Semesterarbeit

Concurrent Garbage Collector

Dozent:

Tomas Pospisek
Hochschule für Technik Zürich
Lagerstrasse 41
Postfach 1568
8021 Zürich

Autor:

Daren Thomas
Kleinalbis 80
8045 Zürich
dthomas@hsz-t.ch
+41 43 333 13 32

Zürich, November 2007

Inhaltsverzeichnis

1 Einleitung ... 3
1.1 Motivation ... 3
1.2 Ziele ... 3
2 Vorgehen ... 3
2.1 Erreichen des Hauptziels ... 3
2.2 Erreichen des Sekundärziels ... 4
2.3 Erreichen der persönlichen Ziele ... 4
3 Lösungsschritte ... 4
4 Beschreibung der Lösung ... 5
4.1 Die Programmiersprache Sina ... 5
4.2 Testalgorithmus ... 6
4.3 Parser/Lexer ... 7
4.4 Virtuelle Maschine ... 7
4.5 Interpreter ... 8
4.6 Garbage Collectoren ... 9
5 Dateistruktur ... 9
6 Die Garbage Collection Algorithmen ... 11
6.1 Two Space Copy ... 11
6.2 Concurrent Garbage Collector ... 12
6.2.1 Synchronisation des Farbtausches und die Free List ... 13
6.2.2 Annahmen ... 14
7 Probleme bei der Umsetzung ... 14
8 Vergleich TwoSpace vs. Concurrent ... 15
8.1 10'000 Zeilen mit je 10 Zeichen mit zwei Prozessoren ... 16
8.2 10'000 Zeilen mit je 500 Zeichen auf zwei Prozessoren ... 17
8.3 1'000 Zeilen mit je 10 Zeichen auf zwei Prozessoren ... 18
8.4 10'000 Zeilen mit je 10 Zeichen auf einem Prozessor ... 19
8.5 10'000 Zeilen mit je 500 Zeichen auf einem Prozessor ... 20
8.6 Interpretation und Schlussfolgerung ... 20
9 Literaturverzeichnis ... 21
10 Funktionsreferenz der Programmiersprache Sina ... 21

1 Einleitung

1.1 Motivation

Während der Vorlesung “Systemnahe Software” des Wintersemesters 2006 / 2007 hat Tomas Pospisek einen Artikel von Professor E. W. Dijkstra präsentiert, in dem ein nebenläufiger Garbage Collector (Concurrent Garbage Collector) vorgestellt wurde. Herr Pospisek fragte die Klasse, warum dieser Algorithmus in industriellen Programmiersprachen und deren Runtimes nicht verwendet wird und stattdessen der TwoSpace Algorithmus und andere, “Halt the world” Varianten verwendet werden. Ich hätte da gerne eine Antwort gegeben, konnte aber keine finden. Entweder, dachte ich mir, ist dieser Artikel nicht an die richtigen Leute gelangt (eher unwahrscheinlich) oder die Implementierung (auf die Professor Dijkstra verzichtete) ist nicht möglich oder es lohnt sich nicht. Da wir jetzt aber eine neue Ära beginnen, in der Heimcomputer und Laptops zusehends mit Mehrprozessorarchitekturen bestückt sind, dachte ich mir, es könnte Zeit sein, diese Frage von Neuem anzugehen.

Diese Aufgabe schien auch eine gute Gelegenheit zu sein, eine Programmiersprache, deren Grundprinzipien mir schon lange im Kopf herumschwirrten, genau zu definieren, zu implementieren und nach meiner Frau Sina zu nennen.

1.2 Ziele

Das Hauptziel dieser Arbeit besteht in einer funktionierenden Implementation des Algorithmus von Dijkstra für nebenläufige Garbage Collection. Falls der Algorithmus nicht implementierbar sein sollte, sollen die Gründe dafür erforscht werden.

Das Sekundärziel dieser Arbeit ist ein Vergleich der beiden Algorithmen TwoSpace und Concurrent. Hier sollen Erklärungen für die Vorliebe der Industrie, TwoSpace zu implementieren, gefunden werden.

Ein persönliches Ziel ist, Erfahrungen mit C, lex und yacc zu sammeln und so einerseits meine Unsicherheit im Umgang mit Pointern zu verlieren und ausserdem vertiefte Kenntnisse im Umgang mit Parsergeneratoren zu erlangen.

Ein weiteres persönliches Ziel ist, eine (einfache) Programmiersprache zu erfinden und zu implementieren.

