Methoden zur Code-Komprimierung

Christian Biesinger
0225227

Dezember 2006/Januar 2007

Zusammenfassung

Diese Arbeit vergleicht zwei Ansätze zur Komprimierung von Code. Komprimierung ist sinnvoll, um in Geräten mit limitiertem Speicher dennoch komplexe Anwendungen ausführen zu können. Eine der Methoden erlaubt auch die Erstellung von schnellen, plattformunabhängigen Anwendungen und basiert auf einer Serialisierung des Syntaxbaums. Die andere Vorgehensweise ist besonders für embedded devices gedacht, weshalb Performance und Codegröße im Vordergrund stehen; bei ihr wird der endgültige Maschinencode komprimiert. Gemeinsam ist ihnen die deutliche Verringerung der Programmgröße.

1 Einführung

Dieses Paper vergleicht zwei Methoden ([FK97], [DE02]), die Größe von Programmen mithilfe von Komprimierung zu verringern.

Für die Verwendung solcher Methoden gibt es mehrere Gründe. Die Haupmotivation der „slim binary“-Methode [FK97] ist der Wechsel von Hardwarearchitekturen, der normalerweise neu kompilierte Programme voraussetzt, oder eine langsame Emulation der alten Architektur. Um das zu vermeiden, ließen sich „fat binaries“ einsetzen, die allerdings zu sehr großen Dateien führt, was zum Beispiel beim Anbieten von Programmen über das Internet nachteilig ist. Dennoch wird diese Methode derzeit von Apple genutzt, um dasselbe Binary für Intel- und PowerPC-Architekturen verwenden zu können [App06].

Außerdem ergibt sich eine schnellere Ausführung des Codes – das Dekomprimieren sowie das Erzeugen von Maschinencode geht schneller, als unkomprimierten Maschinencode von der Festplatte zu lesen. Zudem erlaubt die Verwendung dieser Methode, für den tatsächlich vorhandenen Prozessor optimierten Code zu erzeugen.

Die „profile-guided code compression“-Methode [DE02] geht im Gegensatz dazu von Systemen aus, auf denen der Speicherplatz knapp ist (embedded systems). Deshalb wird hier primär selten ausgeführter Code komprimiert, da der Dekomprimierungsoverhead hier als hoch eingeschätzt wird.

2 Slim Binaries

2.1 Kodierung

Die in [FK97] vorgestellten slim binaries funktionieren mithilfe eines Zwischencodes, der vom Compiler erzeugt wird. Anders als etwa bei Java [LY99] wird hier eine Serialisierung des abstrakten Syntaxbaums (AST) gespeichert.

Zusätzlich wird der Code komprimiert, mit einer LZW [Wel84]-ähnlichen Methode. Der begrenzte Geltungsbereich von Variablen lässt sich hier gut ausnutzen, denn sobald eine Variable den aktuellen Scope verlässt, kann sie aus dem Wörterbuch entfernt werden. Zusätzlich zu den Variablen des aktuellen Geltungsbereichs enthält das Dictionary Operationen wie Addition, Multiplikation, etc. sowie sichtbare Funktionen.Um oft vorkommende Ausdrücke effizient zu kodieren, werden auch vorrausschauend Wörterbucheinträge hinzugefügt, s.u.

Zusätzlich wird auch eine Symboltabelle im resultierenden Binary gespeichert, was für die Codeerzeugung zur Laufzeit weitere Optimierungen erlaubt. Zudem wird sie für den Aufbau des Wörterbuches benötigt.

Die Kodierung des eigentlichen Codes funktioniert folgendermaßen (vgl. auch Abb. 1). Der AST wird traversiert und entsprechend dem aktuellen Wörterbuch kodiert. Beispielsweise wird ein funktionsaufruf $P(i + 1)$ als *procedure call* und *addition* kodiert, zusammen mit Datensymbolen *Funktion* P, *Variable* i und Konstante 1. Weil aber davon ausgegangen wird, dass das Programm viele einander ähnliche Codestücke enthält, werden auch Symbole für $P(.)$, $i+.$, $i+1$ etc. zum Wörterbuch hinzugefügt. Das erlaubt eine kürzere Kodierung, falls ähnliche Codestücke später im Code vorkommen.

