Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Optimierungen zur Compile-/Laufzeit

Seminar „Effiziente Programmierung“

vorgelegt von

Mats Schrader

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6837119
Betreuer:	Michael Kuhn

Hamburg, 28.04.2020

Abstract

Compiler übersetzen Code aus einer höheren Programmiersprache in ausführbaren Maschinencode. Wann und wie der Code kompiliert und optimiert wird, beeinflusst die Compilezeit und die Laufzeit vom resultierenden Maschinencode.

In dieser Ausarbeitung wird die Optimierung zur Compilezeit mit der Optimierung zur Laufzeit verglichen. Dazu werden Ahead-Of-time-Compiler (AOT-Compiler) und Just-in-time-Compiler (JIT-Compiler) betrachtet, die jeweils einen Ansatz repräsentieren.

AOT-Compiler trennen Compilezeit von der Laufzeit, Kompilieren und Ausführen sind zwei getrennte Vorgänge. Man kann hier viel Zeit in die Optimierung investieren, um anschließend einen Maschinencode mit geringer Laufzeit zu erhalten.

JIT-Compiler kompilieren den Code sobald er ausgeführt werden soll, Kompilieren und Ausführen sind für den Nutzer nicht zu unterscheiden. Das Kompilieren sorgt für einen langsameren Start des Programms, kann aber mehr und spezifischere Optimierungen durchführen, da Laufzeitinformationen verfügbar sind.

Contents

1 Einleitung **4**

2 Aufgabe **5**

3 Optimierung zur Laufzeit **6**

3.1 Polymorphic Inline Caching 6

3.2 Polymorphic Inline Caching Performance 8

3.3 Adaptive Optimierung 9

4 Nutzung von JITs **10**

5 Vergleich von AOT und JIT **12**

6 Zusammenfassung **14**

7 Quellen **15**

1 Einleitung

In der Einleitung sollen einige Begriffe und Konzepte erklärt werden, auf denen später aufgebaut wird.

Interpreter sind Programme, die Code in einer höheren Programmiersprache, wie z.B. Java, JavaScript, C etc. direkt ausführen können. Es wird also nichts extra kompiliert (von einer Programmiersprache in Maschinecode übersetzt) und es entsteht keine neue Datei.

Da Maschinencode schneller ausgeführt werden kann und beim Kompilieren in der Regel optimiert wird, ist die Laufzeit von Code mit einem Interpreter deutlich länger als von kompiliertem Code. Ein Vorteil von Interpretern ist, dass man sich den Vorgang des Kompilierens und dadurch auch Zeit spart.

Ahead-Of-time-Compiler sind "klassische" Compiler, wie sie z.B. meistens für C Code verwendet werden. AOT-Compiler kompilieren Quelldateien zu ausführbaren Programmen. AOT-Compiler können aufwändige Optimierungen durchführen, da das Kompilieren getrennt von der Ausführung ist, und erzeugen schnellen Maschinencode. Der Maschinencode ist hardwarespezifisch, sollte also auf jedem Computer neu kompiliert werden. AOT-Compiler haben keine Informationen über das Laufzeitverhalten des Codes, alle Optimierungen basieren auf einer statischen Analyse des Codes.

Just-in-time-Compiler kompilieren den Code zur Laufzeit. Nach außen wirken sie wie ein Interpreter, da Kompilieren und Ausführen in Einem geschieht und auch hier keine neue Datei erstellt wird. Im Gegensatz zu Interpretern wird der Code kompiliert und optimiert und es entsteht effizienter Maschinencode.

JIT-Compiler laufen am Anfang langsam, da der Code zunächst nur interpretiert wird, während parallel kompiliert und optimiert wird. Obwohl keine neue Datei entsteht, wird der Code intern in Bytecode oder eine andere "intermediate representation" übersetzt. Da zur Laufzeit kompiliert wird, können JIT-Compiler Laufzeitinformationen nutzen und wissen, auf welcher Hardware gearbeitet wird; dies ermöglicht spezifische Optimierung.

