Statische Analyse von Bibliotheken als Grundlage dynamischer Optimierung

Michael Thies und Uwe Kastens
Universität-GH Paderborn, Fachbereich 17, Fürstenallee 11
D-33102 Paderborn, Germany
{mthies,uwe}@uni-paderborn.de

Inhaltsangabe Dieser Beitrag schlägt einen neuartigen Ansatz zur optimierten Ausführung von Java-Bytecode vor, der die dynamische Optimierung eines Java-Programms durch statische Programmanalyse vorbereitet. Die Analyseinformation wird unabhängig von der Programmausführung bezogen auf alle Klassendateien einer Softwarebibliothek ermittelt und gespeichert. Zur Laufzeit unterstützt die an den Bibliotheksschnittstellen komponierte Information insbesondere Optimierungen, die zusätzlich dynamische Programmeigenschaften ausnutzen.

1 Einleitung

Einer der Hauptkritikpunkte an existierenden Java-Programmen ist die geringe Ausführungsgeschwindigkeit, verglichen mit äquivalenter Software, die in C++ geschrieben wurde. Soll nun die Ausführung durch Optimierungen beschleunigt werden, muß man den besonderen Spracheigenschaften von Java Rechnung tragen. Dazu gehört der Java-Bytecode als plattformunabhängige Programmrepräsentation und die Möglichkeit, Klassen erst dynamisch zur Laufzeit zu laden.

In diesem Beitrag möchte ich einen neuartigen Ansatz zur optimierten Ausführung von Java-Bytecode vorschlagen, der Optimierungen dynamisch zur Laufzeit eines Programms vornimmt, diese aber durch eine statische Programmanalyse vorbereitet. Die Analyse betrachtet jeweils eine komplette Softwarebibliothek als Einheit, und versiegelt diese gegen nachträgliche Veränderungen, um die Gültigkeit der gewonnenen globalen Information zu gewährleisten. Als Ausgangspunkt dienen die als Übersetzungsergebnis entstandenen Klassendateien, die mit den ebenfalls plattformunabhängigen Analyseergebnissen annotiert werden. Ein spezielles Laufzeitsystem setzt die Informationen beim Laden der Klassendateien zusammen, bereitet auf dieser Grundlage Programmbeobachtung vor und führt Optimierungen durch.

Da die Analyse vor der Laufzeit stattfindet und ihre Ergebnisse über mehrere Programmläufe hinweg gespeichert bleiben, können auch aufwendigere Probleme wie Typinferenz oder Aliasanalyse sinnvoll gelöst werden. Die Analyse verfolgt vier Ziele:

1. Bytecode statisch optimieren
2. überflüssige Laufzeittests erkennen, z.B. Indexüberprüfungen
3. aussichtsreiche dynamische Optimierungsstellen identifizieren
4. erforderliche Stellen für die Programmbeobachtung markieren

Gegenüber Ansätzen, die vollständig auf Programmbeobachtung basieren, liefert diese gemischte Form der Analyse stärkere Aussagen. Das Wissen über gemeinsame Eigenschaften aller Implementierungen einer Methode erlaubt beispielsweise Optimierungen auch an echt polymorphen Aufrufstellen.

Dadurch, daß die Untersuchung im Kontext einer Softwarebibliothek erfolgt, können die Ergebnisse zusammen mit der Bibliothek wiederverwendet werden. Bei einer Aktualisierung der Bibliothek genügt es, diese einmal neu zu analysieren, um alle darauf aufbauenden Programme zukünftig gemäß der neuen Implementierung dynamisch zu optimieren. Damit eignet sich diese Technik besonders in solchen Einsatzgebieten, wo nicht komplette Programme, sondern in Bibliotheken gekapselte Softwarekomponenten auf heterogenen Zielmaschinen wiederverwendet werden. Dies sind zugleich die Bereiche in denen statisch auf Programmebene optimierte und in Maschinencode übersetzte Sprachen, wie C++, vorteilhaft durch Java abgelöst werden könnten.

Im nächsten Abschnitt möchte ich auf den Stand der Technik bei der gegenüber einem einfachen Interpretierer optimierten Ausführung von Bytecode eingehen und die Grenzen der eingesetzten Methoden aufzeigen, sowie meinen Vorschlag von ähnlichen Ansätzen abgrenzen. Der dritte Teil stellt dann das Konzept der statischen Analyse vor der Programmausführung im Kontext einer einzelnen Klasse vor. Abschnitt vier motiviert die Ausweitung des Analysekontexts auf Bibliotheken. Schließlich gibt der fünfte Abschnitt eine Zusammenfassung und bietet einen Ausblick auf die praktische Umsetzung der vorgestellten Ideen.

2 Optimierte Ausführung von Java-Bytecode

Der plattformunabhängige Java-Bytecode ist auf der Zielmaschine normalerweise nicht direkt ausführbar, sondern muß von einer virtuellen Maschine (JVM) interpretiert oder in maschinenspezifischen Binärcode transformiert werden. Dieser Abschnitt stellt Konzepte von Java-Laufzeitumgebungen vor, die über einfache Interpretierer hinausgehen. Durch den Einsatz von JIT-Übersetzern (Just In Time-Übersetzer) erschließen sich hier mit der dynamischen und der optimistischen Optimierung auch spezielle Methoden zur Programmverbesserung. Ein Überblick einiger Ansätze zur Optimierung von Bytecode rundet das Bild ab.

2.1 Interpretierer und JIT-Übersetzer

Bei der effizienten Ausführung von Java-Bytecode auf einer konkreten Zielmaschine sind typische JVMs als Hybridsysteme realisiert, die einen klassischen Interpretierer mit einem JIT-Übersetzer kombinieren. Ein JIT-Übersetzer transformiert den Java-Bytecode einer Methode in eine äquivalente Folge von auf der Zielmaschine direkt ausführbaren Maschinenbefehlen, so daß bei mehrfacher Ausführung der Methode gegenüber einem Interpretierer das wiederholte Holen und Dekodieren der Bytecodebefehle entfällt. Allerdings benötigt die Umsetzung des Bytecodes in Maschinenbefehle zusätzlichen Laufzeitaufwand, der sich erst durch wiederholte Ausführung der übersetzten

Methode amortisiert. Grundsätzlich wird deshalb die Übersetzung feingranular auf der Ebene einer einzelnen Methodenimplementierung vorgenommen und sie erfolgt bedarfsgesteuert so spät wie möglich, nämlich erst unmittelbar vor dem Aufruf der betroffenen Methode.

Einfache Systeme verzichten lediglich auf die JIT-Übersetzung von Methoden, die nur einmal (bzw. selten) ausgeführt werden, z.B. die statischen Methoden zur Initialisierung einer gerade geladenen Klasse. Anspruchsvollere Strategien wägen den geschätzten Zeitaufwand für die Übersetzung gegen den bisher durch Interpretation entgangenen Laufzeitgewinn ab [14].

Die durch JIT-Übersetzung erreichten Geschwindigkeitszuwächse bleiben bei realen Programmen oft hinter den Erwartungen zurück. Gegenüber einem Interpretierer entfällt lediglich das mehrfache Holen und Dekodieren der Bytecodebefehle. Ansonsten folgt der generierte Maschinencode sehr eng der Struktur des Bytecodes, allenfalls einfache Optimierungen auf Basisblockebene sind wegen der Zeitrestriktionen möglich.

