung, die aber nur für single-threaded (`st`) Anwendungen möglich ist und bei multi-threaded (`mt`) Anwendungen nicht verwendet werden kann. Die Tabelle zeigt, dass JAsCo die Ausführung von konstituierenden Methoden am wenigsten verlangsamt, dafür aber inakzeptabel langsam ist, wenn der aktive Kontrollfluss abgefragt werden muss. Im single-threaded-Fall ist `abc` deutlich schneller als `ajc`, im allgemeinen multi-threaded-Fall sind sie jedoch vergleichbar. Betrachtet man den multi-threaded-Fall und lässt den Vorteil von JAsCo bei der Kontrollfluss-Konstituierung außer Acht, so ist die Laufzeit des **cflow**-Tests von STEAMLOOM^{ALIA} zwei Größenordnungen schneller als der aller anderen Umgebungen. Die Kontrollfluss-Konstituierung von STEAMLOOM^{ALIA} ist unter denselben Bedingungen mindestens zehn mal schneller. Und selbst im single-threaded-Fall ist STEAMLOOM^{ALIA} noch etwa zehn beziehungsweise dreimal so schnell wie die spezielle Optimierung von `abc`.

	STEAMLOOM^ALIA	Aspect-Werkz	JAsCo	ajc	abc st	abc mt
deploy	3 ms	3.360 ms	2.685 ms	N/A	N/A	N/A
undeploy	3 ms	3.922 ms	–	N/A	N/A	N/A
konst.	166,0 %	7.370,0 %	134,0 %	3.839,3 %	505,3 %	1.855,7 %
test	66,5 %	4.240,0 %	$5{,}6{\cdot}10^6$ %	1.379,7 %	686,8 %	2.108,1 %

Tabelle 1: Performance einzelner AO Konzepte gemessen mit Micro-Benchmarks.

	STEAMLOOM^ALIA	Aspect-Werkz	JAsCo	ajc	abc
dyn. Depl.	-0,3 %	121,0 %	1,7 %	N/A	N/A
cflow	6,8 %	328,0 %	N/A	104,4 %	11,0 %

Tabelle 2: Overhead der Infrastruktur von Aspekt-orientierten Konzepten gemessen mit der SPEC JVM98 Macro-Benchmark-Suite.

5.2 Macro-Benchmarks

Da bisher keine Macro-Benchmarks für Aspekt-orientierte Ausführungsumgebungen existieren, wurden in dieser Arbeit neue entwickelt. Diese basieren auf der etablierten SPEC JVM98 Benchmark-Suite [SPE] für Java virtuelle Maschinen und stellen somit einen Kontext dar, wie er in echten Java Anwendungen anzutreffen ist. In die Benchmark-Anwendungen wurden zusätzlich Aspekte eingefügt, die dynamisch deployt werden beziehungsweise die **cflow**-Quantifizierung benutzen. Die Benchmark-Anwendungen wurden jeweils mit und ohne die zusätzlichen Aspekte mit jeder untersuchten Umgebung ausgeführt. Aus den Ausführungszeiten der Benchmark-Anwendungen in beiden Fällen ist es möglich pro Ausführungsumgebung zu errechnen, wie stark die Ausführung von den Aspekt-orientierten Konzepten verlangsamt wurde.

In Tabelle 2 wird der Laufzeitoverhead gezeigt, der von der Infrastruktur für dynamisches Deployment beziehungsweise **cflow**-Unterstützung von den einzelnen Laufzeitumgebungen in realistischen Programmkontexten verursacht wird. Es wird jeweils der Durchschnitt über den Overhead aller SPEC JVM98 Benchmark-Anwendung dargestellt; dabei sind in der Suite sowohl single-threaded als auch multi-threaded Anwendungen enthalten. Aufgrund der schlechten Laufzeit des **cflow**-Tests in JAsCo, wurde für diese Ausführungsumgebung der Overhead der **cflow**-Infrastruktur nicht ermittelt. Die Ergebnisse zeigen, dass STEAMLOOM^ALIA auch in realistischen Ausführungskontexten den kleinsten Laufzeitoverhead aufweist. Im Fall der Infrastruktur für dynamisches Deployment ist sogar gar kein Overhead vorhanden[2].

[2] Der negative Wert, also sogar eine Beschleunigung der Ausführung, war nicht zu erwarten. Eine Beschleunigung oder Verlangsamungen dieser Größenordnung, kann durch implizite Effekte veränderter Speicherbelegung, die jedoch nicht direkt mit der Optimierung im Zusammenhang stehen, erklärt werden und sollte ignoriert werden.

6 Fazit

Diese Arbeit enthält drei wesentliche Beiträge. Erstens wird eine Architektur zur Implementierung Aspekt-orientierter Sprachen vorgeschlagen, die auf einem Meta-Modell als Intermediate-Repräsentation für Aspekt-orientierte Sprachkonzepte beruht. Ein Framework wurde entwickelt, das diese Architektur verkörpert und das die Implementierung verschiedener Ausführungsumgebungen und Programmiersprachen unterstützt. Um dies zu untermauern, wurden unter anderem eine Ausführungsumgebung implementiert, die auf Bytecode-Generierung beruht und portabel ist, sowie eine Umgebung, die Aspekt-orientierte Konzepte tief in eine virtuelle Maschine integriert. Es wurden mehrere populäre und mächtige Aspekt-orientierte Programmiersprachen, darunter der de facto Industriestandard AspectJ, auf das entwickelte Meta-Modell abgebildet. Für AspectJ, JAsCo und Compose* wurde der Prozess, den Quelltext in LIAM-Modelle zu übersetzen sogar automatisiert. Weiterhin wird in der Arbeit anhand eines Tutorials gezeigt, wie neue Aspekt-orientierte Sprachen einfach mit Hilfe des Frameworks implementiert werden können.