2 Aufgabe

Um die Optimierung zur Laufzeit und zur Compilezeit zu vergleichen, stelle ich zunächst Optimierungen vor, die nur zur Laufzeit ausgeführt werden können. Optimierungen zur Compilezeit setzte ich als bekannt voraus und werde ich nicht weiter erläutern. Des Weiteren gehe ich auf Anwendungen von JIT-Compilern ein.

3 Optimierung zur Laufzeit

In diesem Kapitel werden zwei Beispiele für Optimierungen zur Laufzeit erklärt.

3.1 Polymorphic Inline Caching

Inlining ist ein Verfahren beim Optimieren von Code, bei dem Funktionsaufrufe durch den Code der Funktion ersetzt werden. Dadurch wird der Code länger, aber man spart sich einen Sprung und damit Laufzeit, da Sprünge relativ langsam sind.

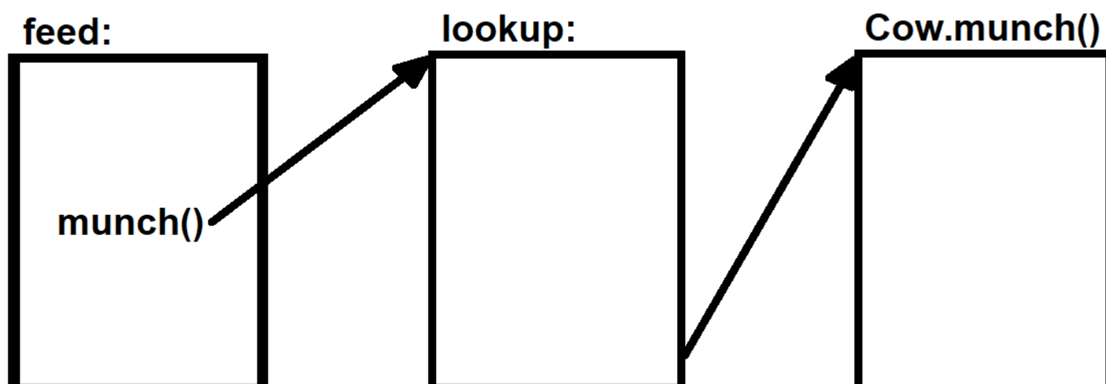
Ein Problem beim Inlining ist, dass nicht immer erkennbar ist, welche Implementation aufgerufen wird:

```
1 function feed(animal, food) {
2     animal.munch(food);
3 }
```

Listing 3.1: Codebeispiel für Inlining

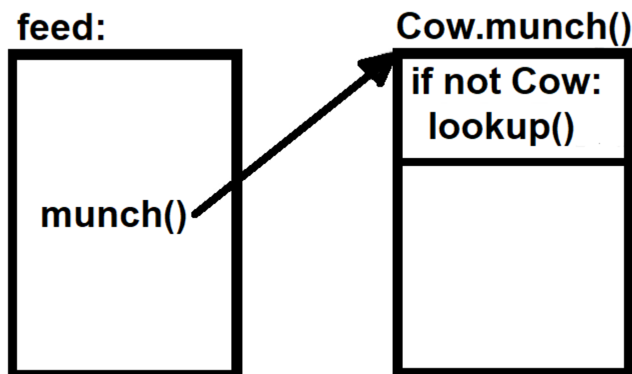
In diesem Beispiel soll *animal* ein Interface sein mit mehreren Implementationen. Welches Tier (welcher Typ) als Parameter übergeben wird, bestimmt welche Funktion in Zeile 2 aufgerufen wird. Es kann beliebig viele Implementationen von *munch()* geben.

Bei diesem Beispiel kann ein AOT-Compiler nicht inlinen, stattdessen würde zur Laufzeit ein sogenannter lookup stattfinden, eventuell sogar durch mehrere Ebenen.



Wird z.B. die *feed* Funktion mit dem Typ *Cow* aufgerufen, muss von Zeile 2 aus in ein "lookup table" gesprungen werden, um die passende Implementation zu finden, um wiederum zu dieser zu springen. Die Sprünge und das Durchsuchen des lookup table kosten viel Zeit und sollten möglichst verhindert werden.