Bei einem Interpretierer tritt die für die Ausführung einer Instruktion benötigte Zeit in den Hintergrund, während ein JIT-Übersetzer starke Abweichungen zwischen den verschiedenen Gruppen von Bytecodeinstruktionen produziert. Arithmetische Operationen lassen sich sehr effizient in Maschineninstruktionen umsetzen, Befehle zur Manipulation des Operandenstapels entfallen durch eine Abbildung auf die Register der Zielmaschine mitunter sogar völlig, aber das Instanziieren von Objekten oder der Aufruf einer Methode werden kaum beschleunigt [12]. Daher übertreffen die bei den eher imperativ angelegten Benchmarks von JIT-Übersetzern erzielten Ergebnisse die in der Praxis zu beobachtenden Verbesserungen bei weitem.

2.2 Dynamische Optimierung

Eine Möglichkeit, die Effektivität von JIT-Übersetzern zu verbessern, liegt im Einsatz von Optimierungen, die das dynamische Programmverhalten ausnutzen, anstatt pessimistische Annahmen zu machen. Die Implementierung der prototypbasierten, dynamisch typisierten, objekt-orientierten Programmiersprache SELF basiert auf zweistufiger JIT-Übersetzung [9]. Zunächst transformiert ein einfacher JIT-Übersetzer auszuführenden SELF-Bytecode möglichst schnell in maschinenspezifischen Binärcode recht geringer Qualität. Für häufig benutzte Programmteile steht ein zweiter, optimierender JIT-Übersetzer zur Verfügung, der mehr Zeit für die Übersetzung benötigt, aber, speziell durch Inlining, schnelleren Maschinencode generiert. Auch hier muß zwischen investierter Übersetzungszeit und zukünftigen Laufzeiteinsparungen abgewogen werden.

Als überraschend wirkungsvoll stellte sich dabei die einfache Strategie heraus, die optimierte JIT-Übersetzung einer Methodenimplementierung zu verzögern bis eine gewisse Anzahl Aufrufe innerhalb einer kurzen Zeitspanne erfolgt. Wichtiger als die genaue Festlegung der Strategieparameter zeigte sich die Wahl der zu optimierenden Methodenimplementierung. Oft weist die von der Strategie identifizierte Methode nur einen kleinen Rumpf auf, dessen Optimierung keine nennenswerten Einsparungsmöglichkeiten bietet. Statt dessen wird einer der dynamischen Vorgänger dieser Methode optimiert und der zu kleine Rumpf direkt dort integriert [9].

Das SELF-System stützt seine Optimierungen fast ausschließlich auf dynamische Analyse des Programmverhaltens, denn der Benutzer verändert beim Programmieren fortgesetzt inkrementell das laufende System. Außerdem ist die Sprache SELF dynamisch typisiert, wobei jedes Objekt potentiell einen eigenen Typ besitzt, und führt kompromißlos selbst elementare Rechenoperationen und den Kontrollfluß auf den Aufruf von Methoden zurück. Daraus resultiert ein sehr einfach strukturierter Bytecode mit nur acht verschiedenen Befehlen.

Im Gegensatz zu SELF bietet Java eine strenge Trennung zwischen Übersetzungszeit und Ausführungszeit eines Programms, obwohl im übersetzten Programm die Reflexion über definierte Klassenelemente möglich ist. Dazu kommt eine statische Typisierung, die zusammen mit einigen Laufzeitüberprüfungen Typsicherheit garantieren soll [6]. Außerdem repräsentiert der deutlich komplexere Java-Bytecode den Kontrollfluß in Form von Sprungbefehlen und kodiert viele Elementaroperationen auf den in Java vordefinierten Grundtypen direkt. Angesichts dieser Unterschiede dürfte die Wirksamkeit einer statischen Programmanalyse bei Java wesentlich höher als bei SELF liegen und im Hinblick auf dynamische Optimierungen verwertbare Informationen liefern.

2.3 Optimistische Optimierung

In der von Sun angekündigten, aber noch nicht verfügbaren, JVM HotSpot sollen die im Rahmen des SELF-Projekts entwickelten Techniken möglichst direkt auf Java übertragen werden [1], trotz der Verschiedenartigkeit der beiden Sprachen. Neben einem verbesserten Verfahren zur Speicherbereinigung liegt der Schwerpunkt von HotSpot bei der dynamischen Optimierung der Programmstellen mit den höchsten Laufzeitanteilen, den *hot spots*.

Dabei findet die Programmanalyse vollständig zur Laufzeit statt und auch die so gewonnenen Ergebnisse leben nur bis zum Ende eines Programmlaufs, so daß die Ausführung jedesmal erst nach einer Aufwärmphase beschleunigt wird, die der Beobachtung des dynamischen Programmverhaltens dient. Bei den Transformationen zielt das System auf das Inlining von Methodenrümpfen und den sich daraus ergebenden Folgeoptimierungen.

Während klassische Optimierung zur Übersetzungszeit pessimistische (konservative) Annahmen über das zu optimierende Programm machen muß, kann HotSpot auch Transformationen durchführen, die auf optimistischen Annahmen basieren. Hat die Programmbeobachtung ergeben, daß bei der Ausführung einer Methode *M* sehr oft das Prädikat *P* erfüllt ist, wird Code für eine entsprechend spezialisierte Variante der Methode *M* erzeugt. Für die selteneren Fälle, in denen das Prädikat nicht gilt, generiert das System bei Bedarf eine zweite Variante von *M*. Beide Varianten unterliegen separat den dynamischen Optimierungsmechanismen, das heißt in der Regel wird nur eine von beiden so häufig ausgeführt, daß eine JIT-Übersetzung bzw. optimierte JIT-Übersetzung erfolgt. Typische Beispiele für zur Spezialisierung benutzte Prädikate sind Aussagen über den aktuellen Laufzeittyp eines Methodenparameters, was Inlining im spezialisierten Rumpf von *M* ermöglicht, oder die Annahme, daß bei der Ausführung von *M* keine Ausnahme (*Exception*) auftritt. Letztere Bedingung kann aber normalerweise nicht im

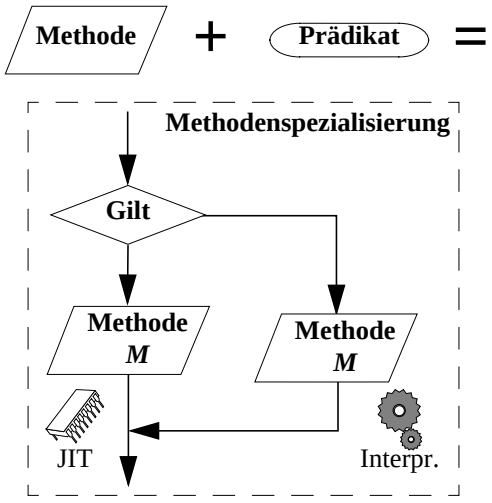

Abbildung 1: Prinzip der optimistischen Optimierung

voraus überprüft werden und erfordert daher Vorkehrungen zum Wiederaufsetzen der Methodenausführung, falls sich die Annahme als falsch erweist.