2 Vorgehen

2.1 Erreichen des Hauptziels

Um das Hauptziel, den Concurrent Garbage Collector zu implementieren, zu erreichen, sind diverse Vorbedingungen zu erfüllen:

- Es muss eine Runtime vorhanden sein, welche auf einem Heap Speicher alloziiert.
- Die Funktionsweise dieser Runtime muss genau bekannt sein, damit man den Algorithmus vollständig umsetzen kann.
- Ausserdem muss ein Problem vorliegen, das mit dieser Runtime zu lösen ist, welches

genügend Allokationen macht, um die Wirkung des Garbage Collectors zu testen.

Der Algorithmus von Dijkstra bezieht sich auf einen LISP Interpreter. Leider weiss ich über LISP herzlich wenig und wollte nicht die notorisch lange Einarbeitungszeit in diese Sprache und dann, darauf aufbauend, in den Interpreter auf mich nehmen. Ich habe mir aber schon seit längerer Zeit Gedanken über eine minimale stackorientierte Programmiersprache und dazugehöriger Runtime gemacht, welche in ihrem Aufbau ähnlich wie LISP funktioniert, aber auf imperative Weise Probleme abarbeitet. Dies schien mir für die Implementation des Concurrent Garbage Collectors geeignet.

2.2 Erreichen des Sekundärziels

Um das Sekundärziel zu erreichen, TwoSpace und Concurrent Garbage Collector zu vergleichen, brauchte ich einen Testalgorithmus, der mit der Runtime und jeweils dem einen oder anderen Allokator laufen sollte, welcher möglichst ohne Änderung am Testalgorithmus dazu gebracht werden kann, auf Wunsch wenig oder viel Speicher zu verwenden. Der Testalgorithmus sollte also abhängig vom Input mehr oder weniger Speicher aufs Mal verwenden und länger oder weniger lang laufen.

2.3 Erreichen der persönlichen Ziele

Die persönlichen Ziele sollen durch Festlegung der Programmiersprache für die Arbeit auf ANSI C und Entwicklung eines Parsers für den Inpterpreter mittels flex und bison (GNU Varianten der Tools lex und yacc) und der Definition der Programmiersprache Sina erreicht werden.

3 Lösungsschritte

Da diese Arbeit aus lauter Unbekannten bestand, von denen ich die Entwicklungszeit nicht vorhersagen konnte, entschloss ich mich, die Programmierung explorativ vorzunehmen, indem ich jeweils einen Schritt nach dem anderen implementierte. Folgende Schritte drängten sich auf:

1. Definition der Programmiersprache (Syntax und Semantik) im Hinblick auf die einfache Implementierung eines Interpreters dafür.
2. Finden eines geeigneten Testalgorithmus für die zwei Garbage Collection Algorithmen und Implementation in der Programmiersprache Sina.
3. Entwicklung eines Parsers für die Programmiersprache Sina. Der Parser muss den Testalgorithmus parsen können und eine Repräsentation des Programms im Speicher aufbauen. Dafür muss die Schnittstelle für Speicheralloziierung festgelegt und mit einem Dummy-Allokator implementiert werden.
4. Implementieren des Interpreters mit dem Dummy-Allokator und testen des Testalgorithmus (Debugging des Interpreters und des Testalgorithmus).
5. Implementieren des TwoSpace Garbage Collectors als einfachere Variante.
6. Implementieren des Concurrent Garbage Collectors.
7. Vergleich der beiden Garbage Collectoren.
8. Dokumentation der Resultate.

