Universität Passau
Fakultät für Informatik und Mathematik

JIT-Kompilationstechniken

im Hauptseminar „Multicore-Programmierung“

Sandra Kapfer
Passau, 1.12.2011

Inhaltsverzeichnis:

1.Motivation

Wie das Moore's Law besagt, sollten sich alle zwei Jahre die Anzahl der Transistoren verdoppeln. Doch wie die derzeitige Entwicklung zeigt, ist diesem Trend irgendwann eine Grenze gesetzt und eine Leistungsverbesserung ist nur noch über alternative Methoden möglich.
Eine Möglichkeit ist die Verwendung von Multicore-Prozessoren wie beispielsweise bei der aktuellen Intel Core-i-Serie, die bereits vorgestellt wurden.
Eine weitere Möglichkeit ist die Dynamisierung von Codeverarbeitungen. Zur Zeit ist es noch üblich, vor der Compilation eines Codes den „richtigen“ Compiler zu wählen, der dann möglichst die gewünschte Leistungsverbesserung bringt. Allerdings kann man dabei oft feststellen, dass für einige Codestellen ein anderer Compiler geeigneter gewesen wäre. Deshalb wäre eine dynamische Wahl der Compiler sehr hilfreich, um eine bessere Leistung zu erreichen. Bei der dynamischen Wahl wird vom Code selbst der geeignete Compiler gewählt. Hierbei könnten z. B. verschiedene JIT-Compiler verwendet werden. Diese JIT-Compiler sollen nun näher vorgestellt werden.

2.Geschichte [1]

Die JIT-Technologie ist aber keine neue Erfindung, wie man jetzt glauben könnte. JIT-Compilierung gibt es bereits seit der Frühzeit der Programmierung.
Eine der ersten Erwähnungen findet man in McCarthy's Paper von 1960, in dem er beschreibt, wie in der Sprache LISP die Funktionen in Maschinencode compiliert werden und das ohne einer Speicherung der Ausgabe des Compilers, weil dieser Vorgang schnell genug läuft, dass die Speicherung nicht notwendig ist.
Thompson fand einen Möglichkeit heraus, reguläre Ausdrücke von LISP in IBM 7094 Code in Laufzeit zu compilieren.
Ein weiterer Fortschritt gelang Mitchell 1970 mit seiner Beobachtung, dass compilierter Code vom Interpreter in Laufzeit interpretiert werden kann, wenn man die Aktionen des Interpreters speichert, wenn dieser das erste mal gelaufen ist.
Abrams beschreibt in seiner Dissertation ein Verfahren, in dem er versucht eine Codeoptimierung in APL durch zwei Methoden zu erreichen. Er arbeitet dabei mit einer Drag-along-Phase, in der die Auswertung von Ausdrücken so lange wie möglich aufschiebt, und einer Beating-Phase, in der versucht wird, die Datenmanipulation während der Auswertung möglichst gering zu halten. Die Drag-along-Phase ist heute auch unter „lazy Evalutation“ bekannt. Diese zwei Phasen kann man auch zur JIT-Compilation zählen, denn APL ist eine dynamische Sprache, weil die Typen und Attribute erst zur Laufzeit bekannt sind.
Mixed Code wurde 1973 von Dakin und Poole und von Dawson dazu verwendet, dass weniger Speicher benötigt wird, aber auch kein Geschwindigkeitsverlust entsteht. Mixed Code ist eine Mischung aus ursprünglichem Code und interpretiertem Code, wobei die häufig gebrauchten Codeteile in ursprünglichem Code vorhanden sind und die weniger häufigen Teile in interpretiertem Code. Allerdings waren die Anfänge nicht sehr erfolgreich. Erst durch weitere Verbesserungen konnte diese Methode zum gewünschtem Ziel führen. Beispielsweise lies Pittman 1987 die ursprünglichen Codezeilen als Maschinenbefehle unter den interpretierten Teil mischen. Da man aber dafür sowohl Interpreter als auch Compiler während der Laufzeit braucht, bekommt man auf heutigen Maschinen keine Platzersparnis.
Throw-Away-Compilierung, von Brown 1976 entwickelt, dient der Speicherplatzoptimierung. Hierbei werden bestimmte Teile dynamisch anstatt statisch compiliert, d.h. um Speicher zu sparen, wird der compilierte Code zu großen Teilen oder ganz weggeworfen und nur im Bedarfsfall regeneriert. Die Testumgebung für die Throw-Away-Compilierung war BASIC.
Hansen erarbeitete 1974 einex der ersten JIT-Systeme, lauffähig auf Fortran, in dem das Programm automatisch seine „Hot-Spots“ zur Laufzeit herausfindet. Dazu stellte er drei wichtige Fragen auf:
1.Welcher Code soll optimiert werden? Dies will Hansen mit seinem kostensparendem Modell durch einen Zähler lösen, der pro Code-Block mitzählt, wie oft dieser ausgeführt wird.
2.Wann soll der Code optimiert werden? Ist der vorher eingeführte Counter hoch genug, wird der Code dazu optimiert. Er verwendet dazu Mixed Code.
3.Wie soll der Code optimiert werden? Hansen optimiert seinen Code durch „constant Folding“, dann durch Teilausdruckelimination, als drittes durch „code motion“, usw.

