Erfahrungen bei der Auswahl von maschinenunabhängigen Optimierungen im Bereich des mobilen Codes

Wolfram Amme

Institut für Informatik
Friedrich-Schiller-Universität Jena
Jena, Deutschland
Wolfram.Amme@uni-jena.de

Mobiler Code wird heutzutage auf der Basis einer Just-in-Time (JIT)- Übersetzung ausgeführt. Da die für eine JIT- Übersetzung notwendige Zeit direkt in die Ausführungszeit der Programme einfließt, ist man darauf bedacht, die für eine solche Übersetzung notwendigen Zeiten möglichst gering zu halten. Eine Möglichkeit dieses Vorhaben zu unterstützen, ist schon während der Erzeugung von mobilen Programmen auf der Produzentenseite eines Systems zum Transport mobiler Programme maschinenunabhängige Optimierungen auszuführen.

Im Vortrag wird die Wirkungsweise vorgestellt, die eine Verwendung von maschinenunabhängigen Optimierungen auf die Ausführung von mobilen Programmen hat. Alle dabei betrachteten Experimente wurden unter Verwendung des SafeTSA-Formats durchgeführt, eines auf der SSA-Form basierenden Zwischencodeformats für mobilen Code.

SafeTSA ist die erste absolut referenz- und typensichere auf *Static Single Assignment-Form* basierende Zwischencoderepräsentation für mobilen Code. Die in [1] näher beschriebene Methode wurde als eine Alternative zur JVM und dessen Java-Bytecode entworfen, und hat gegenüber diesen mehrere entscheidene Vorteile: (1) SafeTSA ist besser als Eingabe für einen optimierenden JIT-Compiler geeignet und erlaubt, viele maschinenunabhängige Codeoptimierungen bereits während der Erstellung der Zwischencoderepräsentation auf der Produzentenseite durchzuführen. (2) SafeTSA garantiert die Referenz- und Typensicherheit des gesendeten Programmes durch Konstruktion. Diese Eigenschaft reduziert die zur Sicherheitsgarantie notwendigen Verifikationsarbeiten eines JIT-Compilers auf ein Minimum. (3) SafeTSA besitzt separate Instruktionen zur Nullreferenz- und Arrayindexüberprüfung, damit können überflüssige Nullreferenz- und Arrayindexüberprüfungen schon vor Erzeugung des Zwischencodes eliminiert werden, was zu einem erheblich besseren Laufzeitverhalten von Java-Applikationen führt. (4) Neben diesen Vorteilen ist SafeTSA zusätzlich kompakter als Java-Bytecode.

Zur Überprüfung der Leistungsfähigkeit des von uns geschaffenen SafeTSA-Formats wurde ein vollständiges System zum Transport von mobilen Code entwickelt, mit dem SafeTSA-Programme einerseits generiert bzw. andererseits unter Verwendung einer JIT-Übersetzung ausgeführt werden können. Der prinzipielle Aufbau der für die Programmiersprache Java entworfenen Produzentenseite des Systems ist in Abbildung 1 wiedergegeben. Die Produzentenseite führt

