Christian-Albrechts-Universität zu Kiel

Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

Prof. Dr. M. Hanus

JITC – Just-In-Time-Compilation

Seminar
Implementierung von Programmiersprachen

vorgelegt von

Ingo Strunk

betreut von

Dipl. Inf. Klaus Höppner

April 2004

1. Motivation

Heutzutage existieren eine Vielzahl unterschiedlicher Programmiersprachen, wie Assembler, Basic, Pascal, Java, uvw. Unabhängig, in welcher Programmiersprache ein Programm erstellt wurde, haben alle die Umsetzung von Quellprogramm in Maschinensprache gemeinsam. Im wesentlichen finden hier zwei Varianten Verwendung: Compilation und Interpretation. Bei der Compilation wird das Programm einmal auf einem Rechner statisch übersetzt und kann beliebig oft und sehr schnell auf der gleichen Systemumgebung ausgeführt werden. Dazu ist ein spezialisierter Compiler nötig, der aufwendig Implementiert werden muss. Im Gegensatz dazu, wird beim Interpretieren der Quellcode vor bzw. während jeder Programmausführung übersetzt. Dies ist in der Regel deutlich langsamer, jedoch vielseitiger einsetzbar, sofern der Interpreter für weitere Systemumgebungen vorhanden ist. Einen Interpreter für eine andere Rechnerarchitektur zu konstruieren ist allerdings einfach, so dass sich schnell viele Interpreter realisieren lassen. Da Quellcodes in abstrakteren Programmiersprachen deutlich kleiner sind als äquivalente compilierte Programme ergibt sich die Frage, auf welche Punkte mehr Wert gelegt werden soll: Ausführgeschwindigkeit oder Größe. Weiterhin ist die Möglichkeit des Debuggen bei einem Interpreter deutlich besser, da sofort die Stelle angezeigt werden kann, in der der Fehler passiert und auch die zugehörigen Speicherinhalte. Auch in Bezug auf Sicherheit ist es einfacher, einen abstrakten Code auf Manipulationen und ungewünschte Nebeneffekte zu überprüfen. Da viele Programme ihren Weg durch das Internet finden, lassen sich so bereits einige potentielle Sicherheitslücken schließen.

Wird die Interpretervariante bevorzugt, so wird versucht, das Interpretieren so geschickt durchzuführen, dass der Benutzer keine Interpretierpausen wahrnehmen kann, um in den Vorteil der Ausführgeschwindigkeit und der Programmgröße zu gelangen. Diese Übersetzung in Echtzeit wird Just-In-Time-Compilation (JITC) genannt.

2. Verfahren für JITC

Seit Anfang des Computerzeitalters in den frühen 60er Jahren existiert die Idee, sich selbst veränderbaren Code zu verwenden. Die Idee der JIT-Compilation ist zurückzuführen auf McCarthy im Jahre 1960 in einem Artikel zu LISP. Das Ziel war, den möglichst schnell erzeugten Code des Interpreters zu verwenden, anstatt ihn explizit abspeichern zu müssen, wie es ein Compiler machen würde. Um 1965 ist durch die Universität von Michigan festgehalten worden, dass das Laden des Quellprogramms und das Compilieren während des Ausführens durchführbar sei. Von diesem Zeitpunkt an wurden diverse Implementierungen einer JIT-Lösung für Programmiersprachen durchgeführt. Es wurde von Mitchell im Jahre 1970 festgehalten, dass die Ausgabe des Interpreters zur Laufzeit und das Resultat eines Compilers identisch ist. Dies wurde von Mitchell an

einer experimentelle Programmiersprache namens LC² herausgefunden. Dies legt nahe, Interpreter und Compiler innerhalb eines Programms zu nutzen, um die beide Vorteile, Geschwindigkeit und Größe, für sich nutzen zu können.

Obwohl es viele verschieden Programmiersprachen gibt, so lassen sich alle Ansätze in drei Kategorien einordnen [1]:

- Expliziter Aufruf des JIT-Compilers, z. B. HiPE in Erlang
- Ausführbarkeit auf unterschiedlichen Quell- und Zielarchitekturen
- Bei der Gleichläufigkeit zwischen JIT-Compiler und ausgeführtem Programm darf der Compiler dass ausgeführte Programm nicht ausbremsen.