In Smalltalk wurden durch Deutsch und Schiffmann JIT-Optimierungen vorgenommen. Dazu werden Prozeduren in „native code“ compiliert und bei deren Aufruf wird dieser Code vom Cache abgerufen. Werden diese nicht gebraucht, werden sie weggeworfen und gegebenenfalls regeneriert. Rau entwickelte diese Idee 1978 weiter, indem er den Code in einzelne „virtual machine“- Befehle umwandeln lässt.
Für C und ML wurde eine eigene Technik entwickelt, die sich „staged compilation“ nennt, bei der ein Programm in zwei Stufen aufgeteilt wird. Vor dem Ausführen des Programmes sorgt ein statischer Compiler für sogenannte „Templates“, die Blöcke für die anschließende dynamische Compilierung bilden. Der dynamische Compiler setzt diese dann zur Laufzeit zusammen und füllt die Platzhalter, die in den Templates gelassen wurden. Diese Technik kann allerdings erst zur JIT-Compilierung gezählt werden, wenn im statischen Bereich eine Form erzeugt wird, die Laufzeitübersetzung benötigt oder im dynamischen Bereich Laufzeitoptimierungen vornimmt.
Ein weiterer Ansatz ist die Simulation, wobei die binäre Übersetzung angewandt wird. Bei der dynamischen binären Übersetzung wird Maschinencode einer Maschine in den einer anderen während der Laufzeit überführt. Diese Technik ist sehr spezifisch für die einzelnen Maschinen. Es gibt laut May (1987) drei Generationen von Simulatoren. Die erste bilden Interpreter, die nacheinander jeden Ausgangsbefehl interpretiert. Die zweite bilden dynamisch übersetzte Befehle von der Ursprungs- zur Zielmaschinensprache und speichert diese für den späteren Gebrauch. Die dritte Generation übersetzt ganze Blöcke von Befehlen in die nächste Maschinensprache in einer bestimmten Zeit.
In Java gibt es mittlerweile sehr viele verschiedene Ansätze für die JIT-Compilierung, die später noch vorgestellt werden.

3. JIT-Compiler allgemein

„JIT-Kompilierung (Just In-Time compilation) ist eine Technik, die von Interpretern und Laufzeitumgebungen wie der Java Virtual Machine (JVM) zur Ausführung von Programmen verwendet wird. Im Gegensatz zu Compilern, die in einem Durchgang den gesamten Quelltext in ein ausführbares Programm umwandeln, wird der JIT Compiler hauptsächlich während der Programmausführung tätig. Er kompiliert dann nur die Programmteile, die tatsächlich ausgeführt werden sollen.“ [2]

Ein Ansatzpunkt der JIT-Compilation ist es, dass der größte Teil der Ausführungszeit für einen minimalen Teil des Codes verbraucht wird. Um dies zu vermeiden, gibt es bei unterschiedlichen JIT-Compilern mehrere Optimierungen, wie z. B. das Erkennen von „HotSpot“'s, also den sehr häufig aufgerufenen oder berechnungsintensiven Codeteilen.
Ein weiterer Ansatzpunkt, der zur JIT-Compilation gezählt werden kann, ist der Kompromiss zwischen Ausführungszeit und -platz. JIT-Compiler sollen sowohl Ausführungszeit als auch Speicherplatz sparen und einen Leistungsverlust vermeiden.

Für die folgende Erläuterung von einzelnen JIT-Compilern ist die Definition der SSA-Form notwendig:
„Eine spezielle Klasse von Zwischencode ist die *Static-Single-Assignment-Darstellung* . Sie zeichnet sich dadurch aus, dass im Zwischencode jeder Variablen *statisch* nur einmal zugewiesen wird. Dadurch werden Datenabhängigkeiten zwischen Befehlen explizit dargestellt, was für viele Optimierungen von Vorteil ist. Die Quellprogramme vieler Programmiersprachen lassen sich ohne größeren Aufwand in eine SSA-Darstellung transformieren. Viele moderne Compiler – darunter der GNU C Compiler – verwenden daher SSA-basierten Zwischencode.“ [3]

4. Bereits entwickelte JIT-Compiler

4. 1. Überblick

	Hotpath-VM	ShuJIT	RBC
Anwendungsgebiet:	Ressourcenbeschränkte Geräte	Einsparung von Kosten	Rechenintensive Codes
Methoden:	Schleifenidentifizierung, Trace-Aufzeichnung, Überführung in SSA-Form Trace-Zusammenführung	Stack-Caching, Instruction Folding, Exception Detection, Code Patching, Inlining	Regionsauswahl, das partitielle Inlining und das „Region exit handling“
Ziele:	Einhalten der Hardware-Begrenzungen ohne Leistungsverlust	1.„Einfach im Umgang 2.Kosteneffektive Entwicklung 3.Angemessene Qualität und Leistung für die Praxis.“	Leistungssteigerung

4.2. Hotpath-VM: JIT Compiler für Ressourcen-beschränkte Geräte [4]

Zuerst eine Definition von „Traces“: „Traces“ sind die von der HotpathVM gefundenen häufig aufgerufenen oder berechnungsintensiven Codeteile, die dann opitmiert werden.