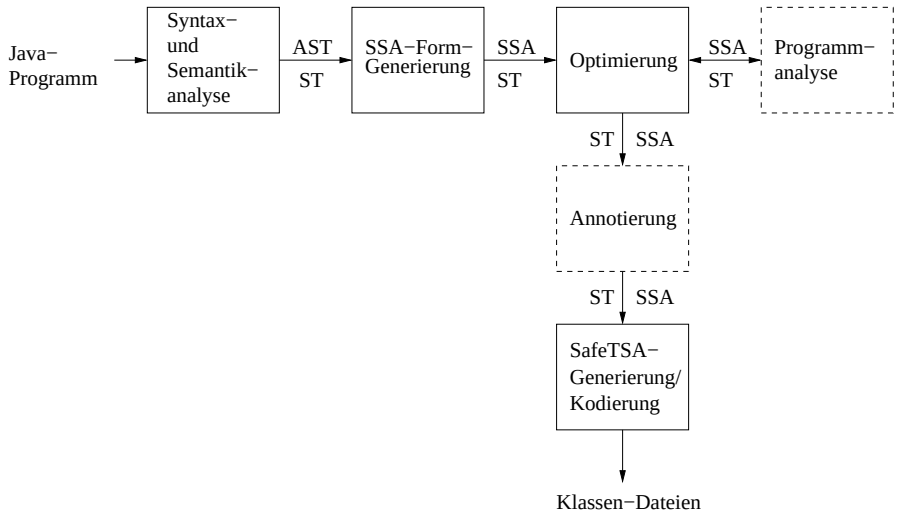

Abbildung 1. Struktureller Aufbau des SafeTSA-Übersetzers

zunächst eine Syntax- und Semantikanalyse aus und transformiert das Eingabeprogramm nach durchgeführter Analyse in einen abstrakten Syntaxbaum (AST) mit zugeordneter Symboltabelle (ST). Nach Transformation in den abstrakten Syntaxbaum wird das zu bearbeitende Programm in SSA-Form gebracht, um nach Durchführung von maschinenunabhängigen Optimierungen in SafeTSA-Format umgewandelt und nach Klassen unterteilt in Dateien abgelegt zu werden. Optional können dem SafeTSA-Format durch Einschalten einer Annotierungskomponente zusätzlich Programminformationen hinzugefügt werden (vergl. etwa [2] oder [3]), die dann vom JIT-Übersetzer zur Durchführung von maschinenabhängigen Optimierungen genutzt werden können.

Bei der Konzeption der Konsumentenseite unseres Systems wurde besonders darauf geachtet, die Systemkomponenten weitestgehend unabhängig von verwendeter Zielarchitektur und Systemumgebung zu realisieren, was eine schnelle Integration der Konsumentenseite in bereits existierende Laufzeitumgebungen unterstützt. Auf der Konsumentenseite wird zunächst vom Dekodierer die SafeTSA-Struktur des Programms wiederhergestellt. Die Überprüfung der Typ- und Referenzsicherheit wird gleichzeitig während der Wiederherstellung des Zwischencodeformats durchgeführt. Der optimierende Codegenerator der Konsumentenseite überführt die verifizierte SafeTSA-Darstellung dann sukzessive in verschiedene Zwischencoderepräsentationen, führt Optimierungen auf diesen aus und bildet in einem letzten Schritt den Maschinencode für die jeweilige Zielarchitektur. Es wurden Konsumentenseiten des Systems für Intels IA32-Architektur und IBMs PowerPC-Architektur geschaffen [4]. Beide Konsumentenseiten wurden in Form eines auf der Zwischencoderepräsentation *SafeTSA* basierenden optimierenden JIT-Compilers für IBMs virtuelle Maschine JikesRVM [5] realisiert.

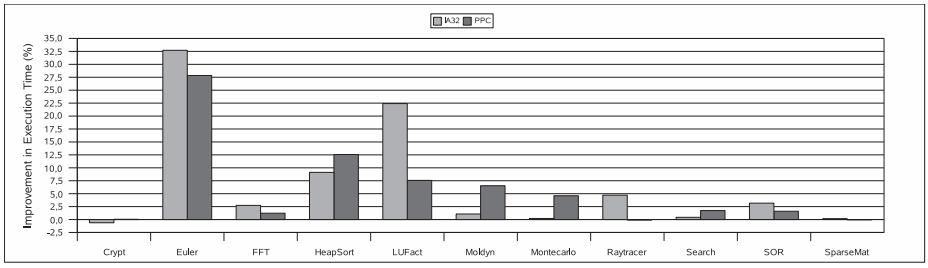

Abbildung 2. Kombination ausgewählter Optimierungen

Zur Auswahl der in die Produzentenseite unseres Systems zu integrieren maschinenunabhängigen Optimierungsarten wurde eine auf den Java-Grande-Forum-Benchmark-Programmen (JGF-Benchmarks) basierende Testumgebung geschaffen. Die JGF-Benchmarks stehen für eine Ansammlung von in der Programmiersprache Java geschriebenen Anwenderprogrammen, die im Jahr 2000 vom *Java Grande Forum* zur Überprüfung der Einsetzbarkeit von Java-Ausführungsumgebungen (JVMs, Java-Übersetzer, Java-Hardware, etc.) für rechenintensive Anwendungen vorgeschlagen wurden [6]. Die zur Überprüfung der Optimierungsarten durchgeführten Messungen wurden auf beiden von der Jikes-RVM unterstützten Rechner-Architekturen durchgeführt. Als Repräsentant der PowerPC-Architektur wurde ein PowerMacG4-Rechner verwendet, der mit einem PowerPC-G4-Prozessor (733 MHz), einem Hauptspeicher von 1,5 GB und einem L2-Cache (256 KB) ausgestattet war. Die Messungen für die IA32-Architektur wurden auf einem Standard-PC ausgeführt, der mit einem Pentium-3-kompatiblen Prozessor (1,5 GHz) und einem Hauptspeicher von ebenfalls 1,5 GB ausgestattet war.