Methodenaufrufe, die dynamische Bindung ohne deutliche Bevorzugung einer bestimmten Methodenimplementierung ausnutzen, kann HotSpot nicht auf diese Weise optimieren. Damit entfallen auch die Folgeoptimierungen in der aufrufenden Methode, während der in diesem Beitrag vorgeschlagene Ansatz auch gemeinsame Eigenschaften aller Implementierungen einer Methode analysieren kann, z.B. das Fehlen von Seiteneffekten, und so Optimierungen über polymorphe Aufrufstellen hinweg ermöglicht. Außerdem könnte er durch statische Analyse bereits zu Beginn der Programmausführung bessere Entscheidungsgrundlagen für die in HotSpot implementierten Optimierungen verfügbar machen.

2.4 Verwandte Arbeiten

Eine Reihe von Ansätzen zur optimierten Ausführung von Java-Bytecode bzw. portablen Programmrepräsentationen finden sich in der Literatur. Clausen stellt in [5] einen klassischen Optimierer vor, der auf Bytecodes arbeitet, und Seiteneffekte von Methoden in verschiedenen Genauigkeitsstufen analysiert. Allerdings bezieht sich die Optimierung auf ein komplettes Anwendungsprogramm und jede Änderung oder Wiederverwendung von Klassen erfordert eine erneute Analyse. Natürlich sind nur solche Transformationen möglich, die sich auf Ebene des Bytecodes ausdrücken lassen. In ähnlicher Weise setzt auch Kennedy in [10] auf die statische Analyse vollständiger Programme und schließt das dynamische Laden von Klassen dadurch aus.

Cerniak und Li entwickelten zunächst einen Optimierer, der vor der Programmausführung Klassendateien für die Speicherhierarchie einer bestimmten Zielmaschine

transformiert [3]. Später haben sie die Optimierungen in einen JIT-Übersetzer integriert [4]. Das Problem des zu hohen Zeitbedarfs für die notwendige Analyse lösen sie jedoch, indem sie die zu lösenden Analyseaufgaben vereinfachen, und so die erzielbare Codequalität potentiell verringern.

Die Idee, Java-Bytecode vor der Programmausführung um zusätzliche Informationen zu erweitern, stellen Azevedo, Hummel, Kolson und Nicolau in [2] vor, zielen dabei jedoch nur auf innerhalb eines Methodenrumpfes lokal ermittelbare Informationen, die Registerzuteilung und Instruktionsanordnung in einem JIT-Übersetzer unterstützen oder überflüssige Laufzeittests eliminieren. Eine Analyse und Optimierung größerer Kontexte findet nicht statt.

Schließlich unterstreicht Franz in [7] die Vorteile einer auf abstrakten Syntaxbäumen basierenden Programmrepräsentation, weil sie einige Programmanalysen gegenüber Java-Bytecode erleichtert. Darauf aufbauend skizziert Kistler in [11] die Möglichkeiten der modulübergreifenden dynamischen Optimierung. Genau wie im SELF-System soll die gesamte Analyse zur Laufzeit des Programms erfolgen, eine statische Untersuchung vorab ist auch für die komplexeren Analyseaufgaben nicht vorgesehen.

3 Trennung von Analyse und Optimierung

Führt eine JVM die für eine dynamische Optimierung notwendige Analyse ebenfalls zur Laufzeit des Programms durch, müssen die erzielten Einsparungen zunächst den Analyseaufwand kompensieren bevor sich eine positive Gesamtwirkung einstellt. Besonders diejenigen Untersuchungen, die zu keiner Codeverbesserung führen, beeinflussen die Bilanz negativ. Aufwendigere Analysen verbieten sich automatisch durch ihr ungünstiges Kosten/Nutzen-Verhältnis, da ihre Ergebnisse nur einen einzigen Programmlauf lang leben. Darüber hinaus erfordern die Entscheidungen über vom dynamischen Programmverhalten abhängige optimistische Optimierungen eine längere Beobachtung der in Frage kommenden Programmstellen. Sie führen zu einer Aufwärmphase des Systems in der das Programm spürbar langsamer abläuft. Diese Aufwärmphase zu verkürzen und generell den notwendigen Analyseaufwand zur Laufzeit zu verringern, sind wichtige Schritte zur Steigerung der Ablaufgeschwindigkeit von Java-Programmen.

3.1 Aufgaben der Analyse

Erreichen lassen sich die angesprochenen Ziele durch eine zusätzliche Analysephase, die im Anschluß an die Übersetzung, also vor der Laufzeit stattfindet, und ihre Ergebnisse in der Klassendatei abspeichert. Eine Aufgabe der Analysephase besteht darin, konkrete Anwendungsstellen für statische Optimierungen zu bestimmen, wobei der Bytecode entsprechend transformiert wird, sofern sich die Optimierung auf dieser Ebene ausdrücken läßt. Andernfalls erzeugt die Analysephase Zusatzinformationen in der Klassendatei, die dann von einer speziell erweiterten JVM bei der Interpretation oder JIT-Übersetzung des Bytecodes berücksichtigt werden. Liefert die statische Untersu-

chung beispielsweise die Information, daß bei einem bestimmten Reihungszugriff keine Indexüberprüfung erforderlich ist, annotiert sie diesen Zugriff entsprechend. Da kein Bytecodebefehl für Reihungszugriffe ohne Indexüberprüfung existiert, scheidet eine direkte Codetransformation aus.

Neben der eigenverantwortlichen Entscheidung über zulässige Programmverbesserungen sammelt die Analysephase auch Informationen, die die dynamische Optimierung vorbereiten. Da die Analyse in der momentan betrachteten Form auf jeweils eine Klasse beschränkt bleibt, müssen Transformationen, welche Aussagen über einen größeren Kontext benötigen, bis zum Laden und Binden der Klassen, also bis zur Laufzeit verschoben werden. Der JVM fällt dabei die Aufgabe zu, die in den Klassendateien gespeicherten lokalen Informationen beim Nachladen von Klassen zu komponieren. Dabei entstehen globale Aussagen, wie z.B. “alle Implementierungen der Methode `getSize()` hängen rein funktional vom Empfänger der Methode ab, und haben keine Seiteneffekte”. Auf dieser Grundlage könnte dann schleifeninvarianter Code verschoben oder gemeinsame Teilausdrücke zusammengefaßt werden.