2.1 Expliziter Aufruf

Die Technik, erst einen Interpreter und dann einen Compiler zu verwenden (siehe APL, 2.3 Gleichläufigkeit) wurde dahingehend verändert, dass nicht der gesamte Quellcode zunächst interpretiert und dann zusätzlich compiliert wird. Stattdessen wird vom Programmierer selbst angegeben, welcher Teil ausschließlich interpretiert und welcher compiliert wird. Dieser Programmierstil wird als „Mixed Code“ bezeichnet. Der Vorteil hierbei ist, dass oft verwendete Programmteil compiliert werden können, um schneller abzulaufen und andere interpretiert werden, um Speicherplatz zu sparen. Kritik hierzu ist die gleichzeitige Bereitstellung von Interpreter und Compiler, die sich die Systemressourcen teilen müssen. Bei Wegwerfcode, oder auch Throw-Away Code, dagegen ist es nicht wichtig, ob die Durchführung effizient ist. Diese Programme werden meist nur einmal ausgeführt und dann nicht mehr benötigt. BASIC beruht größtenteils auf dieser Technik.

Als Realisation einer „Mixed Code“ Programmiersprache ist hier Erlang zu nennen. Der für Erlang entwickelte JIT-Compiler namens HiPE, läuft nicht im Hintergrund ab. Hier kann der Benutzer während der Laufzeit entscheiden, ob jetzt compilierter Code ausgeführt werden soll, oder weiter interpretiert wird. Es ist also dem Anwender überlassen, wann er welche Variante bevorzugt. Als Beispiel:

Statt

```
1>  c(Module, Options)
```

jetzt

```
1>  c(Module, [native|Options])
```

2.2 Ausführbarkeit

Bei Fortran gab es eine Art dynamischer Optimierung der JIT-Compilation. Zunächst wird vom Interpreter selbst bestimmt, welche Teile des Quellprogramms interpretiert und welche compiliert werden sollen. Als nächstes wird bestimmt, wann diese Programmteile compiliert werden, z. B. eine Funktion erst nach einer bestimmten Anzahl von Aufrufen compilieren. Als letztes steht die Frage offen, wie stark der Compiler optimieren soll. Das bedeutet, dass eine Funktion erst eine bestimmte Anzahl von Aufrufen interpretiert wird, dann gering vom Compiler optimiert wird, und nach weiteren Aufrufen immer weiter optimiert wird. Durch diese dynamische Optimierung ist jedoch kein signifikanter Geschwindigkeitsvorteil gegenüber der statischen Compilierung zu beobachten. Jedoch sind Implementationen zur automatischen Beantwortung dieser Fragen durch ein Programm niemals effizient durchgeführt worden.

Für ML und C werden deshalb zwei Compilierungsschritte durchgeführt: Vor der Programmausführung werden statisch „Templates“ erzeugt. Während das Programm ausgeführt wird, werden die vorher erzeugten „Templates“ zusammengefügt und eventuelle Laufzeitvariablen eingesetzt. Wie genau die „Templates“ aussehen wird jedoch vom Programmierer bestimmt. Streng genommen findet hier allerdings keine „wirkliche“ JIT-Compilierung statt.

In der funktionalen Programmiersprache O‘Caml wird zunächst durch statische Compilierung ein Bytecode erzeugt. Während das Programm abläuft, werden Blöcke von Bytecode zunächst interpretiert und dann dynamisch in neue Blöcke, sogenannten „macro opcodes“, übersetzt und erst dann auf der Rechnerarchitektur ausgeführt.

Auch für Prolog wird eine dynamische Compilierung verwendet. Ein Compiler compiliert das Programm und die Blöcke werden zur Laufzeit dynamisch nachgeladen und ausgeführt.

Bei einer Simulation eines Systems hingegen, muss Maschinencode für eine Rechnerarchitektur auf einer anderen ausgeführt werden. Generell wird hier eine Übersetzung on-the-fly durchgeführt. Dazu gibt es drei Versionen: Jede Ursprungsinstruktion wird interpretiert und in der neuen Umgebung ausgeführt. In der zweiten Version wird jede einzelne Ursprungsinstruktion in die neue Umgebung compiliert. Die dritte Version compiliert Blöcke von mehreren Ursprungsinstruktionen. Um die Übersetzung in binäre Programme zu optimieren, werden Hardware-Lösungen untersucht. Dafür werden Chips mit optimiertem Instruktionssatz und breitem Spektrum verwendet. Ziel ist es, durch hochspezialisierte verschiedene Rechnerarchitekturen ein stark optimiertes ausführbares Programm zu erhalten.