Die auf der PowerPC- und IA32-Architektur durchgeführten Messungen zeigen, dass üblicherweise als maschinenunabhängig eingestufte Optimierungsformen tatsächlich oftmals sehr maschinenabhängiger Natur sind. Tatsächlich konnte keine Optimierungsart gefunden werden, die für beide Architekturen und allen Benchmarkprogrammen aussschließlich zu einer Verbesserung im Laufzeitverhalten führt. Die für die IA32-Architektur erzielten Ergebnisse offenbarten insbesondere, dass die Ausführung von Programmen in starkem Maße von der Anzahl an zur Verfügung stehenden Registern und der für die Registerzuordnung verwendeten Registerallokationsstrategie abhängt. Für SafeTSA-Programme zeigte sich dabei, dass die Durchführung von maschinenunabhängigen Optimierungen für die IA32-Architektur oftmals nur dann Sinn macht, wenn diese nicht über Basisblöcke hinweg durchgeführt werden. Weiter deckten die Messungen auch auf, dass bei den Optimierungen durchgeführte Umstrukturierungen zusätzlich das Cache-Verhalten eines Programms ungünstig verändern und damit dessen Laufzeitverhalten verschlechtern können.

Nichtsdestotrotz belegen die durchgeführten Messungen jedoch auch, dass die Verwendung von maschinenunabhängigen Optimierungen auch im Bereich

des mobilen Codes durchaus einen Nutzen haben kann. Insbesondere zeigen die Ergebnisse, dass die Verwendung von nicht über Basisblöcken hinweg durchgeführten Optimierungen – konkret gesehen, eine Eliminierung von nutzlosen Programmcode, eine Konstantenfortpflanzung, eine Eliminierung von Lade- und Speicherbefehlen sowie eine Eliminierung von gemeinsamen Teilausdrücken – eine sinnvolle Kombination maschinenunabhängiger Optimierungen für den Einsatz im SafeTSA-System sein kann [7].

Abbildung 2 zeigt die mit dieser Kombination an Optimierungsarten erreichbaren Laufzeitverbesserungen relativ zum Laufzeitverhalten von nicht optimierten SafeTSA-Programmen. Wie in der Abbildung zu sehen ist, konnte auf der PowerPC-Architektur für drei Benchmarkprogramme mit den durchgeführten Optimierungen eine Laufzeitverbesserung von mehr als 7,5 % erreicht werden. Auf der IA32-Architektur konnte mit den Optimierungen für vier Benchmarkprogramme das Laufverhalten um über 4 % verbessert werden, wobei für zwei der Benchmarkprogramme sogar eine Laufzeitverbesserung von mehr als 20 % zu beobachten war.

Literatur

1. Amme, W., Dalton, N., Franz, M., von Ronne, J.: SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In: Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, Snowbird, Utah, USA (June 20–22, 2001)
2. Hartmann, A., Amme, W., von Ronne, J., Franz, M.: Code annotation for safe and efficient dynamic object resolution. In Knoop, J., Zimmermann, W., eds.: Proceedings of Compiler Optimization Meets Compiler Verification (COCV'2003), Warsaw, Poland (April 2003) 18–32
3. Amme, W., Möller, M.A., Adler, P.: Data flow analysis as a general concept for the transport of verifiable program annotations. Electron. Notes Theor. Comput. Sci. **176**(3) (2007) 97–108
4. Amme, W., von Ronne, J., Franz, M.: SSA-Based Mobile Code: Implementation and Empirical Evaluation. ACM Transactions on Architecture and Code Optimization **4**(2) (2007)
5. Alpern, B., Attanasio, C.R., et al.: The Jalapeno virtual machine. IBM System Journal **39**(1) (February 2000)
6. Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S., Davey, R.A.: A benchmark suite for high performance Java. Concurrency: Practice and Experience **12**(6) (May 2000) 375–388
7. Amme, W., von Ronne, J., Adler, P., Franz, M.: The effectiveness of producer-side machine-independent optimizations for mobile code. Softw., Pract. Exper. **39**(10) (2009) 923–946