Weil die JVM zur Laufzeit eines Java-Programms bei Bedarf immer wieder Klassen nachladen muß, ändert sich unter Umständen auch die durch Komposition entstandene Information. Können dabei bestimmte Aussagen nicht mehr aufrecht erhalten werden, muß die JVM unter dieser Annahme transformierten Bytecode oder entsprechend JIT-übersetzten Binärcode invalidieren. Bezogen auf das obige Beispiel träte dieser Fall ein, wenn eine neu nachgeladene Unterklasse die Methode `getSize()` mit globalen Seiteneffekten neu implementiert. Während das Laden zusätzlicher Klassen die über das Gesamtprogramm vorliegenden Informationen höchstens abschwächt, kann die Spei-

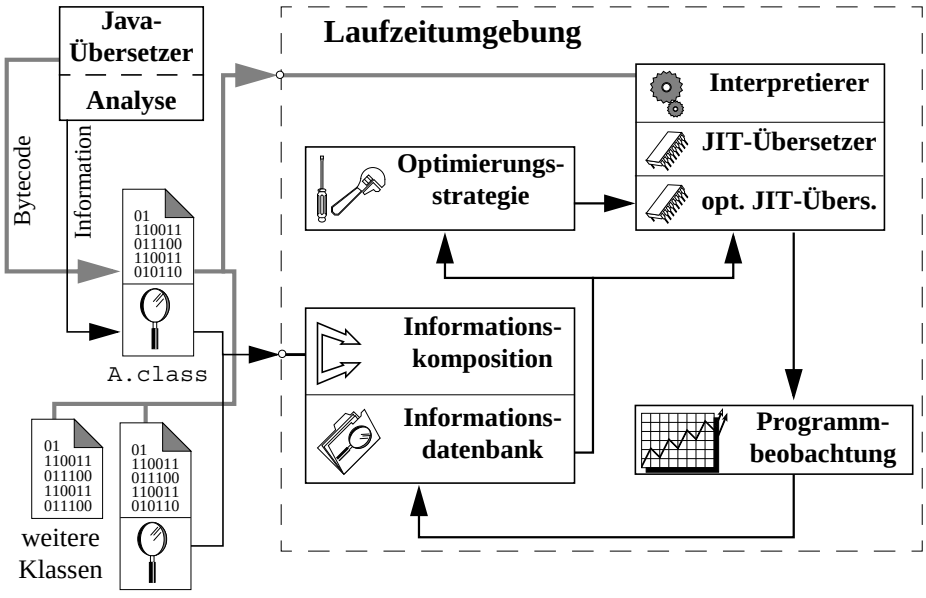

Abbildung 2: Laufzeitumgebung mit vorgeschalteter statischer Analyse

cherbereinigung nicht mehr benutzte Klassen entfernen, und so umgekehrt zu einer Verschärfung führen. Jedoch muß dieses Phänomen für eine korrekte Programmausführung nicht zwingend berücksichtigt werden und nur Spezialanwendungen entwikkeln eine entsprechend hohe Klassendynamik.

Auch bei der problematischen mit der Programmbeobachtung verbundenen Aufwärmphase kann eine vorgeschaltete statische Analyse helfen. Sie kann sowohl vielversprechende Programmstellen für eine solche Beobachtung auswählen, und damit den Gesamtaufwand für die Analyse zur Laufzeit reduzieren, als auch Hinweise für die bevorzugte JIT-Übersetzung bestimmter Methoden generieren, z.B. Methoden, die Schleifen mit statisch ermittelbarer hoher Durchlaufzahl enthalten. Bei der Reduzierung des Analyseaufwands wirken drei Effekte zusammen. Erstens kann eine als statische Analyse durchgeführte konkrete Typinferenz vor oder nach der Informationskomposition beim Laden von Klassen die gewünschten Ergebnisse ohne jede Programmbeobachtung liefern. Zweitens lassen sich potentielle Beobachtungsstellen ausschließen, weil dort echte, ausgewogene Polymorphie nachgewiesen werden kann. Drittens können die verbliebenen Programmstellen gemäß einer statischen Abschätzung der zu erwartenden Durchlaufhäufigkeit vorsortiert oder weiter eingeschränkt werden. Die ähnlich arbeitende Priorisierung von Methoden für die JIT-Übersetzung vermeidet den Nachteil vieler Übersetzungsstrategien, die nur häufig aufgerufene Methoden für signifikant halten, aber Methoden mit extrem hohem lokalen Laufzeitaufwand pro Aufruf benachteiligen.

3.2 Speicherung der Analyseresultate

Wie schon dargestellt schlagen sich die Ergebnisse der statischen Analyse nur zum Teil direkt in geändertem Bytecode nieder. Darüber hinaus muß der Bytecode bzw. die analysierte Klasse mit denjenigen Informationen annotiert werden, die erst die JVM zur Laufzeit weiterverarbeiten und auswerten kann.

Zur Speicherung beliebiger Zusatzinformationen über einzelne Elemente einer Klasse oder die Klasse als Ganzes unterstützt das Format von Java-Klassendateien benannte Attribute [13]. Eine Reihe von Attributen wird in der JVM Spezifikation definiert und dokumentiert, z.B. das Attribut ‘`Code`’, welches den Bytecode einer einzelnen Methode speichert. Zusätzlich auftretende Attribute mit unbekanntem Inhalt sind für auf Klassendateien operierende Anwendungen transparent, sofern sie nicht durch Änderungen an der Datei den Inhalt des Attributs invalidieren. Eine existierende Anwendung dieses Konzepts stellt die Speicherung von Debuginformationen in den beiden optionalen Methodenattributen ‘`LocalVariableTable`’ und ‘`LineNumberTable`’ dar.

Eine Reihe von Vorteilen sprechen für eine Speicherung der Analyseinformation in einem neuen, zusätzlichen Attribut. Die angereicherten Klassendateien können problemlos auf herkömmlichen virtuellen Maschinen ausgeführt werden, die die Zusatzinformationen nicht unterstützen. Ebenso bleibt es möglich, herkömmliche Klassendateien und solche mit abgespeicherten Analyseergebnissen zu mischen. Weiter entstehen bei der Neuübersetzung eines Java-Quellprogramms automatisch neue Klassendateien, denen dieses Attribut fehlt, so daß keine veralteten Resultate in der Datei verbleiben können.

Es gilt noch festzulegen, in welcher Form die Analyseinformation in einem dafür vorgesehenen Attribut innerhalb der Klassendatei gespeichert werden soll. Zwei Ziele beeinflussen diese Entwurfsentscheidung. Zum einen ist die Information möglichst kompakt zu speichern, damit sich die Übertragungszeiten für die angereicherten Klassendateien in Netzwerken nicht spürbar erhöhen. Zum anderen soll die JVM die gespeicherte Information zur Laufzeit schnell komponieren und effektiv einsetzen können.

Eine Möglichkeit gleichzeitig die zu speichernde Informationsmenge zu reduzieren und die Effektivität der gespeicherten Aussagen zu vergrößern besteht darin, präzise Informationen nur über vielversprechende Sachverhalte zu speichern. Beispielsweise könnten die Ergebnisse einer statischen, pessimistischen Aliasanalyse nur für Variablen mit bis zu drei potentiellen Aliasen gespeichert werden. Dies beschränkt die maximal je Variable zu speichernde Information und vermeidet gleichzeitig aufwendige Untersuchungen zur Laufzeit aus denen höchstwahrscheinlich sowieso keine Optimierungsmöglichkeiten resultieren. Analog könnte man bei der Untersuchung von Methoden auf Seiteneffekte auf die Behandlung von Rümpfen mit sehr vielen Abhängigkeiten von anderen Klassen verzichten und direkt ein negatives Ergebnis eintragen.