2.3 Gleichläufigkeit

Für die Sprache APL wurde 1970 von Abrams während seiner Dissertation die „APL-Maschine“ entwickelt, die zunächst den Quellcode während des Ausführens in eine Postfix-Notation mit Hilfe von *lazy evaluation* interpretiert. Währenddessen läuft parallel dazu ein Compiler, der die Ausgabe des Interpreters aufnimmt, compiliert und zum nötigen Zeitpunkt ausführt. Allerdings war dies nur eine Idee von Abrams. Ein Versuch der Implementierung wurde drei Jahre später von Schroeder und Vaughn durchgeführt, aber nicht beendet. Die Technik selbst wurde jedoch durch andere aufgegriffen und als grundlegende JIT-Compilation weiter verwendet. Realisiert wurde sie dann 1977 in APL\ 3000 von Hewlett-Packard.

Ähnlich dazu wird bei Smalltalk der Quellcode in einer virtuellen Maschine compiliert, und zwar nur dann, wenn die Prozedur aufgerufen wird. Also wird hier eine Art *lazy evaluation* für das Compilieren verwendet.

Zu der Programmiersprache Self gab es drei Generationen. In der ersten Generation aus dem Jahr 1989 wurden die einzelnen Prozeduren bei Aufruf mit dem zugehörigen Kontext compiliert. Also eine ganz spezielle Compilierung, die jedoch selten wieder verwendet werden konnte. Also eine Art Wegwerfcode. Dennoch war diese Implementierung sehr effizient. Die zweite Generation produzierte ein deutlich schnelleres ausführbares Compilerergebniss, jedoch unter starken Einbußen der Compilierzeit (15 bis 35 fach langsamer). Daher wurde diese Version kaum genutzt. Die dritte Generation compiliert den Quellcode on-the-fly, wodurch allerdings für den Benutzer merkliche Compilierpausen auftreten.

Slim Binaries enthalten maschinenunabhängigen compilierten Code. Werden Teile von dem Programm ausgeführt, so werden sie on-the-fly in ausführbaren Code compiliert. Im speziellen Fall wurde diese Technik für Oberon-Module entwickelt. Allerdings ist kein Zeitunterschied zwischen dieser Version und dem Laden von Standard-Programmen zu bemerken. Weitere Entwicklungen brachten ein System hervor, welches sich während der Programmpausen ständig weiter compiliert mit immer stärkeren Optimierungen.

2.4 Java

Dieser Abschnitt ist nicht als Kategorie anzusehen, sondern beschreibt die Sprache selbst. Es lässt sich auch keine eindeutige Kategorie finden, da sie sich im Laufe der Entwicklung verändert hat.

Die ersten Java-Umgebungen waren reine Interpreter. Aufgrund von schlechten Ausführungszeiten wurde Java in statisch compilierten Bytecode für die Java-Virtual-Maschine (JVM) übersetzt. Obwohl Java erst 1997 mit JIT-Compilierung begonnen hat, wird dieser Begriff inzwischen fast ausschließlich durch Java geprägt. Dies hat sicherlich damit zu tun, dass Sun Microsystems viel Geld und Zeit investiert hat, um genauere Untersuchungen über Optimierungsgrad für JIT-Compilierung zu erhalten. Javas Bytecode wurde zu Beginn unter anderem in die Programmiersprache Self konvertiert, um die vorhandenen JIT-Compiler nutzen zu können. Letztendlich hat sich durchgesetzt, oft verwendete Codeelemente zu compilieren, während der Rest interpretiert wird. Dazu werden Interpreter und Compiler parallel ausgeführt. Hierbei kann der Interpreter den Bytecode auf ungewollte Manipulationen und ungewünschte Codeelemente überprüfen, z. B. Virus oder Trojaner. Dies ist ein erheblicher Sicherheitsaspeckt, gerade zu Zeiten des Internets.