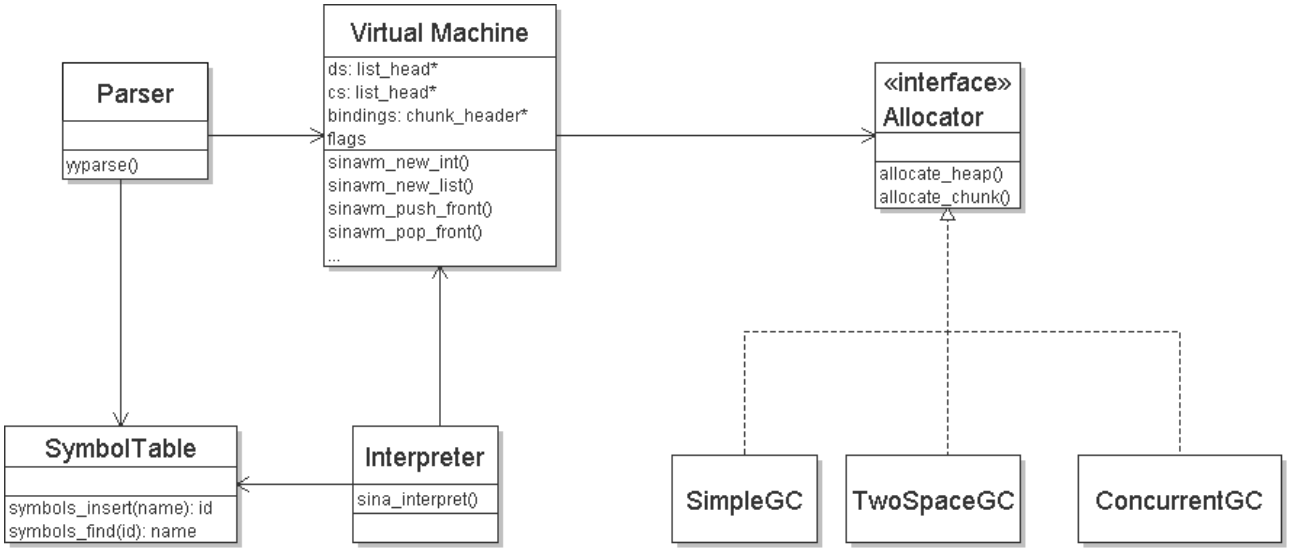
Da im Voraus wenig über die Lösung bekannt war, konnte ich jeweils nicht weiter als bis zum

nächsten Schritt planen. So hatte ich zum Beispiel keine Ahnung, wie ein Parser aussieht und wie er sich in den Interpreter einbauen lässt. Ich liess also die Struktur der Applikation einerseits aus den Vorgaben der verwendeten Tools (vor allem des Parsers) und den einzelnen zu erreichenden Phasen wachsen, andererseits definierte ich schon früh die Schnittstelle, über die die einzelnen Komponenten miteinander sprechen sollten. Diese Schnittstelle wurde minimal gehalten, was in einer späteren Phase dazu führte, dass nachträglich via Hooks gewisse Funktionalitäten ermöglicht werden mussten. Dies ist zwar auf den ersten Blick unschön, war aber kaum vorhersehbar und stört die Architektur kaum.

4 Beschreibung der Lösung

Die Lösung besteht im Wesentlichen aus folgenden Teilen: Der Programmiersprache Sina, dem Testalgorithmus, einem Parser/Lexer für die Programmiersprache Sina, einer Virtuellen Maschine, einem Interpreter, welche die Repräsentation des Testalgorithmus in der virtuellen Maschine ausführt und einem Satz Garbage Collectoren, welche die Speicherverwaltung des Heaps übernehmen.

Die Beschreibung der Lösung folgt zugleich auch der zeitlichen Abfolge der Entwicklung der einzelnen Komponenten, da diese einander bedingen und aufeinander aufbauen.



4.1 Die Programmiersprache Sina

Die Programmiersprache Sina wurde im Hinblick auf einfache Implementierung und Interpretierbarkeit entworfen. Im Prinzip ist es eine Art Assembler für die Sina Virtual Machine. In der Programmiersprache Sina wird kein Unterschied zwischen Daten und Programmcode gemacht. Das bedeutet, ein Programm ist eine Liste von Daten (Objekten), die erst durch die Interpretation eine Bedeutung erhalten.

Die Syntax der Programmiersprache Sina wird durch folgende EBNF gegeben:

```
program := list_elements
list_elements := | list_elements list_element
list_element := integer | symbol | escaped_symbol
               | string | list | block
escaped_symbol := ':' symbol
list := '(' list_elements ')'
block := '{' list_elements '}'
```

wobei *integer* eine Kette von mindestens einer Ziffer oder ein einzelnes Zeichen in Hochkommas ist und *symbol* eine Kette von Buchstaben, Ziffern, dem Minuszeichen und dem Unterstrich ist, welche mit einem Buchstaben beginnt. Ein *string* ist eine Kette von Zeichen, die in Anführungszeichen steht. Innerhalb eines *strings* dürfen keine Anführungszeichen auftreten. Ein *string* wird intern als Liste von Zahlen repräsentiert.