Da die Abspeicherung der Analyseergebnisse die Klassendateien vergrößert, lohnt sich ein Vergleich mit einer gelegentlich vorgeschlagenen Alternative: der Abspeicherung des vom JIT-Übersetzer erzeugten Codes in der Klassendatei. Diese Variante erscheint jedoch wenig attraktiv. In heterogenen Rechnerumgebungen profitiert entweder nur eine Plattform vom vorgehaltenen Binärcode oder mehrere Ausprägungen des JIT-Codes vergrößern die Klassendateien über alle Maßen. Dagegen ist die Analyseinformation auf jeder Plattform nutzbar und belegt weniger Platz als der Binärcode für auch nur eine einzige RISC-Plattform. Hinzu kommt, daß sich einfacher Maschinencode auf modernen Rechnern etwa so schnell im Speicher erzeugen wie von Datenträgern laden läßt [7]. Daß geladener Binärcode zusätzlich noch reloziert und gebunden werden muß, verschärft das Problem weiter. Bei den Resultaten der Analyse liegt der Aufwand für eine Neuberechnung deutlich höher, so daß eine permanente Speicherung über mehrere Programmläufe hinweg lohnt. Außerdem sind die Informationen langfristig wertvoller als vorberechneter JIT-Code, weil sie vom dynamischen Programmverhalten abhängige Optimierungen vorbereiten und unterstützen, während wiederverwendbarer Maschinencode nur auf pessimistischen Annahmen basieren kann. Der hier vorgestellte Ansatz, die Ergebnisse einer statischen Programmanalyse abzuspeichern harmoniert also deutlich besser mit der Plattformunabhängigkeit von Java-Bytecode und der Flexibilität der Java-Laufzeitumgebung.

3.3 Realisierung der Analysephase

Die Analyse soll in einer separaten, der Übersetzung nachgeschalteten Phase auf Bytecode arbeiten und die Klassendateien mit der gewonnenen Information anreichern. Gegenüber einer Integration in den Java-Übersetzer bietet die getrennte Analysephase mehrere Vorteile. Die Analysesoftware bleibt unabhängig vom verwendeten Java-Übersetzer und arbeitet auch mit Übersetzern für andere Programmiersprachen zusammen, die ebenfalls Java-Bytecode generieren. Einige Weiterentwicklungen der Quellsprache Java, wie die Einführung der inneren Klassen in Java 1.1, blieben auf der Ebene

der Klassendateien (fast) ohne Auswirkungen und entkoppeln so die Analyse ein wenig von der rasanten Entwicklung.

Kommerzielle Java-Programme und -Bibliotheken, die nicht als Quellcode, sondern nur als Klassendateien (evtl. in Java-Archivdateien verpackt) vorliegen, können ebenfalls analysiert und damit für eine Optimierung vorbereitet werden. Schließlich versetzt diese Lösung den Benutzer in die Lage, die Analyseinformation für zweifelhafte Klassendateien neu berechnen zu lassen, denn gefälschte Aussagen, etwa über die Unbedenklichkeit eines ungeprüften Reihungszugriffs, stellen eine Sicherheitslücke dar. Alternativ oder zusätzlich könnten digitale Unterschriften die Korrektheit der Analyseergebnisse garantieren. Da die semantische Lücke zwischen einem Java-Quelltext und der resultierenden Klassendatei recht klein ist, und fehlende Informationen problemlos durch einfache Analysetechniken gewonnen werden können, empfiehlt sich die in Abbildung 3 dargestellte, vom Übersetzer unabhängige Analysephase.

4 Versiegeln von Bibliotheken

Die bisher vorgestellte Analyse betrachtet jeweils nur isoliert eine einzelne Klasse und gelangt erst durch die beim Binden der Klassen erfolgende Informationskomposition zu Aussagen über größere Programmeinheiten. Natürlich wäre eine klassenübergreifende statische Analyse wünschenswert, da sie aus dem größeren Kontext genauere Informationen direkt gewinnen kann und somit zusätzliche potentielle Anwendungsstellen für Optimierungen aufdeckt. Dadurch würde der Analyse auch ein echter Vorteil gegenüber dem Java-Übersetzer verschafft, der auf eine einzelne Klasse bzw. eine Klasse mit den eingeschachtelten inneren Klassen beschränkt bleibt. Weil die Komposition der Analyseergebnisse zusammen mit dem ohnehin schon aufwendigen Laden und Binden der Klassen zur Laufzeit erfolgt, ist eine Vorauswahl vielversprechender Informationen erforderlich. Vergrößert man den Analysekontext verlagert sich ein Teil der Komposition in die vorgeschaltete Analysephase, das heißt der erforderliche Auf-

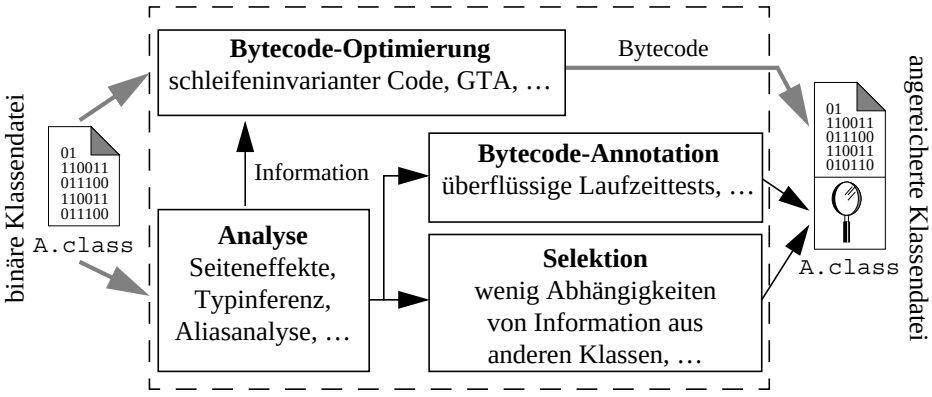

Abbildung 3: Realisierung der Analyse als separate Phase

wand zur Laufzeit wird reduziert und gleichzeitig die Qualität der Information verbessert, weil keine zwischengeschaltete Selektion potentiell verwertbare Ergebnisse verwirft.

4.1 Probleme der klassenübergeifenden Analyse

Als nicht praktikabel erweist sich der direkte Ansatz, aus der Analyse mehrerer Klassen gewonnene Ergebnisse an den relevanten Stellen innerhalb der einzelnen Klassendateien zu speichern. Es ist in Java ausdrücklich erlaubt, eine Klassendatei zwischen Übersetzung und Ausführung oder zwischen zwei Ausführungen eines Programms durch eine andere, binärkompatible Klassendatei zu ersetzen. Zwar gibt [8] keine konsistente formale Definition für Binärkompatibilität, meint damit aber offensichtlich für Nicht-Schnittstellenklassen eine aufwärtskompatible Erweiterung (oder Beibehaltung) der sichtbaren Klassenelemente. Diese Flexibilität führt dazu, daß eine JVM viele Überprüfungen wiederholen muß, die bereits der Java-Übersetzer vorgenommen hat, weil die zur Laufzeit benutzte Implementierung einer Klasse von der zur Übersetzungszeit herangezogenen Version verschieden sein kann.

Analog können natürlich einzelne Klassen zwischen der statischen Analyse und der Ausführung des Programms ausgetauscht werden, so daß veraltete, unzutreffende Informationen die dynamische Optimierung steuern. Das Beispiel in Abbildung 4 demonstriert dieses Phänomen anhand einer Analyse, die rein funktionale Methodenimplementierungen ohne Seiteneffekte als Vorbereitung für eine Optimierung gemeinsamer Teilausdrücke markiert. Obwohl der Austausch der Klasse `B` die in der Klasse `A` gespeicherte Analyseinformation eigentlich invalidiert hat, erfolgt die entsprechende Optimierung und bewirkt hier eine sichtbare Veränderung im Programmverhalten.