Der Java JIT-Compiler, ab JVM 1.2, lädt zunächst nach dem Starten alle benötigten Klassen und die darin referenzierten Klassen ein. Danach werden diese Klassen gelinkt, im konkreten Fall werden Tabellen für die Sprünge zu den Methoden aufgebaut. Die Tabelleneinträge zeigen dann auf kurze Stub-Routinen, die entweder bei Bedarf den eigentlichen Compiler aufrufen oder im Fall, dass die Methode schon compiliert wurde, den compilierten Code direkt aufrufen.

3. Lohnt sich JITC ?

Da es den Begriff JIT-Compilierung schon seit mehreren Jahrzehnten gibt, kaum aber jemand etwas darüber kennt ist die Frage nach dem Sinn durchaus berechtigt. Da die Entwicklung schon lange andauert, kann es nicht als „Modeerscheinung“ bezeichnet werden. Ein Beispiel von [2] soll zeigen, ob JIT-Compilierung relevant werden kann. Hier wurde Bubble-Sort in drei Implementierungen getestet. Als Programmiersprache ist C++, Java 1.1.5 (reiner Interpreter) und Java 1.2 (JIT-Compiler) benutzt worden. „sort_ pointer“ sortiert ein Array mit Referenzen, „sort_object“ sortiert ein Array mit Werten und „sort_vector“ sortiert die Objekte in einem STL-Vektor. Für Java wurden äquivalente Programme wie für C++ geschrieben, wobei auf die gleiche Anzahl und Komplexität der Instruktionen geachtet wurde:

	C++	JVM 1.1.5 (Interpreter)	JVM 1.2 (JITC)
Sort_pointer	11,6	427	47
Sort_object	16,7	1804	343
Sort_vector	17,6	/	187

Die Ergebnisse sind in Sekunden, ausgeführt auf einer Sun Ultra 2/2170 mit Solaris 2.6 (SunOS 5.6)

Deutlich zu sehen, und nicht anders erwartet, schneidet C++ am besten ab. Das bemerkenswerte ist jedoch, dass durch die Verwendung des JIT-Compilers in der etwas neueren Version einen sehr großen Geschwindigkeitsvorteil von Faktor 9, bzw, Faktor 5, mit sich bringt. Also ist die Verwendung vom JIT-Compiler durchaus begründet. Zum Ausführen wird nur der relativ kleine Bytecode von Java benötigt und kann dann auf dem Zielsystem in der JVM ausgeführt werden. Gerade zu der Zeit des Internets und der weiten Verbreitung von Java, können die Programme weltweit transportiert werden. Daher ist ein kleines Programm durchaus sinnvoll. Weiterhin besteht die Frage nach der Sicherheit, gerade wenn die Daten durch das Internet transportiert werden. Doch auch hier ist ein zu interpretierender Bytecode sinnvoll. Durch Quersummenüberprüfung kann leicht festgestellt werden, ob die erhaltene Datei vom richtigen Absender kommt und im Internet nicht verändert wurde. Weiterhin ist es in dem abstrakten Bytecode möglich, die ungefähre Aufgabe des Programms heraus zu finden, und so für den Benutzer ungewollte Programmausführungen zu unterbinden, wie z. B. Dateimanipulation auf lokalen Datenträgern. Ist die Herkunft bestätigt, kann das Programm in der JVM interpretiert, werden, wobei auch auf den eingebauten JIT-Compiler zurückgegriffen werden kann. Nachteilig ist, dass es zur Zeit keinen JIT-Compiler gibt, der compilierten Java-Bytecode cachen kann, so dass dieser auch bei einem Neustart des JIT-Compilers zur Verfügung stehen würde.

Es ist also durchaus sinnvoll, eine JIT-Compilierung der ausschließlichen Interpretierung vorzuziehen. Zwar ist C++ deutlich schneller, aber es wurde ein spezieller Compiler dafür geschrieben. Auch ist die Plattformunabhängigkeit dadurch nicht mehr vorhanden. Des weiteren besteht die Möglichkeit, auch Java-Code statisch durch javac zu compilieren. Dann sind die Geschwindigkeitsvorteile gegenüber nicht mehr so groß. Weiterhin ist die Entscheidung nicht unbeeinflusst von der Art des Programms. Besitzt dass Programm lange Ausführzeiten wie z. B. bei einem Betriebssystem, so sollte ein hochoptimierter Compiler verwendet werden, und nicht eine JIT-Compilierung. Hierbei macht sich eine längere statische Compilierung bezahlt. Jedoch ist für Toolprogramme, die geschrieben wurden, um einmal ausgeführt zu werden, ein so großer Aufwand nicht nötig. Ebenso für Skripte aus dem Internet, die sich durch ihre Plattformunabhängigkeit auszeichnen und nicht durch die aussergewöhnlich schnelle Ausführungszeit. Daher verdient sich JIT-Compilierung einen Platz in den modernen Programmiersprachen.