Monomorphic Inline Caching ist eine Form des Inlining, das zur Laufzeit ausgeführt wird. Hierbei wird die zuletzt genutzte Implementation inlined. Das vorherige Beispiel würde mit Monomorphic Inline Caching wie folgt aussehen:

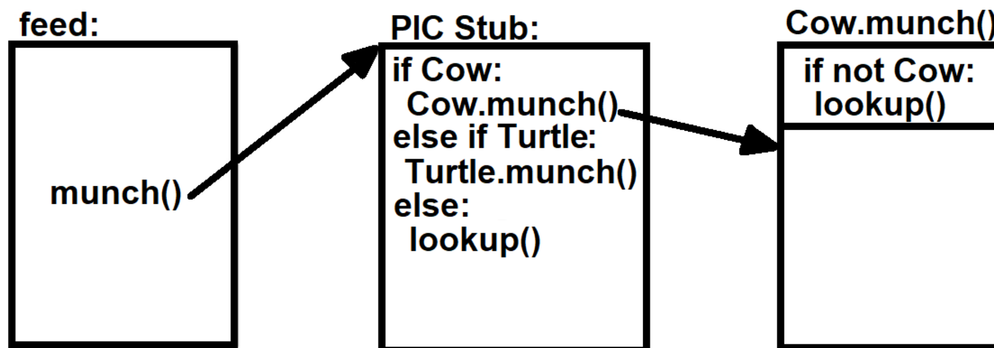


Nach diesem Inlining würde man in dem Programm einen Sprung sparen, wenn *feed* wieder mit einem Parameter von Typ *Cow* aufgerufen wird, sonst hätte man einen Sprung mehr im Programm.

Wenn man sich z.B. überlegt, dass dieser Code in einem Farmsimulator benutzt wird, kann man sich gut vorstellen, dass die Tiere Art für Art gefüttert werden (erst alle Schweine, dann die Kühe etc. ...) und somit pro Tier einer Art ein Sprung im Code gespart wird und pro Art Tier ein Sprung mehr benötigt wird. Das sollte in der Regel zu einer geringeren Laufzeit führen.

Probleme gibt es bei diesem Ansatz, wenn ständig unterschiedliche Implementationen ausgeführt werden, da dann zusätzliche Arbeit geleistet werden muss.

Polymorphic Inline Caching (PIC) berücksichtigt, dass eventuell einige wenige Implementationen - aber mehr als eine - hauptsächlich aufgerufen werden. Es werden bei diesem Ansatz die letzten paar Implementation inlined (wie viel genau variiert aber meist sind es nicht mehr als 4), da sonst die Laufzeit unter den zusätzlichen Vergleichen leidet.



Das Bild zeigt, wie der Code aussehen könnte, nachdem *feed* noch einmal mit einem Parameter vom Typ *Turtle* aufgerufen wird. Es wird ein sogenannter PIC stub erstellt, der die zuletzt verwendeten Implementationen speichert. Hierbei wird zwar kein Sprung mehr eingespart, aber die Suche durch den lookup table. Dieser PIC stub könnte z.B. die letzten vier Implementationen speichern, die aufgerufen wurden.

3.2 Polymorphic Inline Caching Performance

Eine unten genannte Studie hat objektorientierte Programme in der Sprache SELF untersucht und versucht, sie mithilfe von Polymorphic Inline Caching zu optimieren.
Im Median wurde ein speedup von 11% erreicht.
Des Weiteren wurden beim PIC Laufzeitinformationen gesammelt, die genutzt werden konnten, um einen speedup von 27% zu errreichen. Beim PIC werden Informationen dazu gesammelt, wie oft welche Implementationen aufgerufen werden; besonders häufig genutzte können dann gezielt optimiert werden.
Studie: [7]"Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches"

Allerdings können PICs die Performance auch verschlechtern, wenn zu oft verschiedene Implementationen aufgerufen werden, so dass der PIC stub häufig geändert werden muss und man sich nicht den lookup spart. Solche Fälle werden megamorphic genannt.