Die Programmiersprache Sina ist case sensitive, Gross- und Kleinschreibung wird also unterschieden.

Die Semantik der Programmiersprache Sina ist eng mit der Runtime verknüpft: Die Sina Virtual Machine enthält zwei Listen: den Code Stack und den Data Stack. Nach dem Einlesen des Programms befinden sich die Elemente des Programms als Block auf dem Code Stack. Jedes Element wird, basierend auf seinem Typ, sequentiell vom Interpreter interpretiert.

Um die Programmiersprache definieren zu können, musste ich natürlich die Virtual Machine und den Interpreter in Ansätzen entwerfen. Dabei habe ich allerdings nur den groben Aufbau und den Ablauf der Interpretation definiert. Grosse Teile dieser Arbeit skizzierte ich schon im Vorfeld der Arbeit auf Servietten und in Tagträumen, sodass ich mich weniger mit der Programmiersprache und dafür umso mehr mit den Garbage Collection Algorithmen befassen konnte.

4.2 Testalgorithmus

Der gewählte Testalgorithmus für die Garbage Collectoren ist der Cesar Shift: Jeder Buchstabe in der Eingabe wird einem Buchstaben in der Ausgabe zugeordnet, der auf den Buchstaben der Eingabe im Alphabet folgt, wobei 'z' 'a' zugeordnet wird (modulo shift). Der Einfachheit halber werden alle Buchstaben als Grossbuchstaben betrachtet. Alle anderen Zeichen werden unverändert ausgegeben.

Dieser Algorithmus ist für das Testen der Garbage Collectoren besonders geeignet, weil die Durchlaufzeit des Algorithmus von der Eingabelänge (linear) abhängt und der Speicherverbrauch (unter der Bedingung, dass die Eingabe Zeilenweise eingelesen wird) von der Zeilenlänge der Eingabe bestimmt wird.

Für die Tests wird von zwei Dimensionen des Speicherverbrauchs ausgegangen: Einerseits der statische Speicherverbrauch (maximaler Speicher, welcher der Algorithmus zu einem bestimmten Zeitpunkt verwendet) und andererseits der dynamische Speicherverbrauch (Allokation und Freigabe von Speicher, also der Verschiebung des statischen Speichers innerhalb des Heaps mit der Zeit).

Der statische Speicherverbrauch lässt sich mit dem Cesar Shift über die Zeilenlänge steuern. Bei grossen Zeilenlängen kann getestet werden, wie gut der zu testende Garbage Collector mit knappem Speicher auskommt.

Der dynamische Speicherverbrauch lässt sich mit dem Cesar Shift über die Anzahl Zeilen steuern – je länger die Eingabe, desto mehr Collection Cycles werden benötigt, um die ganze Eingabe zu verarbeiten. Mit einer grossen Anzahl Zeilen lässt sich messen, wie schnell ein Garbage Collector arbeitet.

4.3 Parser/Lexer

Parser und Lexer wurden mit bison und flex realisiert. Diese Tools generieren anhand einer EBNF einen Parser und binden C Code für jedes erkannte Element ein. Die Programmiersprache Sina wurde auf einfaches Parsen hin entworfen. Ferner wurde darauf geachtet, dass der Unterschied zwischen Sina Quellcode und der Repräsentation des Algorithmus in der Virtual Machine möglichst klein ist: Die Sina Programmiersprache ist eine Notation für die Datenstrukturen, welche vom Sina Interpreter interpretiert werden. Sie ist also eine Art Assembler für die Sina Virtual Machine.

Nach Fertigstellung des Parsers konnten erste Syntaxprüfungen für den Testalgorithmus gemacht werden. Auf diese Art wurden auch ein paar Fehler im Testalgorithmus gefunden und behoben.

Es stellte sich heraus, dass eine wichtige Komponente fehlte: Ein Pretty Printer musste geschrieben werden, welcher die geparste Struktur auf den Bildschirm ausgibt, damit die Korrektheit des Parsers überprüft werden kann. Dieser Pretty Printer wurde dann später beim Debugging des Interpreters sowie der Garbage Collectoren wieder verwendet, da sich damit der Zustand der Sina Virtual Machine gut anzeigen lässt.

Der Parser arbeitet auch mit einer Symbol Table zusammen, welche alle auftretenden Symbole indiziert und zwischen den beiden Repräsentationen (nummerischer Index / alphanummerischer Bezeichner) übersetzen kann.