Erst zur Laufzeit wird tatsächlich Maschinencode erzeugt. Dieser Vorgang ist jedoch „orders of magnitude faster than the traditional compilers and linkers“ [FK97], was auch daran liegt, dass viel weniger von der Festplatte gelesen werden muss, und das Schreiben ganz wegfällt.

2.2 Module

Die Implementierung von [FK97] unterstützt das Verwenden anderer Softwarekomponenten mittels Modulen (konkret anhand der Programmiersprache Oberon[1]). Jedes Modul kann eine Liste von Funktionen exportieren, die von anderen Modulen verwendet wird, sowie Funktionen anderer Module importieren, die es selber verwendet. Das Betriebssystem selbst stellt ebenfalls ein oder mehrere Module dar. Die Typsicherheit kann statisch sichergestellt werden, und auch der dynamische Loader kann die korrekte Verwendung importierter Funktionen nochmals überprüfen.

[1] Die aktuelle Implementierung dieses Oberon-Systems verwendet allerdings keine slim binaries mehr, vgl. `http://www.oberon.ethz.ch/compiler/` (abgerufen am 5. Dezember 2006)

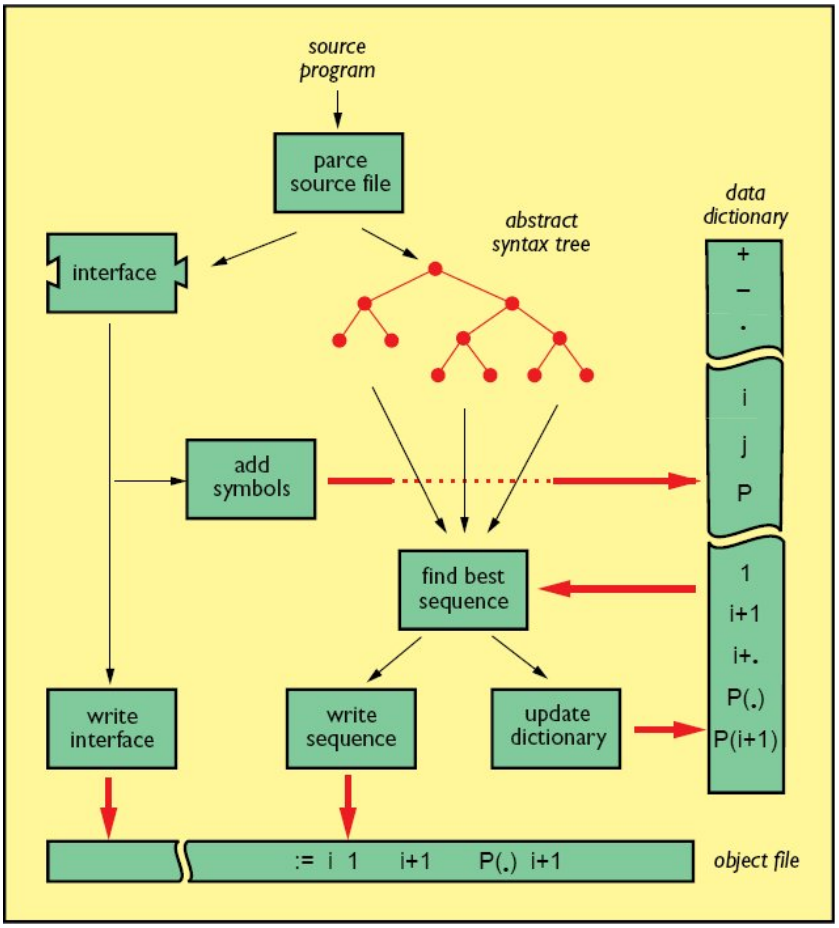

Abbildung 1: Übersetzung von Quelltext zu einem slim binary (Aus [FK97])