Der Hotpath Compiler geht in sieben Schritten vor:
1. Identifizieren der Schleifenköpfe
2. Identifizieren von weiteren „Traces“
3. Umwandlung in SSA-Form
4. Anwendung von Datenflussanalysen
5. Lebensbereichsanalyse der Variablen
6. Codegenerierung nach den Optimierungen
7. Codeoptimierung durch Einsetzen eines zweiten Compilers

Im folgenden werden die einzelnen Schritte genauer erklärt:
Der erste Schritt ist das Identifizieren von Schleifenköpfen. Sieht man sich den Bytecode von einer Schleife an, findet man darin eine Jump-Stelle, in der das Programm zurück an den Anfang der Schleife springt, und diese erneut ausführt. An der Häufigkeit dieser Jumps kann man bestimmen, ob die Schleife als „Hotspot“ gilt. Als „Hotspot“ gilt sie dann, wenn sie häufig ausgeführt wird und somit viel Leistung braucht. Ist die Schleife nun als „Hotspot“ erkannt, kann viel Overhead vermieden werden, indem der Bytecode zwischen Schleifenbeginn und Jumpstelle als „Trace“ gespeichert.

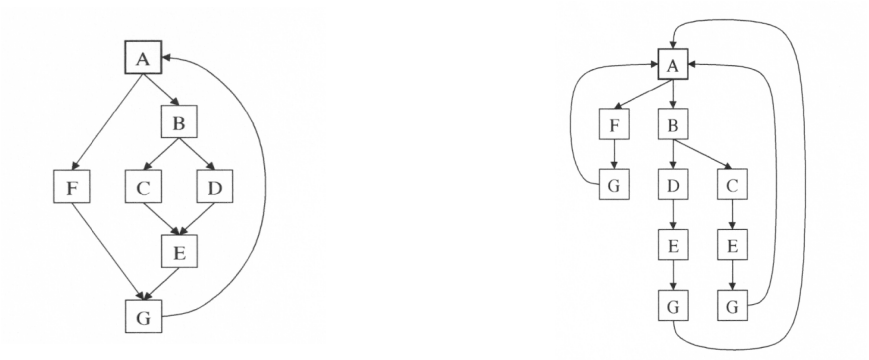
Der zweite Schritt versucht nun auch andere „Traces“ aufzuzeichnen, indem pro ausgeführtem Befehl ein sogenannter Metabefehl erzeugt wird, der dafür sorgt, dass der aktuelle Wert des Programmzählers, der Opcode des ausgeführten Befehls und den aktuellen Eintrag am Stack-Ende gespeichert wird. Während die „Traces“ aufgezeichnet werden, wird auch gespeichert, ob jeder einzelne Zweig gewählt wurde. Die gewählte Richtung wird dann im „Trace“ ein Schutzbefehl, der dafür sorgt, dass der Kontrollfluss des Orginals beibehalten wird. Mit diesen Schutzbefehlen kann man auch sicherstellen, dass die als nächstes aufgerufene Methode die ist, die auch vom Orginalcode aufgerufen werden würde, indem er die gespeicherte Richtung mit der aktuellen Richtung der Methode vergleicht. Wird bei diesen Tests ein Fehler festgestellt, wird wieder der Orginalcode ausgeführt und nicht die optimierte Version.
Die Aufzeichnung von „Traces“ wird beendet, falls:
–die Aufzeichnung abgebrochen wird,
–oder die Speicherung erfolgreich zu Ende kommt.
Mögliche Abbruchbedingungen für die „Trace“-aufzeichnung können das Werfen einer Fehlermeldung oder das Aufrufen einer Maschinencode- Methode sein. Zu einem erfolgreichem Ende kommt die Speicherung, wenn zum Schleifenbeginn zurückgesprungen wird.

Die aufgezeichneten „Traces“ werden nun in den folgenden drei Schritten direkt in Maschinencode umgewandelt.
Dazu wird der Stack in die SSA-Form überführt. Dabei beziehen sich die Operanden nicht auf den Stackspeicherort oder die lokalen Variablen, sondern auf die in den „Traces“ gespeicherten Befehle. Alle Variablen werden in einer „renaming“-Tabelle gespeichert, und dort abgeändert, sofern sie beim Schleifendurchlauf geändert werden. Sobald eine Änderung durchgeführt wird, wird diese Variable als Schleifenvariable gekennzeichnet, um am Ende der Schleife diese in Pseudobefehle umzuwandeln, die dann nur den aktuellen endgültigen Wert beinhalten. Alle anderen Variablen werden als „loop invariant“ gekennzeichnet.
Sobald nun die Umwandlung in SSA-Form vollendet ist, werden darauf eine Reihe von Datenfluss-Analysen durchgeführt. Dabei werden alle Befehle, die nur durch die als „loop invariant“ gekenntzeichneten Variablen operieren, sofort ausgeführt und dann aus der Schleife herausgezogen. Die Pseudobefehle werden dann bei jeder Iteration der Schleife dazu verwendet, dass die vorher berechneten Werte für den nächsten Durchlauf zur Verfügung stehen. Die verbleibenden Befehle werden dann in kleinere Befehle unterteilt, die dadurch, dass sie entweder wiederverwertet werden können oder dass sie elementare Befehle sind, zur

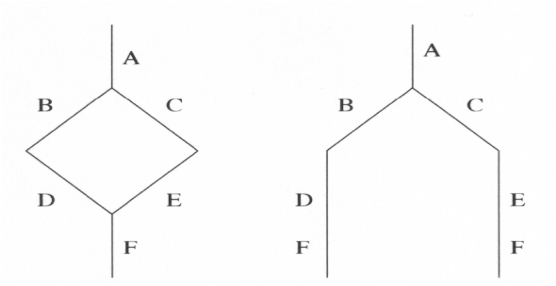
Optimierung beitragen.
Außerdem verwendet diese VM auch die übliche Teilausdruckelimination (CSE) und markiert redundante Befehle, die beim endgültigen Durchlauf nicht beachtet werden.

