ABSCHLUSSBERICHT

MARA - MEMORY AND RESOURCE ALLOCATOR

PROJEKT: WEITERENTWICKLUNG EINES VERIFIKATIONSWERKZEUGES FÜR VERTEILTE SYSTEME

JULIAN GAEDE, TOM MEYER, MARIAN STEIN

FAKULTÄT FÜR INFORMATIK UND ELEKTROTECHNIK
LEHRSTUHL THEORETISCHE INFORMATIK
UNIVERSITÄT ROSTOCK

BETREUER:

PROF. DR. RER. NAT. HABIL. KARSTEN WOLF

28.3.2017

1

EINLEITUNG

Das Tool LoLA (a Low Level Petri Net Analyzer) bezieht Speicher direkt per Betriebssystemaufruf. Es galt herauszufinden, ob eine eigene Speicherverwaltung sowohl Speicheroverhead als auch Laufzeit einsparen kann. Im Laufe des Semesters haben wir eine solche entwickelt und mit dem Betriebssystemaufruf verglichen. Das Ergebnis zeigt, dass wir zwar bei statischen Speicheranfragen (Speicher, der zur Programmlaufzeit nicht wieder freigegeben wird) Einsparungen sowohl in Laufzeit als auch Speicheroverhead erzielen konnten, jedoch bei dynamischen Anfragen die Laufzeitnachteile den eingesparten Speicher überwogen. Als Konsequenz haben wir uns auf statische Anfragen spezialisiert und dadurch in beiden Kriterien weitere Verbesserungen erreicht.

2

ABLAUF UND BEARBEITUNG

2.1 PLANUNGSPHASE

Nach Besprechung der Aufgabenstellung mit Prof. Wolf haben wir zunächst gemeinsam in der Planungsphase zugrundeliegende Datenstrukturen sowie die Kodierung der Metainformationen der Speicherbereiche entwickelt.
Um den Speicheroverhead seitens des Betriebssystems zu minimieren, holen wir uns große Speicherbereiche (im Folgenden: Seiten) auf einmal über die Betriebssystemschnittstelle und verwalten diese intern. Hierbei werden dynamische Speicherbereiche vom Anfang der Seite an alloziert, wohingegen für statische Anfragen Bereiche vom Ende der Seite genutzt werden. Wenn auf den bestehenden Seiten kein ausreichend großer Freispeicherbereich vorhanden ist, um eine Anfrage zu erfüllen, wird eine neue Seite angelegt.
Die freien Speicherbereiche wurden anhand ihrer Größe nach Vorbild von Linux in Klassen (Buckets) eingeteilt. Innerhalb dieser Buckets lagen die verschiedenen Bereiche als verkettete Liste vor, wobei die Pointer in den Freispeicherbereichen selbst geführt wurden. Jede Seite verwaltet ihre Buckets in einem Array.
Zur Verwaltung der Metainformationen werden am linken und rechten Rand jedes (dynamischen) Speicherblocks mehrere Bytes reserviert, in denen die Größeninformation sowie die Belegung des umschlossenen Bereichs kodiert wird. Eine detaillierte Aufschlüsselung dieser Kodierung befindet sich in den Dokumentationskommentaren des Quellcodes. Statische Speicherbereiche benötigen keine solche Kodierung, da diese über die restliche Programmlaufzeit nicht mehr verändert werden.

2.2 IMPLEMENTIERUNGS- UND DEBUGPHASE

Zunächst wurde ein Prototyp von Tom entwickelt, welcher eine Implementierung der Metainformationen von Marian nutzte. Diese wurde anschließend durch ein Testverfahren, welches zufällige Speichergrößen anfordert, mehrfach zum Absturz gebracht. Anhand dieser Abstürze wurden in diversen gemeinsamen Debugsitzungen der Großteil der Fehler behoben. Während das Debuggen hauptsächlich von Tom und Marian betrieben wurde, entwickelte Julian das Testverfahren weiter und implementierte eine Integritätsprüfung, die weitere Fehler aufdeckte, indem überprüft wurde, ob Speicherbereiche illegalerweise überschrieben wurden oder Freispeicherbereiche nicht im richtigen Bucket eingetragen waren. Außerdem erweiterte er das Programm um eine Statistikfunktion, die Buch über die Speicheranfragen führte.

