Seminarausarbeitung
zur Lehrveranstaltung 185.272
"Grundlagen methodischen Arbeitens"
im WS 2006/2007

Space Efficient Conservative Garbage Collection

bearbeitet von

Johannes Buchner
Matrikelnummer: 0625457 Studienkennzahl: 534

Technische Universität Wien
Fakultät für Informatik
Institut für Computersprachen
Arbeitsbereich Programmiersprachen und Übersetzer

Lehrveranstaltungsleiter: A.o. Univ.Prof. Dr. Anton Ertl

Eingereicht am 08.01.2007

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe, und ich diese Arbeit zuvor keiner anderen Stelle oder Institution als Studiums- oder Prüfungsleistung vorgelegt habe.

Wien, den 07.01.2007

1 Kurzzusammenfassung

Der 1993 im PLDI von ACM SIGPLAN veröffentlichte Artikel von Hans-J. Boehm über „Space Efficient Conservative Garbage Collection" wurde als einer der besten und bedeutendsten Artikel ausgewählt. Er behandelt Methoden und Überlegungen, die Unterscheidung von Daten und Pointern bei konservativen Garbage Collectors zu verbessern, um nicht mehr referenzierten Speicher besser zu erkennen und freizugeben.

Dieser Artikel analysiert den Text und die von Hans-J. Boehm 20 Jahre später geschriebene Retrospektive. Es zeigt sich, dass einige große Projekte von dem Artikel entscheidend beeinflusst wurden und Forschungen in diesem Gebiet auf seinen Artikel aufbauten.

2 Motivation

Garbage Collectors sind vielen Programmierern wohl mit der Programmiersprache Java oder Mono das erste Mal begegnet. Die Vorarbeit, dass Garbage Collection effizient funktioniert und eine reine malloc/free Umgebung abgelöst hat, wurde unter anderem von dem analysierten Artikel geleistet. Es ist außerdem bemerkenswert, dass Voraussagen im Artikel später von Studien bestätigt wurden. Dass der Artikel spätere Projekte und Arbeiten beeinflusst hat, ist ein Grund, sich damit näher zu beschäftigen.

3 Begriffsfestlegungen

Der englische Name „Garbage Collector" wird verwendet, da dieser Ausdruck in die deutschen Sprache übernommen wurde. Es gibt Übersetzungen wie „Automatische Speicherbereinigung" oder „Freispeichersammlung". Der Begriff „konservative Garbage Collectors" wird einer vollständigen englischen Wortzusammensetzung vorgezogen. [3]

„Artikel" und „Autor" u.ä. verweist im folgenden immer auf die Originalveröffentlichung.

4 Inhalt des Artikels

Die folgenden Untersektionen analysieren die Teile des Artikels.

4.1 Einleitung

Die grundsätzliche Idee hinter Garbage Collectors ist, dass die Daten, die nicht mehr vom Programm referenziert werden, wieder zur Allokierung freigegeben werden. Wenn ein Pointer mit Daten den Kontext des Programmflusses verlässt, und vernichtet wird, bleiben die Daten normalerweise allokiert und

belegen Speicherplatz, da es ja sein könnte, dass ein anderer Pointer darauf referenziert. Wenn jedoch bei der Programmierung nicht darauf geachtet wird, die Daten wieder freizugeben, wenn kein Pointer mehr darauf verweist, entsteht ,,Garbage“, also Daten, die nur Speicher belegen, aber nicht mehr verwendet werden. Garbage Collectors sind dazu da, diesen belegten, aber nicht mehr referenzierten Speicher zu finden und wieder freizugeben. Der Autor bezieht sich zum Thema ,,Garbage Collectors“ auf eine Erhebung der benutzten Techniken aus 1992.

Konservative Garbage Collectors sind Garbage Collectors, die nicht die vollständige Information über den Aufbau der Daten im Stack und Heap haben. Hierzu wird auf früher geschriebene Arbeiten des Autors hingewiesen.

Das Hauptproblem, das in dem Paper dargelegt wird, ist, dass die Pointer prinzipiell nicht von Daten unterschieden werden können. Der konservative Garbage Collector muss annehmen, dass alle Adressen, die in den Datenfeldern stehen, noch nicht freigegeben werden dürfen. Es wird auf die Arbeit von Zorn verwiesen, die zeigt, das dies sehr ineffizient ist.