Eine ähnliche Problematik nimmt die Sprachdefinition für Java [8] zwar bei den Werten von Ganzzahlkonstanten (`static final int`) in Kauf, deren Werte direkt in die benutzenden Klassen propagiert werden dürfen, ohne daß bei Neufestlegung der Werte eine Aktualisierung gewährleistet ist, jedoch würde eine solche Nachlässigkeit bezogen auf die Analyseinformationen zu mannigfaltigen, subtilen Fehlerquellen und massiven Abweichungen vom Sprachstandard führen. Eine unkontrollierte globale Analyse mit verteilter Speicherung der so gewonnenen Ergebnisse scheidet daher aus.

4.2 Wahl des Analysekontexts

Es gilt also, einen möglichst großen Analysekontext zu finden, in dem der Verzicht auf die freie Austauschbarkeit von Klassenimplementierungen keine problematische Einschränkung darstellt, und der zugleich einen gewissen Schutz gegen unbeabsichtigte Eingriffe dieser Art bietet.

Eine einzelne Klasse bildet den kleinsten sinnvollen Rahmen für eine Analyse und erhält prinzipbedingt jede Klassendatei als unabhängige, in sich abgeschlossene Einheit. Allerdings ist dieser Kontext für eine effektive Analyse zu klein, sogar kleiner als der einem Java-Übersetzer verfügbare Programmausschnitt. Bei der Übersetzung bilden alle in einer Quelltextdatei definierten Klassen eine geschlossene Einheit. Da die

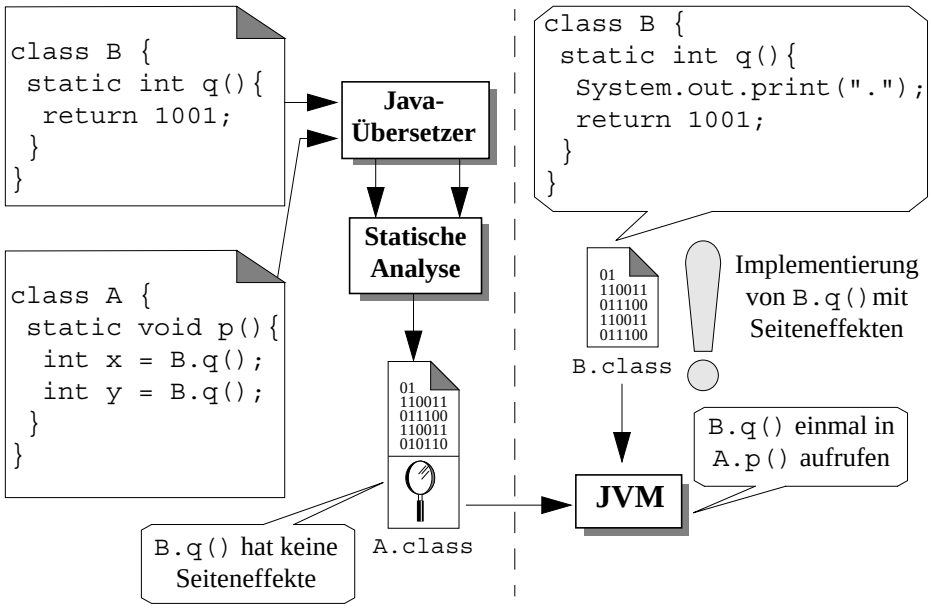

Abbildung 4: Austausch von Klassendateien zwischen Analyse und Ausführung

Klassendefinitionen innerhalb gewisser Grenzen jedoch vom Programmierer frei zu Quelltextdateien gruppiert werden können, bleibt eine Klasse nur mit den in ihr enthaltenen inneren Klassen garantiert zusammen.

Die gemeinsame Analyse aller lexikalisch geschachtelten Klassen eignet sich jedoch nicht, für ein System, welches sprach- und übersetzerunabhängig auf binären Klassendateien arbeitet, da das Konzept bereits auf dieser Ebene nur noch simuliert wird. Der Java-Übersetzer entschachtelt nämlich die inneren Klassen und benennt sie konsistent um (bzw. generiert Namen für anonyme Klassen). Vom Übersetzer erzeugte Zugriffsmethoden überbrücken Unterschiede in den Sichtbarkeitsregeln, die aus der Transformation in gleichberechtigt nebeneinanderliegende Klassen resultieren, z.B. für den Zugriff auf private Instanzvariablen der ehemals umgebenden Klasse aus einer inneren Klasse heraus.

Die nächstgrößere Struktur, die die Sprache Java unterstützt, ist das Paket (*Package*), das zugleich als Namensraum und als mögliche Sichtbarkeitsgrenze für deklarierte Programmobjekte fungiert. Auch wenn die Bildung qualifizierter Paketnamen gegenteiliges suggeriert, bilden Pakete in Java keine Hierarchie, sondern existieren allesamt gleichberechtigt nebeneinander. Berücksichtigt die Analyse eines Pakets restlos alle darin enthaltenen Klassen in genau den zur Laufzeit zu verwendenden Implementierungen, kann sie starke Aussagen über alle Klassen oder Klassenelemente mit Standardsichtbarkeit (kein Sichtbarkeitsmodifikator in der Deklaration) oder dazu äquivalenten Konstrukten (bei Klassen z.B. `protected final`) treffen. So kann die Analyse beispielsweise alle Verwendungsstellen einer entsprechend in der Sichtbarkeit eingeschränkten Klassenvariable ermitteln oder gemeinsame Eigenschaften aller Implemen-

tierungen einer nur im Paket sichtbaren Methode bestimmen. Wichtig für solche Allaussagen ist die garantierte Vollständigkeit der analysierten Klassen. Ob eine im Paket enthaltene Klasse `A` dagegen von einer anderen Klasse `B` statisch referenziert wird, das heißt im Quelltext der Implementierung von `B` tritt `A` als Typbezeichner auf, oder ob `A` dynamisch geladen wird, zum Beispiel durch einen Aufruf der Form `Class.forName("A")`, spielt für die Analyse nur eine untergeordnete Rolle. Lediglich am Programmentwurf orientierte Analyseprobleme, wie das Erkennen nicht benutzter (toter) Klassen, lassen sich in Gegenwart dynamisch geladener Klassen nicht mehr entscheiden.

Um die Korrektheit der für das gesamte Paket gewonnenen Analyseinformationen zu garantieren, muß das Paket im Rahmen der Analyse versiegelt werden. Nach Abschluß der Analyse dürfen also keine Veränderungen mehr am Paket erfolgen, insbesondere ist es unzulässig, zusätzliche Klassen in das Paket aufzunehmen oder Klassenimplementierungen auszutauschen. Viele typische Analyseresultate behalten zwar ihre Gültigkeit, wenn Klassen aus dem Paket entfernt werden, jedoch spricht die dann erforderliche Ausfilterung von Informationen über nicht mehr existierende Programmobjekte gegen eine Sonderbehandlung dieser speziellen Paketänderung. Als Konsequenz muß also vor jeder Veränderung an einem versiegelten Paket das Siegel unter Verlust der über das Paket gewonnenen Analyseinformationen erbrochen werden. Anschließend ist eine erneute Analyse des geänderten Pakets verbunden mit dem Anbringen eines neuen Siegels möglich.