3.3 Adaptive Optimierung

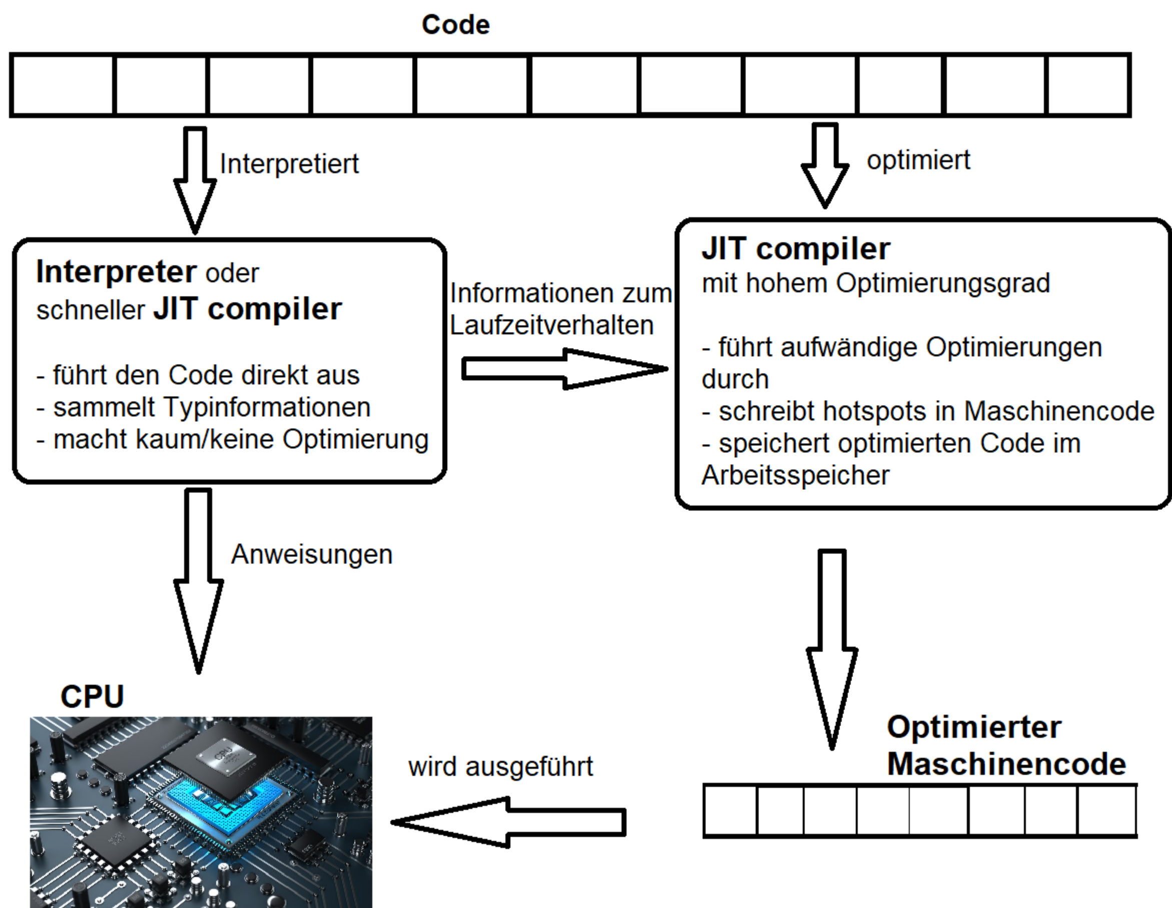
Adaptive Optimierung bedeutet, dass Teile des Codes zur Laufzeit rekompiliert werden. Um Adaptive Optimierung nutzen zu können, müssen Laufzeitinformationen gesammelt werden, wie z.B. beim PIC.

Wenn ein Compiler zur Laufzeit rekompilieren kann, können aggressive Optimierungen durchgeführt und rückgängig gemacht werden, falls nötig. Variablen können als Konstanten angenommen werden, wenn der Compiler merkt, dass sich der Wert lange nicht verändert hat.