Bei der anschließenden Lebensbereichsanalyse wird jeweils von dem aktuellen Befehl ein Pointer auf den letzten Befehl erstellt und vor jedem Schleifendurchlauf überprüft, ob der aktuelle Wert mit dem vorher generierten übereinstimmt. Tritt dieser Fall ein, wird die Analyse beendet. Wird hingegen der endgültige Wert erst beim letzten Durchlauf der Schleife erstellt, werden sowohl der Endwert als auch der aktuelle Wert im selben Register gespeichert.

Anschließend wird beginnend beim letzten Befehl hin zum ersten der Code generiert und die Registerplätze werden bestimmt. Durch die Rückwärtsauswertung wird wie beim Maschinencode ein zweiter Durchlauf zur Zieladressenspeicherung eingespart.
Bei der Codegenerierung wird zuerst versucht, „constant Folding“ dahingehend durchzuführen, dass der Compiler ein Bit mitführt, das speichert, ob alle Operanden konstant sind und dann den der VM als korrekt annimmt, wenn alle konstant waren. Kann das „constant Folding“ bei einem Befehl nicht angewandt werden, so wird ein Register angelegt. In diesem Register wird dann der Wert gespeichert, der bei der Ausführung des für den Befehl benötigten Codeteils herauskommt.
Wird ein Schutzbefehl aufgerufen, werden alle Register-Werte in den aktuellen Speicher geladen und dann wird der Ersatzcode, der für jeden Seitenausgang einzeln erzeugt wurde, ausgeführt.
Der letzte Schritt bei der Codeoptimierung ist das Zusammenfügen von Pfaden, denn die vorher aufgeführten Optimierungen sind nur bei sehr regulärem Code effektiv. Da Code aber selten nur regulär ist, ist es nötig, innerhalb einer Schleife beispielsweise mehrere „Hot Paths“ zuzulassen. Dies geschieht, indem bei jedem Seitenausgang ein neuer Pfad begonnen und gespeichert wird.
In den folgenden beiden Graphiken ist dies gut zu erkennen:

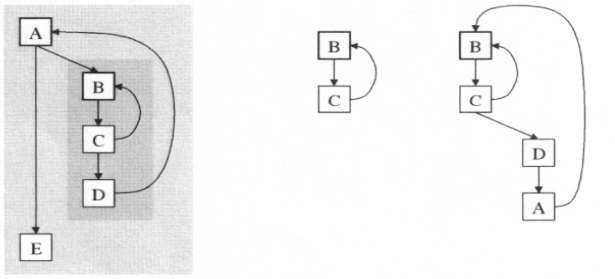


Da dieses Verfahren aber zu Codeverdoppelung führt, wird ein zweiter Compiler eingesetzt, der dann für die selben Befehle einen gemeinsamen Code erzeugt, der nur durch die vorhergehenden unterschiedlichen Ausgangsvariablen anders arbeitet. Im folgenden Beispiel wird aus dem vorher erzeugten Pfaden (rechts) der linke Graph erstellt.



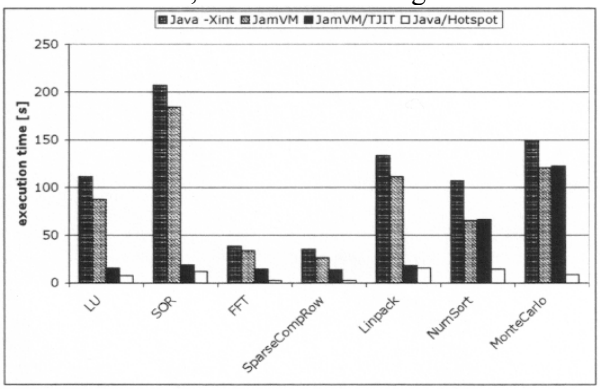
Innerhalb einer Schleife können wiederum Schleifen ausgeführt werden, wie zum Beispiel beim füllen einer Matrix. Da die innere Schleife häufiger ausgeführt wird als die äußere, wird die innere als erster Pfad gespeichert und dessen Header wird der einzige. Springt der Code

schließlich aus der inneren Schleife heraus zum nächsten Befehl wird ein Kindpfad erzeugt, der dann zurück zur äußeren Schleife springt, falls diese erneut die innere ausführen soll. Die folgende Graphik zeigt das beschriebene Vorgehen

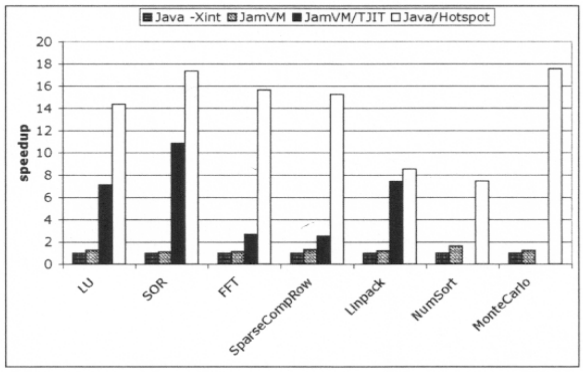


Der somit erzeugte Code ist nun vollständig für die Hotpath-Methode optimiert.