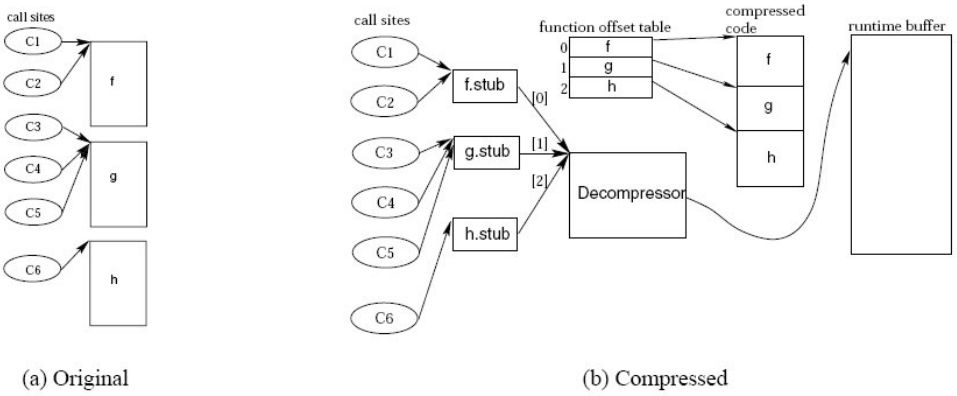

Abbildung 2: Stubs und Dekomprimierung (Aus [DE02])

3 Code Compression

Die „profile-guided code compression“-Methode aus [DE02] basiert im Gegensatz zu der bisher beschriebenen Methode bereits auf Maschineninstruktionen. Daher erlaubt sie nicht die Vorteile der Architekturunabhängigkeit.

Es werden bei dieser Vorgehensweise nur selten ausgeführte Funktionen gepackt; zwecks besserer Effizienz werden häufig aufgerufene Funktionen nicht komprimiert. Das vermeidet auch die leicht erhöhte Startzeit, die die vorher beschriebene Methode der slim binaries mit sich bringt.

Um eine Funktion zu komprimieren, wird sie durch eine „stub“-Funktion ersetzt, die den Dekomprimierer aufruft (vgl. Abb. 2). Zusätzlich gibt es eine *function offset table*, die die Stelle der komprimierten Funktionen im komprimierten Code enthält. Der Dekomprimierer entpackt diese Funktion dann in den Speicher und springt zu dem so generierten Code.

Ein potentielles Problem ergibt sich, wenn die so dekomprimierte Funktion eine andere Funktion aufruft, die ebenfalls komprimiert ist. Wird die aufgerufene Funktion in denselben Speicherbereich entpackt, gibt es Probleme, wenn die Funktion zum Aufrufer zurückkehren will, denn dieser existiert nicht mehr im Speicher. Entscheidet man sich andererseits, für jede Funktion einen neuen Speicherbereich anzulegen, wächst der benötigte Speicher stark. Deshalb wird in [DE02] stattdessen die erste Alternative genommen, aber bei der Rückkehr der Funktion die alte Funktion erneut dekomprimiert.

Dazu wird in den dekomprimierten Code vor dem Rücksprung ein Aufruf zu einer *CreateStub*-Funktion eingefügt, die diesen Stub erstellt. Dessen Aufgabe ist es, die richtige Funktion zu dekomprimieren und an die passende Stelle zu springen. (Stubs werden solange aufgehoben, wie sie benötigt werden; das wird mittels eines Referenzzählers sichergestellt)

Die eigentliche Komprimierung erfolgt mittels des „splitting streams“-

Ansatzes [EEF+97]. Hierbei werden Maschineninstruktionen aufgeteilt in die Felder die sie enthalten (opcode, register, displacement, etc.) Dann wird für jeden Typ ein eigener Stream angelegt, der nur Werte dieses Typs enthält. Bei der Dekomprimierung wird dann anhand des Opcodes erkannt, von welchen Streams die weiteren Werte gelesen werden müssen. Jeder Stream wird dann separat mit einer Huffman-Kodierung gepackt (canonical Huffman encoding [WMB94]).