4.4 Virtuelle Maschine

Die Grundzüge für die Sina Virtual Machine sind eng mit der Programmiersprache Sina verknüpft: Eine Liste von Symbolen und anderen Daten soll interpretiert werden, indem zwei Stacks, dem Data Stack (ds) und dem Code Stack (cs) manipuliert werden.

Die Virtuelle Maschine besteht also aus den zwei Stacks, die, um möglichst alle Elemente der Lösung mit den Datenstrukturen der Programmiersprache Sina zu implementieren, als einzeln verkettete Listen implementiert sind (dies ist der gleiche Typ, der auch für Listenkonstrukte innerhalb von Sina Programmen verwendet wird). Ferner besitzt die Virtuelle Maschine eine Liste von Symbolen.

Die Virtuelle Maschine kennt folgende Datentypen (definiert in sina_types.h):

- INTEGER: Eine ganze Zahl. Daten werden mithilfe von diesem Datentyp angegeben. INTEGERs dienen auch zur Repräsentation von Zeichen.
- SYMBOL: Der nummerischer Index eines Symbols.
- ESCAPED_SYMBOL: Ein Datentyp, welches eigentlich ein Symbol repräsentiert, aber eine andere Laufzeitsemantik erfordert.
- LIST_HEAD: Einer von zwei Datentypen, mit denen Listen aufgebaut werden. Listen sind zentral für die Sina Virtual Machine und werden unter anderem auch für die Implementation des Data Stacks und des Code Stacks verwendet. Ein LIST_HEAD repräsentiert eine ganze Liste, indem er auf den ersten LIST_NODE und den letzten LIST_NODE verweist.
- LIST_NODE: Der andere der zwei Datentypen, mit denen Listen aufgebaut werden. Eine

LIST_NODE repräsentiert ein Element der Liste, indem sie auf einen anderen Datentyp (das Datum des Elements) und auf das nächste Element (LIST_NODE) der Liste verweist.

- BLOCK: Ein Block fügt zu Listen eine zusätzliche, ausführbare Semantik hinzu, indem er auf eine Liste (LIST_HEAD) verweist, deren Elemente als Anweisungen zu interpretieren sind, und auf das aktuell zur Ausführung vorgesehene Element (LIST_NODE) der Liste verweist. Der Code Stack besteht ausschliesslich aus BLOCKs.
- NATIVE: Repräsentiert einen Verweis auf eine vordefinierte, eingebaute Funktion der Programmiersprache Sina (siehe sina_builtins.c). Mit Hilfe von NATIVE Elementen können Grundfunktionalitäten der Virtual Machine (Verwaltung der Stacks, I/O, Arithmetik, Iteration und Alternative) angesprochen werden.

4.5 Interpreter

Der Interpreter für die Programmiersprache Sina ist ziemlich einfach gestrickt: Jedes Datum in der Virtual Machine hat einen bestimmten Datentyp. Auf dem Code Stack befinden sich Blöcke (BLOCK), wobei vom obersten Block jeweils das nächste Element angeschaut und interpretiert wird. Wurde das letzte Element eines Blockes interpretiert, wird der Block vom Code Stack entfernt. Die Verarbeitung ist beendet, wenn der letzte Block vom Code Stack entfernt wurde.

Dabei wird jedes Element eines Blockes wie folgt (gemäss seinem Datentyp) interpretiert:

- ein INTEGER wird interpretiert, indem es auf den Data Stack gelegt wird.
- ein SYMBOL wird interpretiert, indem das verwiesene Element interpretiert wird (ist ein SYMBOL mit einem NATIVE verknüpft, wird dieses ausgeführt, ist es aber mit einem INTEGER verknüpft, wird es auf den Data Stack gelegt). SYMBOLs, welche auf einen BLOCK verweisen, sind speziell: Hier wird nicht der BLOCK interpretiert (dies würde dazu führen, dass der entsprechende BLOCK auf den Data Stack gelegt wird), sondern der BLOCK wird auf den Code Stack gelegt und somit ausgeführt.
- ein ESCAPED_SYMBOL wird interpretiert, indem das entsprechende SYMBOL auf den Data Stack gelegt wird.
- ein LIST_HEAD wird interpretiert, indem er auf den Data Stack gelegt wird.
- eine LIST_NODE kann nicht interpretiert werden. Listen können nur als Ganzes betrachtet werden, wobei eingebaute Funktionen bestehen, um einzelne Elemente von einer Liste abzutrennen und neue Elemente hinzuzufügen. Ein Element einer Liste kann zwar selber eine Liste (also LIST_HEAD) sein, jedoch nie eine LIST_NODE. Dies kann mit der Programmiersprache Sina schlicht nicht ausgedrückt werden.
- ein BLOCK wird interpretiert, indem es auf den Data Stack gelegt wird.
- ein NATIVE wird interpretiert, indem die entsprechende Funktion ausgeführt wird.