Wegen seiner Funktion als Namensraum bildet ein Paket eine praktikable Einheit für diese Vorgehensweise. In sich abgeschlossene Softwarekomponenten werden in einem Paket verborgen, so daß nur die Anwenderschnittstelle außerhalb des Pakets sichtbar ist. Um Konflikte mit zukünftigen Versionen der Komponente zu vermeiden, dürfen Anwender weder Änderungen noch Ergänzungen innerhalb des Pakets vornehmen. Damit unterliegen separat erstellte und speziell zugekaufte Softwarekomponenten ohnehin bereits Einschränkungen wie sie die Analyse auf Paketebene und die dazu benötigte Versiegelung des Pakets mit sich bringt. Zusammen mit jeder Wiederverwendung der Komponente geht dann auch eine Wiederverwendung der Analyseinformation einher; die Einbettung in das umgebende Programm erledigt die an der Paketschnittstelle stattfindende Informationskomposition zur Laufzeit.

Den größtmöglichen für die Programmanalyse verfügbaren Kontext stellt das gesamte Programm dar, was aber wohl nur in Ausnahmefällen sinnvoll eingesetzt werden kann. Vorteilhaft ist die vollständige Information über alle Programmobjekte beliebiger Sichtbarkeitsstufe, nachteilig die zwangsweise damit verbundene Versiegelung des kompletten Programms einschließlich der benutzten Teile der Standardbibliothek. Dies macht eine Wiederverwendung der Analyseergebnisse unmöglich und verschlechtert das Verhältnis zwischen statischem Analyseaufwand und daraus resultierenden Laufzeiteinsparungen. Für Programme, die zur Laufzeit dynamisch um Plugin-Module oder Java Beans erweiterbar bleiben sollen, sind außerdem noch spezielle Vorkehrungen bei der Analyse zu treffen, da dann eine vollständige statische Untersuchung unmöglich wird. Rechtfertigt ein bestimmtes Anwendungsprogramm durch seine zu erwartende Ausführungshäufigkeit einen so hohen Analyseaufwand, sollte man über die Alternati-

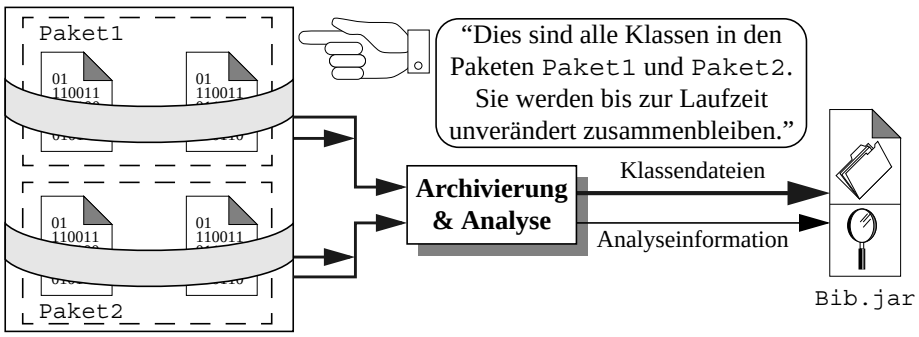

Abbildung 5: Analyse und Versiegelung einer Bibliothek

ve der statischen Vorübersetzung des Programms für eine bestimmte Zielmaschine nachdenken [15].

4.3 Bibliotheken als ideale Granularitätsebene

Nach den obigen Betrachtungen empfiehlt sich eine Analyse, die alle Klassen eines Pakets einbezieht als Kompromiß zwischen einem möglichst großen Analysekontext einerseits und dem daraus resultierenden Verzicht auf die freie Austauschbarkeit von Klassenimplementierungen andererseits. Die Annahme, daß jedes Paket isoliert von allen anderen Paketen eine unabhängige Softwarekomponente kapselt, erweist sich in der Praxis als zu streng. Bibliotheken, die Gruppen von relativ eng miteinander verbundenen Komponenten bereitstellen bilden die typische Einheit der Kapselung und Wiederverwendung. Auf die Sprache Java bezogen heißt das, eine Bibliothek besteht aus einer Menge von Paketen, wobei jedes Paket eventuell mehrere einander sehr ähnliche Komponenten realisiert, und verschiedene Pakete die nur lose gekoppelten Funktionsbereiche der Bibliothek voneinander trennen. Die Regeln für die freie Bildung hierarchisch qualifizierter Paketnamen basierend auf den für das Internet registrierten Domainnamen legt die Verwendung von Paketen sowohl zur internen Strukturierung von Bibliotheken als auch zur Abgrenzung von anderen Bibliotheken und vom benutzenden Anwendungsprogramm nahe.

Demzufolge sollte die statische Programmanalyse jeweils auf kompletten Bibliotheken, also Mengen von Paketen arbeiten und auch das Versiegeln gegen nachträgliche Veränderungen der analysierten Klassen bzw. Hinzufügen oder Entfernen von Klassen erfolgt mit dieser Granularität. Vorteile gegenüber der Analyse einzelner Pakete liegen in präziseren Informationen bei verringertem Kompositionsaufwand, wenn sich die in einer Bibliothek enthaltenen Pakete explizit gegenseitig benutzen. Daß eine bestimmte Bibliothek in der Regel von vielen Programmen gemeinsam genutzt wird, sichert eine häufige Wiederverwendung der bei der Bibliotheksanalyse gewonnenen Information und verbessert das Verhältnis zwischen Analyseaufwand und den daraus resultierenden Laufzeiteinsparungen entscheidend. Gleichzeitig unterstützt es die inkrementelle Wei-

terentwicklung von Software durch Austausch von kompletten Bibliotheken gegen neuere, aufwärtskompatible Versionen.

Das wohl deutlichste Beispiel für die Effektivität dieses Ansatzes bildet die Java-Standardbibliothek. Nachdem sie einmal analysiert wurde, profitiert jedes Java-Programm von den dynamischen Optimierungen, die interne Eigenschaften der verwendeten Bibliotheksimplementierung und die Art der Bibliotheksbenutzung durch das jeweilige Programm ausnutzen. Erscheint eine weiterentwickelte Version der Standardbibliothek, genügt ausschließlich eine erneute Analyse der Bibliothek, um alle Programme nicht nur weiterhin korrekt ablaufen zu lassen, sondern auch dynamisch auf die neue Bibliotheksimplementierung hin zu optimieren. Informationskomposition findet nur noch an den Schnittstellen zwischen aufeinander aufbauenden Bibliotheken und zum Anwendungsprogramm hin statt, so daß sich die Zahl der zu komponierenden Ebenen gegenüber der Länge einer Methodenaufrufkette reduziert. Zusätzlich unterliegt der Bibliothekscode, mit seinem oft beträchtlichen Anteil an der Gesamtlaufzeit eines Programms, vollständig der direkten statischen Analyse ohne zwischengeschaltete Informationsselektion, die sonst den Kompositionsaufwand verringern hilft.