Zweitens wurden Optimierungen für dynamisches Aspekt-Deployment und den **cflow** Quantifizierungs-Mechanismus entwickelt, die nur durch die Integration in eine virtuelle Maschine ermöglicht wurden, da sie von direktem Zugriff auf Hardwareressourcen Gebrauch machen. Dadurch sind diese Optimierungen mächtiger als konventionelle AO Sprach-Implementierungen, die Java Bytecode generieren.

Drittens wurden Verfahren entwickelt, um dynamische Sprach-Charakteristiken wie zum Beispiel Aspekt-Deployment und **cflow**-Quantifizierung zu benchmarken, die zum Teil auf realistischen Programmen beruhen. Weiterhin wurden in einer Fallstudie die aktuell verfügbaren Aspekt-orientierten Sprach-Implementierungen verglichen. Dadurch wurde bestätigt, dass die hier entwickelten Optimierungen existierende Sprach-Implementierungen weit übertreffen.

Die hier entwickelten Optimierungen ergeben noch einen weiteren, konzeptionellen Beitrag dieser Arbeit. Die verbesserte Effizienz, die sich aus der Integration mit der virtuellen Maschine ergibt, zeigt, dass Aspekt-orientierte Quantifizierungs-Mechanismen wie **cflow** und Deployment einen Vorteil gegenüber herkömmlichen Programmiersprachen haben, der bislang nicht diskutiert wurde. Bislang war das Hauptargument für Aspekt-orientierte Programmierung die verbesserte Modularität. Während das sicherlich ein wesentlicher Vorteil von AO Programmierung ist, zeigt diese Arbeit, dass AO Sprachen auch das Potential haben, die Effizienz von Programmen verglichen mit einer Objekt-orientierten Implementierung zu steigern. Der Schlüssel hierzu ist, die Aspekt-orientierten Konzepte auch zur Laufzeit explizit zu machen und dadurch umfassende Optimierungen zu ermöglichen, die außerhalb der Reichweite konventioneller Sprach-Implementierungen liegen.

Literatur

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam und Ju-

lian Tibble. Optimising AspectJ. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, 2005.

[AFG+05] Matthew Arnold, Stephen Fink, David Grove, Michael Hind und Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. In *Proceedings of the Institute of Electrical and Electronics Engineers*. IEEE, 2005.

[AR02] Matthew Arnold und Barbara G. Ryder. Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer Verlag, 2002.

[CLI06] Common Language Infrastructure (CLI). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, 2006.

[CvdBE07] Thomas Cottenier, Aswin van den Berg und Tzilla Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *Industrial Track at the 6th Internatonal Conference on Asect-Orienter Software Development*, 2007.

[DS84] Peter Deutsch und Allan Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the Symposium on Principles of Programming Languages*, 1984.

[Jav] Homepage of Java Reflection API. `http://java.sun.com/docs/books/tutorial/reflect/`.

[JVM] Homepage of Java Virtual Machine Tool Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.

[LY99] Tim Lindholm und Frank Yellin, Hrsg. *The Java Virtual Machine Specification*. Addison-Wesley, 2. Auflage, 1999.

[oE] AOSD-Europe Network of Excellence. Adoption of AOSD in Industry. `http://gateway.comp.lancs.ac.uk:8080/c/portal/layout?p_l_id=1.94`.

[SPE] Homepage of SPEC JVM98 Benchmark Suite. `http://www.spec.org/osg/jvm98/`.

Dr. Christoph Bockisch ist seit Januar 2009 Assistenzprofessor an der Universität von Twente, Niederlande. Im Juli 2008 hat er seinen Doktorgrad „mit Auszeichnung“ von der Technischen Universität Darmstadt erhalten, wo er seit 2003 als wissenschaftlicher Mitarbeiter in der Gruppe von Prof. Dr. Mezini tätig war. Seine Dissertation wurde 2009 vom European Network of Excellence on Aspect-Oriented Software Development als signifikant und fortschrittlich akkreditiert. Christoph Bockisch ist Autor und Co-Autor zahlreicher Veröffentlichungen, unter anderem in den Proceedings der Konferenzen OOPSLA, ECOOP, AOSD und VEE. Er ist mit-Begründer und Co-Organisator des Workshops „Virtual Machines and Intermediate Representations“, der bereits 2007 (AOSD) und 2008 (OOPSLA) sehr erfolgreich abgehalten wurde. 2009 und 2010 ist er Program Co-Chair der European Summer School on AOSD. Im Rahmen des Internationalen Masterprogramms „EMOOSE“ hat er 2006 und 2007 eine Vorlesung in Nantes, Frankreich gegeben.