Mit diesen einfachen Regeln und ein paar eingebauten Funktionen können im Prinzip alle Algorithmen beschrieben werden. Der Beweis hierzu fehlt zwar, allerdings konnte exemplarisch der Algorithmus Cesar Shift implementiert werden.

Um den Interpreter testen zu können, wurde ein Dummy-Allokator erstellt, welcher die Schnittstelle für Allokatoren (und damit auch Garbage Collectoren) testet, indem einfach für jede Speicherallozierung ein malloc() ausgeführt wird. Nach Fertigstellung des Interpreters und den im Testalgorithmus verwendeten eingebauten Funktionen konnte der Testalgorithmus erstmals ausgeführt werden. So wurden im Testalgorithmus weitere Fehler gefunden und behoben.

4.6 Garbage Collectoren

Um die Garbage Collectoren vom Interpreter zu trennen (nötig für die Vertauschbarkeit der Garbage Collectoren) wird eine einfache Schnittstelle verwendet: Die Virtuelle Maschine bietet Funktionen zur Allokation der verschiedenen Typen an. Diese wiederum rufen den Garbage Collector über die Funktion allocate_chunk() auf. Wie die Allokation gemacht wird, ist Sache der Implementierung des Garbage Collectors. Um die Heap Grösse festzulegen, wird vor dem ersten Aufruf von allocate_chunk() die Funktion allocate_heap() mit der gewünschten Heap Grösse aufgerufen.

5 Dateistruktur

In diesem Abschnitt folgt eine Auflistung der Dateien, die zum Quellcode dieser Arbeit gehören. Der grösste Teil dieser Dateien liegt als Quellcode in der Programmiersprache C vor. C kennt sowohl Source- wie auch Headerdateien. In den Headerdateien werden Funktionen deklariert, die dann in den Sourcedateien definiert werden. Jede Funktion ist mit einem Blockkommentar versehen, welcher die Funktion, deren Begründung und eventuell die Funktionsweise beschreibt. Um die SPOT (Single Point Of Truth) Regel nicht zu verletzen, wird jeweils nur die Funktionsdeklarierung (in der Headerdatei) kommentiert. Eine Kopie der Funktionsbeschreibung in der gleichnamigen Sourcedatei würde schnell zu inkonsistenten Beschreibungen zwischen Header- und Sourcedatei führen.

Folgende Dateien sind Bestandteil der Lösung:

Dateiname	Beschreibung
Makefile	Das Makefile enthält die Regeln für die Kompilation des Interpreters. Über verschiedene Targets kann der gewünschte Allokator/Garbage Collector gewählt werden. Ausserdem können mit dem Target “clean” Abfallprodukte der Kompilation (*.o, die ausführbare Datei und die Kompilate des Parser Generators bison) wieder entfernt werden. Für die Garbage Collectoren stehen die Targets “simple”, “twospace” und “concurrent” zur Verfügung. Der Default ist “concurrent”. Das Makefile beruht auf GNU make und verwendet einige dessen Erweiterungen. Unter anderem kann beim Aufruf die Variable HEAP_SIZE (Default: 1024) angepasst werden.
cesar_shift.sina	Enthält den Testalgorithmus in der Programmiersprache Sina.
concurrent_allocator.c/.h	Implementiert die Schnittstelle in sina_allocator.h für den Concurrent Garbage Collector.
main.c	Startet den Parser mit dem ersten Argument als Sourcedatei. Weist den Allokator/Garbage Collector an, sich mit einer bestimmten Heap Grösse zu initialisieren und startet den Interpreter.
pprinter.c/.h	Formatiert Sina Objekte und gibt sie auf den Bildschirm aus. Diese Funktionalität wird für das Debugging des Interpreters verwendet und kann mit dem Sina Befehl trace-on eingestellt werden.
simple_allocator.c	Ein einfacher Allokator, der keine Garbage Collection betreibt: Jeder angeforderte Chunk wird mit malloc() alloziiert.
sina_allocator.c/.h	Beschreibt die Schnittstelle für die drei Allokatoren “simple”, “twospace” und “concurrent”. Diese Schnittstelle besteht aus allocate_heap() und allocate_chunk(). Ausserdem implementiert sina_allocator.c den Register Stack(siehe Kapitel Probleme).
sina_builtins.c	Enthält die Liste aller eingebauten Funktionen der Programmiersprache Sina und verknüpft sie mit den entsprechenden Symbolen während der Initialisierung der Virtual Machine.
sina_error.c/.h	Implementiert die Ausgabe von Fehlermeldungen und der daraus resultierenden Verarbeitungsabbrüchen.
sina_interpreter.c/.h	Implementiert den Interpreter für die Programmiersprache Sina. Führt die Sina Virtual Machine iterativ in den jeweils nächsten Zustand, bis die Verarbeitung beendet ist. Dazu werden die Elemente im Code Stack interpretiert.
sina_lexer.l	Definiert den Lexer für die Programmiersprache Sina.
sina_parser.y	Definiert den Parser für die Programmiersprache Sina sowie die Aktionen, die beim Erkennen der einzelnen Konstrukte auszuführen sind, damit am Ende des Parsens eine interne Repräsentation des Programms vorhanden ist und mit der Interpretation begonnen werden kann.
sina_symbols.c/.h	Implementiert die Symboltabelle für den Parser und den Interpreter.
sina_types.c/.h	Definiert die verwendeten Datenstrukturen für die verschiedenen