Es entstand außerdem das Problem von Speicherlecks, also stark anwachsende Speicherbereiche, da immer mehr Speicher fälschlicherweise nicht freigeben wird. Dazu verweist der Autor auf Veröffentlichungen zur Speicherleck- und Zugriffsfehlererkennung.

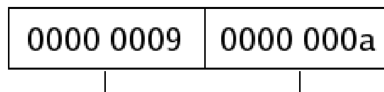
Die Problemstellung erscheint dem Autor insbesondere deswegen wichtig, da die Effizienz der Speichernutzung insbesondere bei langlaufenden Programmen leidet. Es ist besonders für Serveranwendungen untragbar, wenn man Programme regelmäßig beenden muss, weil der verfügbare Speicher ausläuft.

Der Beitrag, den dieser Artikel leistet, ist, dass er die Fähigkeit des konservativen Garbage Collectors, Speicheradressen richtig zu erkennen, auf ein hohes Maß steigert.

4.2 Hauptteil

4.2.1 Fehlerhafte Pointererkennung

Das größte Problem des konservativen Garbage Collectors ist, dass jede Integervariable theoretisch eine gültige Speicheradresse darstellt. Ebenso können zwei nebeneinander stehende Datenfelder eine gültige Adresse ergeben. Ein Beispiel im Artikel illustriert, wie aus zwei kleine Integerwerte eine gültige Pointeradresse entstehen kann.



Dies kann durch Wahl eines bestimmten Layouts beim Allokieren verbessert werden, sodass die Wahrscheinlichkeit, dass durch nebeneinander stehende Daten ein gültiger Pointer entsteht, sinkt. Eine weitere praktische Lösung ist, Speicher mit vielen Nullen beim Allokiervorgang zu vermeiden, um solche Verwechslungen zu verhindern. Dies bedarf jedoch in C/C++ eine Veränderung der Standardbibliotheken.

Außerdem ist es wünschenswert, dem Garbage Collector mitteilen zu können, dass bestimmte große Datengebiete (z.B. komprimierte Bitmaps) keine Pointer enthalten.

4.2.2 Systematische Techniken

Es wird auf eine Technik namens „Blacklisting" eingegangen, die verhindert, dass Daten dort allokiert werden, wo Pointer bereits hinzeigen, um falsche Gültigkeitsprüfungen von vornherein zu vermeiden. Dazu werden zuerst beim Programmstart alle Datenfelder auf Pointer geprüft und diese zur Blacklist hinzugefügt. Beim Allokieren werden dann die Adressen in der Blacklist nicht verwendet. Das Aktualisieren der Blacklist wird im Laufe des Programmflusses immer wieder wiederholt. Ein vereinfachter Pseudocode illustriert die Methode.

Eines der größten Ursachen für falsche Erkennung, nämlich dass Werte, die in Konstanten enthalten sind als Pointer behandelt werden müssen, ist damit gelöst. Dies wird von Tests untermauert.

Großzügig wird beim Blacklisting nicht nur die Pointeradresse, sondern die ganze Speicherseite für die Allokierung „verboten". Die Markierung findet mittels einem Hashtable von einem Bit statt.

Des weiteren werden noch andere Ursachen von Datenverwaisung und Fragmentierung dargelegt, und mögliche Lösungen dargelegt. Ein Problem ist dabei, dass durch das Blacklisting ganzer Speicherbereiche es schwierig wird, große Datenblöcke zu allokieren.

Im Verlauf des Papers wird ausgeführt, dass Blacklisting in den meisten Fällen besonders effizient arbeitet. Dies wird durch Testimplementationen und Benchmarks auf verschiedenen Plattformen untermauert. Es wird intensiver auf die Auswirkungen und Lösungen von falscher Pointererkennung bei nichttrivialen Anwendungen wie mehrfach verketteten Listen eingegangen. Verglichen wird der entwickelte Collector mit konservativen Implementationen und Programmen ohne Garbage Collector. Der Hauptteil ist stark praktisch ausgerichtet, er stellt Algorithmen/Methoden vor, die man praktisch umsetzen kann. Er erfasst Vergleiche und Benchmarks, verzichtet dabei aber auf theoretische Hintergründe wie Beweise oder Theoreme.