Technisch realisierbar ist eine solche Bibliotheksanalyse auch auf Basis der Klassendateien, weil diese vollständige Angaben zur Paketzugehörigkeit von Klassen und zur Sichtbarkeit von Programmobjekten enthalten. Die Sichtbarkeitsattribute in Klassendateien orientieren sich direkt an den entsprechenden Modifikatoren der Sprache Java. Bezüge auf Klassen oder Klassenelemente enthalten die Klassendateien immer in einer normalisierten Form, nämlich als vollständig qualifizierte Namen, während die Quellsprache der Bequemlichkeit und Übersichtlichkeit halber auch abkürzende Schreibweisen über `import`-Klauseln erlaubt, aus denen aber Mehrdeutigkeiten oder Fehler bei späteren Ergänzungen in den importierten Paketen entstehen können.

Üblicherweise werden Java-Bibliotheken in Form von Java-Archivdateien (`jar`-Dateien) weitergegeben und verwendet. Das Zusammenfassen der zur Bibliothek gehörenden Klassendateien entspricht der Herstellung einer speziellen Bibliotheksdatei aus Objektcodemodulen in Sprachen wie C++. Die Integration der einzelnen Klassendateien in einer einzigen Archivdatei schützt zumindest vor dem unbeabsichtigten Austausch von Klassenimplementierungen oder ähnlichen Manipulationen an der Bibliothek. Durch weitere Konsistenzprüfungen läßt sich die Sicherheit noch erhöhen. Um die Analyse der Bibliothek und das Versiegeln der enthaltenen Pakete zu realisieren, ersetzt oder ergänzt man das zur Konstruktion der Archivdateien verwendete Werkzeug `jar`. Simultan führt das neue Werkzeug die statische Analyse auf Bibliotheksebene und den Aufbau der Archivdatei durch, so daß die Übereinstimmung zwischen archivierten Klassendateien und gespeicherter Analyseinformation garantiert ist.

Zwei Alternativen bestehen für die Speicherung der gewonnenen Analyseergebnisse in der Archivdatei. Entweder legt man die Resultate, wie für einzelne Klassen vorgeschlagen, direkt in zusätzlichen Attributen der archivierten Klassendateien ab oder man ergänzt das Archiv um eine zusätzliche Datei, die ausschließlich die Analyseinformation für die gesamte Bibliothek enthält. Aus der Informationsablage in den archivierten Klassendateien ergibt sich zwar, daß dieses Konzept unmittelbar mit außerhalb von Bibliotheken separat analysierten Klassen und solchen ohne Analyseinformation kombinierbar ist. Zerlegt aber jemand die Archivdatei in ihre Bestandteile, bleiben auf der Ba-

sis der gesamten Bibliothek gewonnene Ergebnisse den einzelnen Klassen zugeordnet, und verursachen die im Abschnitt 4.1 beschriebenen Probleme.

Umgekehrt erfordert die zentrale Informationsspeicherung in einer separaten Archivkomponente eine Sonderbehandlung für isoliert analysierte Klassen, zugleich werden aber bei Auflösung des Archivs die Analyseresultate von den Klassen getrennt und somit automatisch invalidiert. Das Bibliothekswerkzeug muß lediglich sicherstellen, daß niemals aus anderen Archiven extrahierte Informationsdateien einer Bibliothek hinzugefügt werden können, sondern nur aktuell ermittelte Analyseergebnisse. Eine entsprechende Vorgehensweise findet bereits bei der Ablage von Metainformationen über spezielle Java-Softwarekomponenten (*Beans*) Anwendung.

5 Zusammenfassung und Ausblick

In diesem Beitrag habe ich eine neue Vorgehensweise zur optimierten Ausführung von Java-Bytecode vorgeschlagen. Die Programmanalyse erfolgt dabei statisch vor der Programmausführung auf der Ebene kompletter Softwarebibliotheken, die zugleich gegen Änderungen versiegelt werden. Neben der statischen Optimierung bereitet die Analyse vor allem dynamische Optimierungen und die dazu notwendigen Programmbeobachtungen vor. Erst zur Laufzeit komponiert eine speziell erweiterte JVM die Untersuchungsergebnisse und wertet sie aus. Dadurch ist eine Wiederverwendung der Analyseinformation zusammen mit den analysierten Bibliotheken möglich und der inkrementelle Einsatz der Technik bei der Entwicklung und Wartung größerer Softwaresysteme wird praktikabel.

Im nächsten Schritt sollen durch Untersuchung existierender Softwarebibliotheken und Anwendungsprogramme geeignete Eigenschaften als Kriterien für die Analyse ausgewählt werden. Dann ist eine kompakte, effizient komponierbare Informationsrepräsentation festzulegen und die in Abschnitt 3.3 beschriebene Analysephase zu implementieren. Parallel dazu soll eine existierende JVM, beispielsweise `Kaffe` [16], um die auf den Analyseergebnissen aufbauenden dynamischen Optimierungen erweitert werden.

Referenzen

1. Eric Armstrong. HotSpot: A new breed of virtual machine. JavaWorld, März 1998.
2. A. Azevedo, J. Hummel, D. Kolson und A. Nicolau. Annotating the Java Bytecodes in Support of Optimization. ACM 1997 Workshop on Java for Science and Engineering Computation , Las Vegas, Nevada, Juni 1997.
3. Michal Cierniak und Wei Li. Optimizing Java bytecodes. In *Concurrency: Practice and Experience*, 9(6):427-444, Juni 1997.
4. Michal Cierniak und Wei Li. Just-in-time optimizations for high-performance Java programs. In *Concurrency: Practice and Experience*, 9(11):1063-1073, November 1997.
5. Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. In *Concurrency: Practice and Experience*, 9(11):1031-1045, November 1997.

6. Sophia Drossopoulou und und Susan Eisenbach. Java is Type Safe — Probably. 11th European Conference on Object Oriented Programming, Juni 1997.
7. Michael Franz. Java — Anmerkungen eines Wirth-Schülers. In *Informatik-Spektrum*, 21(1): 23-26, Februar 1998.
8. James Gosling, Bill Joy und Guy Steele. The Java Language Specification. In *The Java Series*. Addison-Wesley, 1996.
9. Urs Hölzle, David Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. ACM Transactions on Programming Languages, Juli 1996.
10. Ken Kennedy. An Environment for Compiling Java for High Performance on Servers. ACM 1997 Workshop on Java for Science and Engineering Computation, Las Vegas, Nevada, Juni 1997.
11. T. Kistler. Dynamic Runtime Optimization. Technical Report No. 96-54, Department of Information and Computer Science, University of California, Irvine, November 1996.
12. Christian Krusel. Portierung und Leistungsanalyse von Java-Laufzeitumgebungen auf MIPS-Architekturen. Diplomarbeit, Universität-Gesamthochschule Paderborn, Juni 1997.
13. Tim Lindholm und Frank Yellin. The Java Virtual Machine Specification. In *The Java Series*. Addison-Wesley, 1996.
14. Michael P. Plezbert und Ron K. Cytron. Does “Just in Time” = “Better Late than Never”? Symposium on Principles of Programming Languages, Paris, Januar 1997.
15. Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham und Scott A. Watterson. Toba: Java For Applications — a way ahead of time (WAT) compiler. COOTS '97, Juni 1997.
16. Tim Wilkinson. KAFFE — A free virtual machine to run Java™ code. Kommentierter Quelltext. `http://www.transvirtual.com`