Experimentelle Vergleiche mit Benchmarkprogrammen von SciMark2 und Java Grande zeigen, dass die Ausführungszeit bei sehr regulären oder sequentiellem Code gut ist, als beispielsweise die Standard Java VM oder die HotspotVM. Bei nicht regulärem Code ist der Slowdown-Faktor 2. Für den MonteCarlo-Code und die numerische Sortierung ist die vorgestellte Hotpath-Methode gar nicht anwendbar, da dazu notwendige Methoden fehlen.

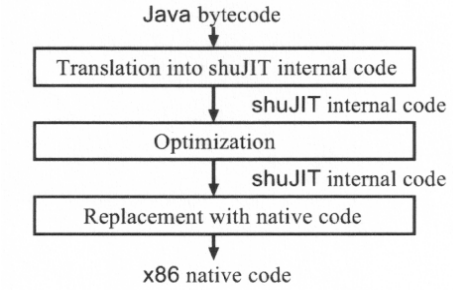


Auch bei der folgenden Speedup-Graphik kann man diese Nachteile erkennen.



4.3. ShuJIT-Compiler: Kosteneffektive Compilation [5]

Eine weitere JIT-Compilationstechnik wurde von Kazuyuki Shudo, Satoshi Sekiguchi und Yoichi Muraoka entwickelt.
Sie wollen mit ihrem Compiler im Gegensatz zum Hotspot-Compiler, der für berechnungsintensiven Code geschrieben wurde, mit ihrem Kosten einsparen, indem sie die folgenden drei Prinzipien verfolgen:
1.„Einfach im Umgang als Basis für Forschungen
2.Kosteneffektive Entwicklung, also wenig Arbeit und relativ viel Effekt
3.Angemessene Qualität und Leistung für die Praxis.“
Sie legen den Hauptaugenmerk auf Zeit- und Speicherersparnis und nicht auf Leistungssteigerung.
Dazu verwenden sie die Vorlagen-Technik, bei der der Compiler die unterschiedlichen Vorlagen von Maschinencode in Befehlsreihenfolge zusammenfügt.
In der folgenden Graphik sieht man das angewandte Verfahren:



Der Compiler wurde dazu in C programmiert, da die Entwickungskosten für Assemblercode relativ hoch sind. und kann auf x86-Maschinen ausgeführt werden. Die Templates sind in Assembler geschrieben, das von den x86-Maschinen ausgeführt werden kann und wurden vor dem Start compiliert. Dadurch braucht die Codegenerierung nur lineare Zeit und auch die Kalkulations- und Speicherkosten sind mit O(n) begrenzt, da ein solches in Assembler geschriebenes Template mit allen Befehlen lediglich 30 Kilobyte Speicher benötigt.
Um den ShuJIT umzusetzen, wurde im Compiler das Stack Caching eingeführt, das dafür sorgt, dass die Registerbenutzung verbessert wird. Dazu werden 5 Cache-Status definiert, die der Speicherung von Stacks entsprechen. Für einen internen Befehl werden somit 5 Templates benötigt, die dann bei der Generierung des Maschinencodes in Abhängigkeit des vorherigen Zustandes, das neue Template bestimmen. Ein Beispiel hierfür ist folgende Graphik:

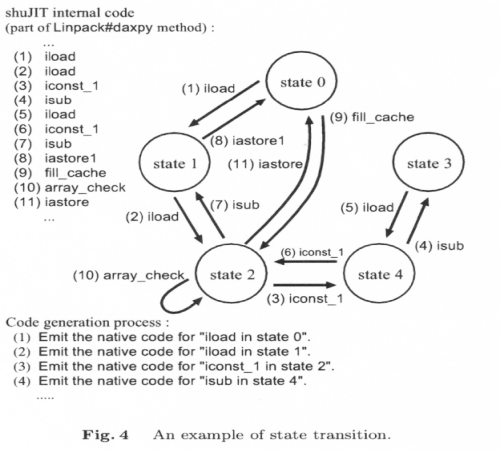


Fig. 4 An example of state transition.

Eine x86-Maschine besitzt lediglich 8 Allzweckregister, und hat somit begrenzt Platz, da einer davon für einen Stackpointer und ein anderer für einen Basispointer bereits belegt sind. ShuJIT belegt die übrigen Register mit zwei für das Stack caching, einen für die Speicherung der Basisadresse der JVM und die anderen 3 für die Arbeit der Templates.

Die Entwickler der ShuJIT verwenden zur Opimierung ihres Compilers den direkten Aufruf zwischen compilierten Methoden, die Fehleraufspüren mittels OS-Signalen und das Code

Patching. Diese Optimierungen laufen in höchstens O(n) Zeit und Speicher.