Dateiname	Beschreibung
	Chunk Typen (Integer, Symbol, Listenkopf etc.) sowie die Datenstruktur für die Virtual Machine.
sinavm.c/.h	Implementiert die Grundfunktionen der Virtual Machine, mit denen der Zustand der Virtual Machine verändert werden kann. Ein Grossteil dieser Funktionen dient der Erstellung und Manipulation von Listen, da die Virtual Machine selber zwei Listen (Code Stack und Data Stack) verwaltet und Listen eine zentrale Datenstruktur für die Programmiersprache Sina darstellen.
testharness.py	Führt die Testreihe mit variierender Heap-Grösse, Inputdatei und Garbage Collector aus und gibt die Laufzeit aus.
twospace_allocator.c	Implementiert die Schnittstelle in sina_allocator.h für den TwoSpace Garbage Collector.

6 Die Garbage Collection Algorithmen

Garbage Collectoren basieren auf der Idee, dass alle zu einem bestimmten Zeitpunkt für ein Programm relevanten Daten über das sogenannte Root Set erreichbar sind. Speicher, der nicht über das Root Set erreichbar ist, kann vom Programm nicht mehr manipuliert werden und darf daher wiederverwendet werden. Die Aufgabe eines Garbage Collectors ist es, solche nicht erreichbaren Speicherplätze aufzuspüren und für neue Speicherallokationen zur Verfügung zu stellen. Im Rahmen dieser Arbeit werden zwei unterschiedliche Strategien untersucht: Two Space Copy und der Concurrent Garbage Collector von Dijkstra.

Garbage Collectoren arbeiten mit dem Allokator zusammen in dem Sinne, dass beide den Heap verwalten. Im Rahmen dieser Arbeit wird der Heap, der dem Programm zur Verfügung steht, als begrenzt betrachtet. Die Sina Virtual Machine interagiert über eine allgemeine Schnittstelle mit den Garbage Collectoren, wobei die Schnittstelle eigentlich nur die Allokation eines Speicherplatzes beschreibt.

Sowohl Allokatoren wie auch Garbage Collectoren arbeiten nicht mit einzelnen Bytes im Speicher, sondern mit sogenannten Chunks, welche alloziierte Stücke des Speichers beschreiben und enthalten: Ein Chunk enthält neben dem zu alloziierenden Speicher zusätzlich Metadaten über diesen Speicher, z.B. die Grösse, oder (wie bei der Sina Virtual Machine) den Typ.