Um effektiv und vor allem effizient zu optimieren, werden so genannte hotspots ausfindig gemacht; gemeint sind damit Teile des Codes, die besonders häufig ausgeführt werden. Hier lohnt es sich, aufwändige und zeitintensive Optimierungen durchzuführen, da selbst kleine Gewinne in der Laufzeit sich durch die große Anzahl an Aufrufen bezahlt machen. Teile im Code, die sehr selten, vielleicht sogar nur einmal ausgeführt werden, bleiben unverändert, da Optimierungen hier vermutlich mehr Zeit brauchen, als sie sparen.

4 Nutzung von JITs

JIT-Compiler werden meist nicht allein verwendet, sondern in Kombination mit einem Interpreter oder anderen Compilern. So kann beim Start des Programmes der Interpreter den Code ausführen und Informationen sammeln, bis der JIT-Compiler den Code so weit übersetzt hat, dass der kompilierte Code ausgeführt wird. Diese Kombination aus Interpreter und JIT-Compiler wird auch "mixed-mode execution engine" gennant.



JIT-Compiler sind von Vorteil für Code, der oft verändert wird, z.B. in der Entwicklung von Code. Mit einem JIT-Compiler lässt sich Code schneller ausführen und testen, als wenn man diesen zuerst Ahead-of-time kompiliert und dann testet.

Auch in der Lehre, wo oft sehr kleine Programme ausgeführt werden, können JIT-Compiler sinnvoll genutzt werden, da das Kompilieren höchstwahrscheinlich länger dauern würde als man Zeit spart.

JIT-Compiler kann es für jede Sprache geben, genau so wie AOT-Compiler, aber die meisten Sprachen benutzen hauptsächlich nur eins von beiden. Java, JavaScript und C# sind Beispiele für Programmiersprachen, die hauptsächlich JIT-Compiler nutzen. Java Code wird meist in einer JVM - Java Virtual Machine - ausgeführt; diese nutzt eine Kombination aus Interpreter und JIT-Compiler; "HotSpot" ist die am weitesten verbreitete JVM von Oracle.

5 Vergleich von AOT und JIT

Compilezeit:

- Ahead-of-time
  - Code muss vor der Ausführung kompiliert werden
- Just-in-time
  - wird nicht extra kompiliert

Laufzeit:

- Ahead-of-time
  - optimierter Code muss nur ausgeführt werden
- Just-in-time
  - zur Laufzeit muss kompiliert und optimiert werden
  - startet langsamer, aber wird theoretisch mindestens so schnell wie AOT

Da ein JIT-Compiler theoretisch jede Optimierung durchführen kann, die ein AOT vornimmt, und Möglichkeiten hat, die AOTs nicht haben, könnte ein JIT-Compiler nach einer gewissen Laufzeit Code haben, der schneller läuft als AOT kompilierter Code.

Arbeitsspeicher:

- Ahead-of-time
  - benötigt zum Kompilieren und Ausführen jeweils Speicher
- Just-in-time
  - benötigt Speicher für beides gleichzeitig
  - der kompilierte Code wird im Arbeitsspeicher abgelegt statt als extra Datei

AOT-Compiler verteilen ihre Last auf zwei Vorgänge und brauchen weniger maximalen Arbeitsspeicher.
JIT-Compiler benötigen zur Laufzeit einiges mehr an Arbeitsspeicher und könnten daher für manche Kombinationen von Anwendung und Hardware ungeeignet sein.

Plattformunabhängigkeit:

- Ahead-of-time
  - nach der Kompilierung ist das Programm auf eine bestimmte Hardware ausgelegt
  - als Nutzer man muss sich das Programm also selber kompilieren oder eine passend kompilierte Datei bekommen
- Just-in-time
  - dem JIT compiler wird Code gegeben, der plattformunabhängig ist
  - allerdings muss man einen JIT compiler installiert haben, der für die eigene Hardware ausgelegt ist.