4. Implementierungen von Java JITC

Obwohl im Bereich JIT-Compilation schon sehr lange geforscht wird, sind momentan alle Forschungen mit Java JIT beschäftigt. Aus diesem Grund werden nachfolgend ein paar Implementierungen für Java JIT-Compilierung präsentiert:

4.1 CACAO

CACAO war zu Anfang eine Diplomarbeit und wird an der Universität Wien weiter entwickelt. Diese Implementierung ist frei verfügbar und unter [3] zu erreichen. Das besondere hieran ist, dass dieser JIT-Compiler von dem Stack-Modell des Java Interpreters abweicht, und eine Implementierung eines Registermodells verwendet. Hierfür werden die existierenden Register auf der Zielarchitektur zur Ausführung mitverwendet, wodurch ausserordentlich hohe Ausführgeschwindigkeiten erreicht werden können.

4.2 KAFFE

KAFFE ist ebenso wie CACAO frei erhältlich [4]. Das besondere an diesem JIT-Compiler ist, dass ein Teil der kompletten Java 1.1 API unterstützt wird, und somit eine große Anzahl von Java-Programmen unter KAFFE lauffähig ist. KAFFE implementiert weiterhin ein komplettes Laufzeitsystem inklusive Unterstützung für das Java Native Interface.

4.3 TYA

Ganz ohne Laufzeitsystem kommt TYA aus. Diese Implementierung [5] linkt sich in den Methodenaufruf eines bestehenden Java Interpreters ein. Wird eine Methode aufgerufen, so tritt stattdessen TYA in Kraft. Ist die Methode noch nicht compiliert, wird dies nachgeholt, ansonsten ausgeführt. Auch diese Implementierung ist frei erhältlich und kann jederzeit ein- und ausgeschaltet werden.

5. Fazit

Just-In-Time-Compilation ist erst nach 30 Jahren durch Java allgemein bekannt geworden. Es sind diverse Ansätze im Laufe der Zeit realisiert worden, wobei sich eine virtuelle Maschinenumgebung durchgesetzt hat, in diesem Fall die JVM. Durch eine einfache Erweiterung, die auch ohne Kenntnis des Interpreter-Quellcodes durchführbar ist [5], ist eine JIT-Erweiterung der alten reinen Interpreter-Umgebung realisierbar. Hierdurch kann durchaus ein deutlich beschleunigter Programmablauf erzielt werden. Daher ist es durchaus sinnvoll, JIT-Compilierung zu verwenden, da es keine Nachteile gegenüber der Interpretation gibt. Der JIT-Compiler wird in der momentanen Version der JVM 1.4.1 weiterhin unterstützt. Da Java auf die JVM festgelegt ist, wird es in Zukunft wohl kaum revolutionäre Erneuerungen in diesem Bereich geben, wohl aber eine größere Fülle von unterstützten Befehlen für den JIT-Compiler. Weiterhin würde eine Methode zum

Cachen des compilierten Codes eine deutliche Verbesserung in der Geschwindigkeit ergeben, sofern das zugehörige Programm öfters auf dem gleichen System ausgeführt wird. Implementierungen hierzu sind noch nicht zu finden.

6. Literaturverzeichnis

[1] John Aycock,
„A Brief History of Just-In-Time“,
ACM Comput. Surv. 35(2), pp. 97-113, 2003.

[2] Ulrich Stern,
„Java vs. C++“,
http://verify.stanford.edu/uli/java_cpp.html, 2001.

[3] Andreas Krall,
„CACAO Home Page“,
http://www.complang.tuwien.ac.at/java/cacao/index.html, 1998.

[4] Kaffe.org,
„Welcome to Kaffe“,
http://www.kaffe.org/, 2004.

[5] Albrecht Kleine,
„TYA 1.8“,
http://www.sax.de/~adlibit/, 2000.