Im Folgenden werden die Optimierungstechniken genauer vorgestellt:

1. Direkter Aufruf zwischen compilierten Methoden
Hierbei kann eine compilierte Methode direkt eine andere aufrufen, ohne dabei einen Overhead zu erzeugen.
Dabei muss die aufgerufene Methode static, private oder final sein. Die aufgerufene Methode muss zudem immer compiliert werden, auch wenn sie dann nicht aufgerufen wird, da sonst ein Compilationsfehler auftritt. Dadurch entsteht gegebenenfalls verschwendete Zeit und umsonst verbrauchter Speicher, was aber durch den eingesetzten C Compiler nicht geschieht.
Kann eine Methode nicht compiliert werden, wird sie dennoch vom Interpreter ausgeführt, da dies besser ist, als ein Programmabbruch.

2. Instruction Folding
Dieses Verfahren ist eine bekannte Optimierungstechnik. Bei dieser werden mehrere einzelne Befehle zu einem zusammengefasst, wobei die ursprüngliche Befehlssemantik erhalten bleibt. Beispielsweise wird im Java Compiler javac ein Pop-Befehl mit einem folgenden Push-Befehl zusammengefügt, um nur einen Speicherzugriff zu benötigen. ShuJIT betreibt das Instruction Folding auf dessen internen Befehlen, um Speicherzugriffe zu sparen.
Hierbei werden z. B. bei mpegaudio ohne Instruction Folding 14.042 Mflops und mit werden 19.083 Mflops bei dem SPEC JVM98 Benchmark auf einem Pentium 4 ausgeführt.

3. Fehleraufspüren mit Signalen
ShuJIT erkennt mithilfe von OS-Signalen NullpointerExceptions, ArithmeticExceptions und StackOverflowError. Durch die Erkennung der Fehler kann die Zeit beim Ausführen eines fehlerauslösenden Codes gegenüber eines normalen Codes bei dem SPEC JVM98 Benchmark um einen Faktor von 90500 auf einem Pentium 4 reduziert werden.

4. Code Patching
Die Java Language Spezifikation schreibt strikt vor, dass eine Klasse dann initialisiert ist, wenn der statische Block ausgeführt ist und die statischen Variablen initialisiert sind. ShuJIT setzt diese Regel mittels Code Patching um.
Durch die Initialisierung einer Klasse erst beim Aufruf einer Methode dieser, würde dem JIT-Prinzip entsprechen, ist aber verboten und auch ineffizient, da bei mehrmaligem Aufruf der selben Methode die Klasse immer wieder neu geladen werden müsste.
ShuJIT löst dieses Problem, indem es Teile des compilierten Codes nach der ersten Ausführung überarbeitet, sodass er beim zweiten mal nicht mehr ausgeführt wird. So hat der Compiler keinen Leistungsverlust.

5. Inlining
Dieses Verfahren ist eine bekannte Technik, um die Anzahl und die Kosten eines Methodenaufrufs zu reduzieren und um ermöglicht mehr Möglichkeiten der intraprozeduralen Analyse und Optimierung.
Wenn eine Methode static, final oder private ist, bzw. wenn die aufgerufene Methode keine jump-Befehle, keine catch-Klauseln enthält und sie maximal 20 Instruktionen hat.
Außerdem verwendet ShuJIT eine spezielle Inlining-Methode für spezielle Methoden, wie z. B. für mathematische Funktionen, um die Ausführung der Methoden schneller zu machen und die Aufrufe von Maschinenmethoden zu reduzieren. Dabei werden diese speziellen Methoden als Templates vordefiniert und bereits in internen Befehlen gespeichert. Die Vorteile dieses speziellen Inlinings kann man in den folgenden Graphiken erkennen:

Table 1 Execution times to invoke the sqrt method 10,000,000 times.

	Pentium 4	Pentium III
Without special inlining	15535	34959
With special inlining	851	1567
Improvement ratio	× 18.3	× 22.3

(milliseconds)

Table 2 Execution times to invoke the sin method 10,000,000 times.

	Pentium 4	Pentium III
Without special inlining	4242	8021
With special inlining	1567	2226
Improvement ratio	× 2.71	× 3.60

(milliseconds)

Abschließend soll noch die Leistung dieses JIT-Compilers betrachtet werden. Hierzu kann man aus folgender Graphik erkennen, dass die Spitzenleistung im Vergleich zu anderen JIT-Compilern relativ gering ist. Dies ist darauf zurückzuführen, dass dem Leistungsgewinn weniger Beachtung geschenkt wurde.

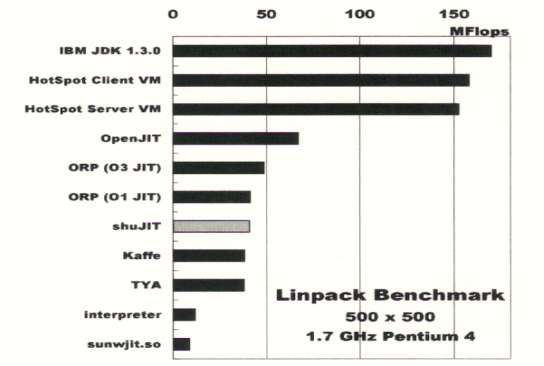


Fig. 12 The result of Linpack Benchmark for Java.