Compiler Updates:

- Ahead-of-time
  - sollte es einen neuen besseren Compiler geben, müssten Programme neu vom Sourcecode kompiliert werden
- Just-in-time
  - eine neue Version vom JIT compiler kann installiert und auf dem alten Code ausgeführt werden

Optimierung:

- Ahead-of-time
  - kann viel Zeit und Ressourcen nutzen, um das Programm zu optimieren
  - besitzt nur statische Informationen und eventuell Profile
- Just-in-time
  - muss während der Laufzeit optimieren
  - nutzt Laufzeitinformationen und kann genau auf Hardware eingehen

Zwei verschiedene CPUs könnten z.B. dem gleichen Standard folgen und trotzdem kleine Unterschiede und Optimierungsmöglichkeiten haben. Ein JIT-Compiler kann zur Laufzeit sehen, welche Hardware verwendet wird, und kann alle Eigenschaften davon nutzen. Für den gleichen Effekt bei einem AOT-Compiler müsste dieser auf genau die verwendete Hardware ausgelegt sein.

6 Zusammenfassung

AOT-Compiler bieten sich an für Programme, die lange unverändert genutzt werden. Man kann einmal eine aufwändige Optimierung durchführen und muss dann nur noch den optimierten Code ausführen. In diesem Anwendungsfall würde ein JIT-Compiler bei jeder Ausführung den Code erneut optimieren und dadurch die Laufzeit stark erhöhen.

JIT-Compiler sind einfacher und schneller bei sehr kleinen Programmen, bei denen der zusätzliche Aufwand, um das Kompilieren überhaupt zu starten, schon den Gewinn durch Optimierungen übersteigt.
Auch bei großen Anwendungen, die lange am Stück laufen, glänzt ein JIT-Compiler, da hotspots erkannt und zur Optimierung genutzt werden können. Ebenso ist ein JIT von Vorteil, wenn nur wenige Teile des Programmes genutzt werden, da keine überflüssigen Optimierungen durchgeführt werden.

Ich denke, für besonders rechenintensive Anwendungen, wie z.B. in der Klimaforschung, sind AOT-Compiler von Vorteil, da diese meist nicht interaktiv sind und in der Regel ein Großteil vom Code ausgeführt wird. Außerdem ist der Arbeitsspeicher eine wertvolle Ressource bei solch großen Anwendungen und man will keine Performance verlieren, dadurch dass ein JIT-Compiler zusätzlich den Arbeitsspeicher belastet.

Für Anwendungen am privaten PC lohnt sich meist ein JIT-Compiler, da diese oft interaktiv sind und Laufzeitinformationen gut zur Optimierung genutzt werden können. Oft werden auch große Teile von Anwendungen gar nicht genutzt. Bei den meisten Anwendungen wird der PC nicht maximal ausgelastet, so dass die Zusatzlast eines JIT-Compiler nicht weiter auffällt.

Optimieren zur Laufzeit und zur Compilezeit hat also jeweils Vor- und Nachteile, die abgewogen werden müssen.

7 Quellen

[1]https://softwareengineering.stackexchange.com/questions/246094/understanding-the-differences-traditional-interpreter-jit-compiler-jit-interp
[2]http://bibliography.selflanguage.org/_static/implementation.pdf
[3]https://de.wikipedia.org/wiki/Just-in-time-Kompilierung
[4]https://en.wikipedia.org/wiki/Just-in-time_compilation
[5]https://sites.cs.ucsb.edu/ urs/oocsb/papers/pldi94.pdf
[6]https://jayconrod.com/posts/44/polymorphic-inline-caches-explained
[7]https://bibliography.selflanguage.org/ _static/pics.pdf
[8]https://en.wikipedia.org/wiki/Interpreter_(computing)
[9]https://stackoverflow.com/questions/2106380/what-are-the-advantages-of-just-in-time-compilation-versus-ahead-of-time-compila
[10]https://talktechnical21.files.wordpress.com/2018/06/1680219.jpg

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Hamburg, 28.04.2020	Mats Schrader
Ort, Datum	Unterschrift