4.3 Verwandte Arbeiten

Im Artikel gibt es keine eigene Sektion für verwandte Arbeiten. Auf die wenigen Veröffentlichungen, die in diesem Gebiet relevant sind, hat der Autor in der

Einleitung verwiesen, was zum einen Implementationen von Garbage Collectors in verschiedenen Programmiersprachen, zum anderen unterschiedliche Ansätze zur Verbesserung der Effizienz der Pointererkennung.

4.4 Zusammenfassung

Der Autor betont, dass es in vielen Fällen nicht unbedingt auf die Implementation des Garbage Collectors, sondern u.a. auf die Architektur, die Umgebung und den Programmierstil ankommt, ob Datenlöcher entstehen, und die Speichernutzung somit nicht optimal ist. Jedoch schließt man, dass Blacklisting mit äußerst geringer Fehlerquote Pointer von Daten unterscheiden kann.

Auf die Auswirkungen der entwickelten Technik und welche Projekte davon betroffen sein werden, wird jedoch nicht eingegangen. Es werden auch keine weiterführenden Überlegungen angestellt, welche Probleme noch gelöst werden sollen. Jedoch wird aus den Schlussworten klar, dass mehr Benchmarks und Vergleiche gewünscht werden, um die Effizienz in allen Anwendungsfällen aufklären zu können.

4.5 Anhang

Es ist Quellcode zum Vergleichen von diversen Plattformen und Entwicklungsumgebungen angehängt. Dazu findet man eine Auflistung der während der Benchmarks gemachten Beobachtungen.

5 Retrospektive

Hans-J. Boehm schreibt in seiner Retrospektive, dass Typ-Erkennung und konservative Garbage-Collectors an Bedeutung gewonnen haben, seit das Paper veröffentlicht wurde. Es gab einige Implementationen der vorgestellten Algorithmen, insbesondere eine durch Geodesic Systems und eine als Open-Source gehaltene Entwicklung, die auch im gcj, dem GNU Java Compiler sowie der GNU Compiler Suite verwendet wird. Verschiedene Sprachen wie Lisp, Java und Mono und Firmen wie Xerox und Amazon haben von den Ergebnissen des Papers profitiert.

Blacklisting wurde hin und wieder in der Literatur erwähnt, es fehlt den Autoren jedoch der Überblick, wieviele Implementationen davon inspiriert wurden.

Es gab eine Studie, die den Erfolg des Algorithmus analysiert, und zum Schluss kommt, dass in (ausgewählt ungünstigen Fällen) die nicht freigegebene, jedoch auch nicht mehr referenzierten Daten bis zu einem Faktor 2 ansteigen, sich jedoch langfristig keine Datenlöcher aufsummieren. Außerdem wurden in anderen weiterführenden Analysen das Problem, dass große Objekte durch Blacklisting nicht mehr allokieren zu können, bestätigt.

Theoretische Studien, wie man das Problem durch bessere Definition der Programmiersprache an der Wurzel tilgen könnte, wurden intensiviert und durch das Thesenpapier angeregt.

6 Zusammenfassung

Der analysierte Artikel ist ein klassisches Beispiel für ein praktisch ausgerichtetes Thesenpapier. Er behandelt eine Technik für die (1992 neuen) Garbage Collectors.

Das Interessante an diesem Artikel ist nicht nur, dass die vorgestellten Methoden gut und praktisch nutzbar sind, sondern dass Veröffentlichungen dritter, die Arbeit aufgegriffen und weiterentwickelt haben. Ebenso ist es faszinierend zu sehen, dass das erfundene Prinzip des Autors 20 Jahre später fast gleich im produktiven Einsatz ist.

Literatur

[1] Boehm, Hans-J.: Space Efficient Conservative Garbage Collection. In: ACM SIGPLAN 1993 PLDI.

[2] Boehm, Hans-J.: Space Efficient Conservative Garbage Collection. In: ACM SIGPLAN 2002, Best of PLDI 1979-1999.

[3] Artikel Automatische Speicherbereinigung. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 1. Dezember 2006, 13:51 UTC. URL: http://de.wikipedia.org/w/index.php?title=Automatische_Speicherbereinigung&oldid=24537938 (Abgerufen: 3. Dezember 2006, 15:36 UTC)