2.3 TEST- UND PERFORMANCEEVALUATIONSPHASE

Nachdem die meisten Fehler behoben waren, traten folgende Fehler erst so spät im Testablauf auf, dass der Arbeitsspeicher unserer Entwicklungsrechner nicht immer ausreichte, weshalb wir von Prof. Wolf Zugang zu einem leistungsfähigen Rechner erhielten. Durch ein von Julian entwickeltes Script konnten auf diesem mehrere deutlich aufwändigere Tests parallel durchgeführt werden, wobei noch mindestens ein Grenzfall entdeckt werden konnte.
Nachdem alle Tests durchliefen, wurde das Testverfahren um eine Messung der Prozessorzeit sowie einen direkten Vergleich mit dem Betriebssystemaufruf erweitert. Zunächst arbeitete unsere Bibliothek etwa halb so schnell wie das Betriebssystem, was uns dazu bewegte, nach Verbesserungen in den Datenstrukturen zu suchen.
Als möglichen Flaschenhals identifizierten wir die Organisation der Seiten, die bisher in einer verketteten Liste erfolgte, wodurch bei hoher Auslastung alle Seiten durchsucht werden mussten. Um dies zu vermeiden, schlossen wir die verkettete Liste zu einem Ring und begannen auf derjenigen Seite mit der Suche, auf der zuletzt erfolgreich eingefügt werden konnte. Eine neue Seite wurde angefordert, wenn der Ring erfolglos komplett durchsucht wurde.
Auf der Suche nach weiteren Optimierungen bemerkten wir, dass die Compileroptimierung nicht aktiviert war. Nach Behebung dieses Problems konnten wir auf einen Laufzeitfaktor von ca. 1,2 bei einer Verteilung von 80% statischen gegenüber 20% dynamischen Speicheranfragen gegenüber dem Betriebssystem aufschließen. Wenn nur statischer Speicher angefordert wurde, konnten wir sogar einen Faktor von 0,9 erreichen.
Nach Rücksprache mit Prof. Wolf wurde entschieden, uns auf die Verwaltung von statischem Speicher zu beschränken und den dynamischen Speicher weiterhin durch das Betriebssystem verwalten zu lassen, um eine maximale Laufzeitverbesserung zu erzielen.

2.4 NEUIMPLEMENTIERUNGSPHASE

Mit dem Wissen, dass einmal angeforderter Speicher erst beim Programmende freigegeben wird, konnten wir das Programm in einer gemeinsamen Sitzung auf unter 100 Zeilen reduzieren. Nachdem wir eine Seite geholt haben, benötigen wir im Wesentlichen lediglich einen Pointer, der auf den Anfang des freien Bereichs der Seite zeigt. Um eine Anfrage zu bearbeiten, wird überprüft, ob noch ausreichend Platz auf der Seite vorhanden ist. Wenn dies der Fall ist, wird dieser "Füllstands-Pointer" zurückgegeben und entsprechend der angefragten Größe verschoben. Ansonsten wird eine neue Seite angelegt und der Rest der alten Seite an das Betriebssystem zurückgegeben.

3 FAZIT UND FUTURE WORKS

Während der Implementierung der ersten Version, die dynamische Speicheranfragen ermöglichte, haben wir durch geschickte Kodierung der Metainformationen den Speicheroverhead nicht unerheblich reduzieren können. Bei dynamischen Aufrufen benötigen wir bei Blockgrößen von bis zu $2^{14} - 1$ Byte lediglich 2 Byte Verwaltungsinformation an jedem Ende des Blocks. Der Betriebssystemaufruf benötigt in den meisten Fällen 4 bis 8 Byte.
Der Geschwindigkeitsunterschied zum Betriebssystem ist vermutlich dadurch zu erklären, dass die Betriebssystemschnittstelle im Kernel-Modus arbeitet und hierdurch Spezialbefehle des Prozessors ausnutzen kann.
Um Laufzeitverbesserungen in dieser Version zu erzielen, könnte man die Datenstrukturen zur Freispeicherverwaltung genauer untersuchen und ggf. ersetzen. So ist beispielsweise denkbar, die Buckets ab einer ausreichenden Größe (in unserem Fall 12 Byte) global und nicht pro Seite zu führen. Außerdem enthalten gerade die späteren Buckets eine große Bandbreite von Speichergrößen, sodass unter Umständen die verkettete Liste weit durchsucht werden muss. Je nach Anwendungsfall könnte man dort etwa auf B-Bäume oder vergleichbare Datenstrukturen zurückgreifen, da der Freispeicher groß genug ist, um darin komplexe Strukturen zu implementieren. Es bleibt abzuwägen, ob der Mehraufwand beim Einfügen in einen Baum die Zeitersparnis beim Suchen nicht überwiegt.