Allerdings kann man bei der nächsten Graphik erkennen, dass man durch die verwendeten Methoden viel Zeit beim Starten des Compilers im Vergleich zu anderen sparen kann.

Table 3 Startup time of Ichitaro Ark.

Initial number of counter	shuJIT	IBM JITC	HotSpot Client VM	HotSpot Server VM
0	4.4 *	22.1	N/A	N/A
3	4.0	14.1	5.9	32.1
1500	4.3	6.3	3.8 *	6.8
2000	4.4	5.7 *	3.7	6.6
10000	4.5	5.2	3.8	5.4 *
∞	3.9	5.0	4.0	4.1

(second)

∗ The default of the initial number

Table 4 Startup time of NetBeans.

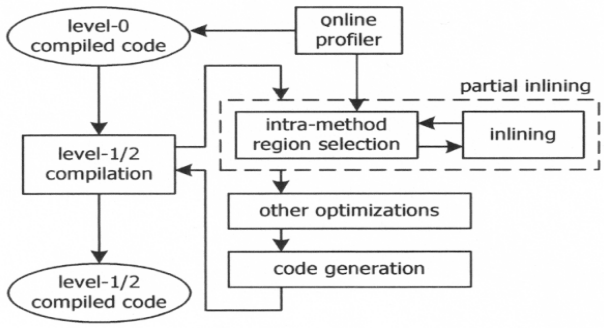
Initial number of counter	shuJIT	IBM JITC	HotSpot Client VM	HotSpot Server VM
0	8.9 *	43.3	N/A	N/A
3	8.4	28.6	12.0	166.6
1500	10.3	10.8	8.6 *	20.7
2000	10.7	10.6 *	8.5	20.9
10000	13.2	10.5	8.9	17.2 *
∞	12.2	14.0	13.0	13.4

(second)

∗ The default of the initial number

4.4. RBC: Region-based Compilation [6]

Toshio Suganuma, Toshiaki Yasue und Toshio Nakatani haben in ihrem Paper eine weitere JIT-Compilationstechnik, nämlich die Regionsbasierende Compilation, vorgestellt.
Im Folgendem soll erst ein Überblick über das genannte System gegeben werden, danach wird das Verfahren erläutert und schließlich werden noch Testergebnisse dargestellt.
1. Das System



Das System besteht aus einem „mixed mode“ Interpreter und drei Compilierungsniveaus, wobei der gesamte Code erst vom Interpreter ausgeführt wird und dabei die Aufrufanzahl und die Anzahl der Schleifendurchläufe gespeichert werden. Die damit festgestellten häufigen oder rechenintensiven Codepassagen werden dann im ersten Compilationsniveau behandelt.
In diesem werden dann nur grundlegende Kopien und konstante Vermehrung zur Optimierung bereitgestellt. Erst im nächsten Niveau, das die häufiger ausgeführten Codeteile erreichen, wird dann das Inlining und mehrere Datenflussoptimierungen angewandt. Im dritten Niveau, angewendet bei sehr häufig aufgerufenen Code, wird der Code dann noch mit den übrigen Optimierungstechniken, wie Abbruchanalyse oder Code-Scheduling, bearbeitet.

2. Region Exit Handling
In der aktuellen Version des vorgestellten Compilers wird an Stellen, an denen der Code eine Abbruchstelle erreicht, wird eine Neucompilierung mit dem selben Optimierungsniveau des ursprünglichen Codes aufgerufen, um keinen Leistungsverlust zu erhalten. Während dieser Neucompilation werden allerdings keine Optimierungen vorgenommen, damit keine Endlosschleife entsteht. Die folgende Graphik zeigt das beschriebene Vorgehen:

3. „Region-based Compilation“
Hierbei sehen sie die Regionsauswahl, das partitielle Inlining und das „Region exit handling“ als die wichtigsten Komponenten dafür an. In der obigen Graphik kann man erkennen, dass alle Optimierungen erst in den beiden höheren Niveaus ausgeführt werden.
Der Compiler startet seine Optimierung mit dem partitiellem Inlining. Dazu wird ausgehend von der Main-Methode ein Aufrufbaum erzeugt. Danach wird dieser durchlaufen und dabei überprüft, ob die aktuelle Methode sehr klein ist, denn dann wird diese sofort inlined. Ist sie eine größere Methode, wird sie mit Regionsauswahl, die später beschrieben wird, behandelt, um dadurch feststellen zu können, ob sie schließlich klein genug zum Inlinen ist. Dabei muss immer sichergestellt sein, dass alle benötigten Variablen von jeder Methode aufgerufen werden können.
Das partitielle Inlining hat den Vorteile für die beschriebene Compilationstechnik, dass durch das vorherige Löschen sämtlicher seltenen Teilen einer Methode kein unnötiges Inlining ausgeführt werden kann. Dadurch kann man die Budgetgrenzen einhalten und erhält keinen Leistungsverlust.

Im folgenden wird nun noch nachträglich die Regionsauswahl erklärt:
Unter der Annahme, dass jede Methode als Kontrollflussgraph mit nur einem Ein- und Ausgang gespeichert ist, beginnt der folgende Algorithmus damit, dass aufgrund von statischen Heuristiken und dynamischen Profilen einige grundlegende Methoden als selten oder häufig markiert werden.