Zu beachten ist auch, dass die Definition einer Funktion für diesen Algorithmus nicht der üblichen Definition entspricht – auch Codeblöcke, die Teil einer Funktion sind, können einzeln komprimiert werden.

Um nun festzustellen, welche Funktionen komprimiert werden sollen (d.h. welche Funktionen selten ausgeführt werden), wird zuerst ein Profil des Programms erstellt, d.h. das Programm ausgeführt und gezählt, wie viele Instruktionen ausgeführt werden. Alle basic blocks, die dann nur selten ausgeführt werden, werden komprimiert (für die genaue Erklärung, wie diese Berechnung funktioniert, siehe [DE02]).

4 Vergleich

Slim Binaries erlauben eine Größenersparung bis zu einem Faktor von 3,16 [FK97] (PowerPC, Oberon Net-Paket). Hingegen lassen sich mit Profile Guided Code Compression nur etwa 20% einsparen [DE02] (Alpha, verschiedene Programme). Durch die Beschränkungen bei der CPU sowie beim Speicher ist es nicht möglich, hier bessere Resultate zu erzielen. Andererseits liegen die Performanceeinbußen nur zwischen 4% und 24% (je nach Wahl des thresholds). Der Programmstart wird bei dieser Methode nicht verlangsamt.

Anders verhält es sich bei den slim binaries. Hier ist ausschließlich die Startzeit von Geschwindigkeitseinbußen betroffen. Der Unterschied zu nativen Programmen liegt hier bei einem Faktor von 2 [FK97]. Der Nachteil wird aufgewogen durch die Kompatibilität mit allen unterstützten Plattformen durch das gleiche Binary sowie deutlich kleinere Binaries.

5 Zusammenfassung

Es wurden zwei verschiedene Methoden vorgestellt, die Programmcode komprimieren. „Slim binaries“ [FK97] serialisieren den AST, wobei ein LZW-basierter Algorithmus und ein dynamisches Wörterbuch verwendet wird. Durch spekulatives Hinzufügen von Symbolen werden weitere Vorkommen von ähnlichem Programmcode effizienter kodiert. Das Speichern der Symboltabelle erlaubt weitere Optimierungen bei der Codegenerierung, die zur Laufzeit stattfindet. Da verglichen mit herkömmlichen Compilern viel weniger Festplatten-Input/Output stattfindet, ist die Erzeugung des Maschinencodes auch deutlich schneller – Schreiben findet nicht statt, gelesen wird

nur der komprimierte Code. Implementiert wurde diese Methode in einem Oberon-System.

Bei dem zweiten Ansatz, „profile-guided code compression“ wird versucht, die Laufzeitnachteile von komprimiertem Code gering zu halten. Dazu werden nur jene Funktionen komprimiert, die selten aufgerufen werden („cold code“). Welche das sind, wird mittels Profilings festgestellt. Um Speicher zu sparen, wird immer nur höchstens eine Funktion gleichzeitig im Speicher gehalten; kehrt eine komprimierte Funktion zu einer anderen komprimierten Funktion zurück, wird diese erneut dekomprimiert. Diese Methode ist Programmiersprachenunabhängig und lässt sich auch auf fertigen Binaries anwenden; auch sie setzt jedoch architekturspezifischen Code vorraus.

Literatur

[App06] Apple Computer, Inc. *Universal Applications*, 2006. Abgerufen am 5. Dezember 2006.

[DE02] Saumya Debray and William Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, New York, NY, USA, 2002. ACM Press.

[EEF+97] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. Code compression. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 358–365, New York, NY, USA, 1997. ACM Press.

[FK97] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, 1997.

[LY99] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley Professional, second edition, 1999.

[Wel84] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[WMB94] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, 1994. Zitiert in [DE02].