Die zweite Variante, die nur statische Speicheranfragen unterstützt, spart sich die Verwaltungsinformationen komplett, sodass zusätzlich zu den des Betriebssystems lediglich Pointer auf Anfang und Ende der Seite sowie auf den Anfang des freien Bereichs gepflegt werden müssen. Aufgrund des einfachen Ablaufs ist vermutlich nicht viel Optimierungspotential vorhanden.
Die beigefügten Beispieldaten zeigen die Ergebnisse einer Testreihe, wobei in jedem Test 80 Millionen Anfragen mit gleichverteilter Größe innerhalb des angegebenen Intervalls durchgeführt wurden. In diesen konnte im Mittel eine Laufzeit von 54,6% gegenüber dem Betriebssystem festgestellt werden. Die Wahl der Seitengröße scheint die Laufzeit in diesen Tests nicht wesentlich beeinflusst zu haben, was jedoch darauf zurückzuführen ist, dass ein Seitenwechsel im Vergleich zum Rest der Tests sehr wenig Zeit in Anspruch nimmt. Außerdem beeinflusst die Seitenanzahl nicht wie in der ersten Version die Suchgeschwindigkeit, da zu jedem Zeitpunkt nur eine Seite verwaltet wird und volle Seiten einfach "vergessen" werden. Auch die angefragte Größe hat nur wenig Einfluss auf die Laufzeit. Vier der fünf Bereiche erreichen im Schnitt einen Wert von 53%, der Bereich zwischen 4 und 1000 Byte stellt jedoch mit einem Mittelwert von 61,5% einen uns nicht erklärbaren Ausreißer.

Wir kommen zu dem Schluss, dass Optimierungen bei der Speicherallokation nur praktikabel sind, wenn aufgrund der konkreten Anwendung Einschränkungen gemacht werden können, und dass es ohne Kernelmodus sehr schwer ist, die Funktion des Betriebssystems in Hinblick auf die Verwaltung von dynamischen Speicher so zu ersetzen, dass eine Laufzeitverbesserung auftritt.

4

ANHANG: TESTDATEN

Page Size	Requested Sizes	Factor of Malloc	Average Pageload
128 MB	4 - 32 B	0.544666	0.94825511
128 MB	4 - 1000 B	0.63806	0.99972053
128 MB	4 - 4000 B	0.513069	0.99964231
128 MB	100 - 500 B	0.542068	0.99728917
128 MB	1000 - 4000 B	0.524987	0.99984453
256 MB	4 - 32 B	0.543755	0.86923382
256 MB	4 - 1000 B	0.598739	0.99639061
256 MB	4 - 4000 B	0.504249	0.99914974
256 MB	100 - 500 B	0.533419	0.99174989
256 MB	1000 - 4000 B	0.510128	0.99985491
512 MB	4 - 32 B	0.541564	0.86923389
512 MB	4 - 1000 B	0.620182	0.99639186
512 MB	4 - 4000 B	0.525197	0.9974839
512 MB	100 - 500 B	0.548342	0.99175053
512 MB	1000 - 4000 B	0.527066	0.99851984
1024 MB	4 - 32 B	0.545349	0.65192546
1024 MB	4 - 1000 B	0.655981	0.98328202
1024 MB	4 - 4000 B	0.516297	0.99416146
1024 MB	100 - 500 B	0.541459	0.97019108
1024 MB	1000 - 4000 B	0.521738	0.99585259
2048 MB	4 - 32 B	0.538169	0.65192546
2048 MB	4 - 1000 B	0.618211	0.98328238
2048 MB	4 - 4000 B	0.498989	0.99416267
2048 MB	100 - 500 B	0.512773	0.92976658
2048 MB	1000 - 4000 B	0.52584	0.99055679
4096 MB	4 - 32 B	0.471802	0.32596266
4096 MB	4 - 1000 B	0.559667	0.93411839
4096 MB	4 - 4000 B	0.590717	0.98108209
4096 MB	100 - 500 B	0.515233	0.92976671
4096 MB	1000 - 4000 B	0.550995	0.99055745