Dabei werden folgende Heuristiken verfolgt:
–Ein vom Compiler erzeugter Backup Block ist selten.
–Ein Block, der eine Fehlermeldung wirft, ist selten.
–Ein Fehlerbehandlungsblock ist selten.
–Ein Block, der nicht initialiserte Klassenreferenz enthält ist selten.
–Nur Teile, die mit einem regulären Return enden, sind häufig.
Beim Vergleichen mit den dynamischen Profilen werden Teile, die nie ausgeführt wurden, als selten und Codepassagen, die öfter als eine vorher festgesetzte Zahl aufgerufen wurden, als häufig markiert.

In der Iterationsphase wird dann ausgehend von den Basisblöcken geschaut, welche Markierung die Vorgänger haben. Sind alle Vorgänger als häufig markiert, wird auch der Basisblock häufig. Kommt es zu einem Konflikt zwischen den Vorgängermarkierungen, wird die ursprüngliche übernommen.
In der letzten Phase werden alle Übergänge von häufigen zu seltenen Blöcken gelöscht und dafür „Exit Points“ gesetzt. Danach werden alle Variablen untersucht, ob sie noch benötigt werden und gegebenenfalls gelöscht. Anschließend wird ein neuer „Region Exit Block“ erzeugt, der dann an die Stelle des alten gesetzt wird und alle noch benötigten Variablen enthält.
Schließlich soll nun noch auf die Testergebnisse eingegangen werden. Hierbei beziehen sich die Ergebnisse auf eine Pentium 4 Xeon Rechner mit 2,8 GHz und 1024 MB Arbeitsspeicher.
Zum Verständnis der folgenden Tabelle müssen noch die vier Abkürzungen RBC-noopt, RBC-nopi, RBC-full und RBC-offline erklärt werden. Dabei werden bei der Ausführung mit RBC-noopt sämtliche Optimierungen abgeschaltet, die Ausführung mit RBC-nopi dient zur Darstellung der Effektivität des Inlining, die Ausführung mit RBC-full erlaubt alle Optimierungen und RBC-offline erlaubt auch alle Optimierungen, bezieht aber seine Ergebnisse offline.

Benchmarks		mtrt	jess	compress	db	mpeg	jack	javac	SPECjbb
Total number of methods executed		455	734	325	321	495	561	1,085	2,818
Level-0 compiled methods		200	334	135	126	219	373	825	566
Level-1 / level-2 compiled methods		33	41	7	7	56	81	157	166
Region-based optimized methods		28	22	1	6	19	58	101	123
Number of region exit points	Profile identified rare path	3	12	1	7	2	7	14	35
	Devirtualized backup path	483	36	0	19	16	136	644	1,244
	Exception throwing path	19	22	0	7	33	114	169	171
	Handler block	2	3	0	6	0	69	54	96
	Uninitialized code path	0	0	0	0	0	1	0	4
Recompiled methods due to region exit		0	0	0	0	1	9	4	1
Number of on-stack replacement		0	0	0	0	6	52	34	10

Table 1. Statistics of the region-based compilation for benchmark runs with RBC-full. The top three rows show execution and compilation statistics, the middle six rows are numbers of RBC optimized methods and how those rare region are identified, and the bottom two show runtime behavior for recompilations and on-stack replacements.

In Reihe 2 – 4 kann man erkennen, wie die Anzahl der behandelten Methoden während der einzelnen Niveaus abnimmt, z. B. bei mpeg von 219 auf nur noch 19 Methoden.
Die nächsten fünf Reihen schlüsseln die einzelnen Region Exit Points auf und die letzten beiden Reihen zeigen die aufgrund der Region Exits aufgerufenen Methoden zur Neucompilierung.
Dabei kann man erkennen, dass die Anzahl von level-1 oder level-2 compilierten Methoden in RBC 50-80% betragen.

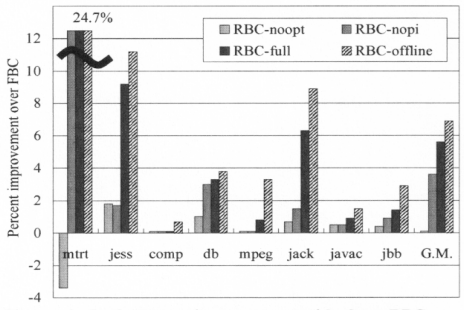


Figure 6. Performance improvement with four RBC approaches over the FBC. Taller bars show better scores.

In der obigen Graphik kann man den Leistungsgewinn der Region-based Compilation erkennen.

5. Literaturverzeichnis:

[1] Übersichtspapier über die Entwicklung 1960-2000: A brief history of Just-In-Time
http://portal.acm.org/citation.cfm?id=857077

[2] http://www.itwissen.info/definition/lexikon/JIT-Kompilierung-just-in-time-compilation.html

[3] http://de.wikipedia.org/wiki/Zwischencode

[4] HotpathVM: an effective JIT compiler for resource-constrained devices
http://portal.acm.org/citation.cfm?id=1134780

[5] Cost-effective compilation techniques for Java Just-in-Time compiler
http://portal.acm.org/citation.cfm?id=1060545

[6] A region-based compilation technique for a Java just-in-time compiler
http://portal.acm.org/citation.cfm?id=781166