6.1 Two Space Copy

Der Two Space Copy Algorithmus teilt den Heap in zwei gleich grosse Hälften, wovon zu einem gegebenen Zeitpunkt jeweils nur eine Hälfte aktiv ist. Für jede Speicherallokation wird ein Zeiger auf den nächsten freien Speicherplatz um die Grösse des alloziierten Speichers verschoben. Falls die aktive Hälfte des Heaps nicht genügend Platz für die Speicheralloziierung hat, wird das Root Set durchlaufen und jedes erreichbare Element wird in die inaktive Hälfte des Heaps verschoben. Zeiger innerhalb der verschobenen Chunks werden auf die entsprechenden, ebenfalls verschobenen Chunks umgelenkt. Anschliessend wird die inaktive Hälfte aktiviert und die bis anhin aktive Hälfte deaktiviert. Bei diesem Vorgang werden nicht mehr erreichbare Chunks nicht mitkopiert und die Alloziierung kann fortgeführt werden.

Für den Two Space Copy spricht die einfache Implementierung und die Tatsache, dass Memory Compaction automatisch erfolgt.

6.2 Concurrent Garbage Collector

Der Concurrent Garbage Collector, wie er im Artikel von Dijkstra beschrieben wird, geht von zwei separaten Prozessen aus: Dem Mutator und dem Collector. Der Mutator arbeitet auf den Daten des ausgeführten Algorithmus (also dem Root Set), während der Collector den Heap nach Chunks durchsucht, die nicht Teil des Root Sets sind und somit nicht mehr gebraucht werden. Dabei müssen sich die zwei Prozesse untereinander koordinieren: Der Collector muss sicherstellen, dass ein Chunk, den er für frei befunden hat, in der Zwischenzeit nicht doch wieder in den Root Set aufgenommen wurde. Dies ist zwar mit sogenannten kritischen Abschnitten (mit Verwendung von Mutexes) erreichbar, führt aber zu einem gewissen Overhead, weil die einzelnen Prozesse je nach Granularität des Locking aufeinander warten müssen. Dijkstra schlägt einen Algorithmus vor, der mit Dirty Reads, also der Möglichkeit, dass der Collector alte Daten liest, umgehen kann und somit kein Locking benötigt:

- Jeder Chunk hat eine Farbe: Grün, Weiss, Grau oder Schwarz.
- Alle Chunks, die vom Root Set aus erreichbar sind, sind entweder weiss, grau oder schwarz.
- Eine Liste mit Chunks, die zur Alloziierung verwendet werden können, enthält grüne Chunks. Diese Liste heisst die “free list”.
- Nur Chunks in der free list sind grün.
- Collector und Mutator färben Chunks, die vom Root Set aus erreichbar sind, ein, wobei immer in Richtung schwarz eingefärbt wird.
- Eingefärbt wird immer nur eine Stufe aufs Mal, wobei die Reihenfolge Grün/Weiss → Grau → Schwarz strikt befolgt wird.
- Ein schwarzer Chunk verweist zu keinem Zeitpunkt auf einen weissen Chunk (Chunks, die Listen und Blocks aufbauen verweisen auf andere Chunks).
- Wenn der Mutator einen neuen Chunk zum Root Set hinzufügt (also einen grünen Chunk aus der free list entfernt), färbt er ihn dunkler (also grau). Ausserdem wird der Chunk, der anschliessend auf den neuen Chunk verweist, auch dunkler eingefärbt, sofern er weiss ist.
- Wenn der Mutator einen bereits vom Root Set aus erreichbaren Chunk umhängt (einen neuen Verweis macht), wird ein ähnliches Verfahren angewandt: Falls der neu verweisende Chunk weiss ist, wird er grau gemacht und, falls der umgehängte Chunk weiss ist, wird dieser ebenfalls grau.
- Damit der Mutator neue graue Chunks produzieren kann, muss also mindestens ein weisser Chunk vom Root Set aus erreichbar sein.
- Der Collector färbt das Root Set grau (im Falle der Sina Virtual Machine sind dies der Kopf des Data Stack und der Kopf des Code Stack).
- Anschliessend durchsucht der Collector den Heap nach grauen Chunks. Fall ein solcher gefunden wird und dieser Chunk auf andere Chunks verweist (LIST_HEAD, LIST_NODE und BLOCK verweisen auf andere Chunks), werden die verwiesenen Chunks grau gefärbt, falls sie weiss sind. Dann wird der graue Chunk schwarz gefärbt.
- Fall der Collector im ganzen Heap keine grauen Chunks mehr findet, sind alle erreichbaren

Chunks schwarz und **der Mutator kann keine grauen Chunks mehr erstellen**. Alle Chunks, die jetzt weiss sind, sind vom Root Set aus nicht erreichbar und können vom Collector eingesammelt, grün gefärbt und in die free list aufgenommen werden.

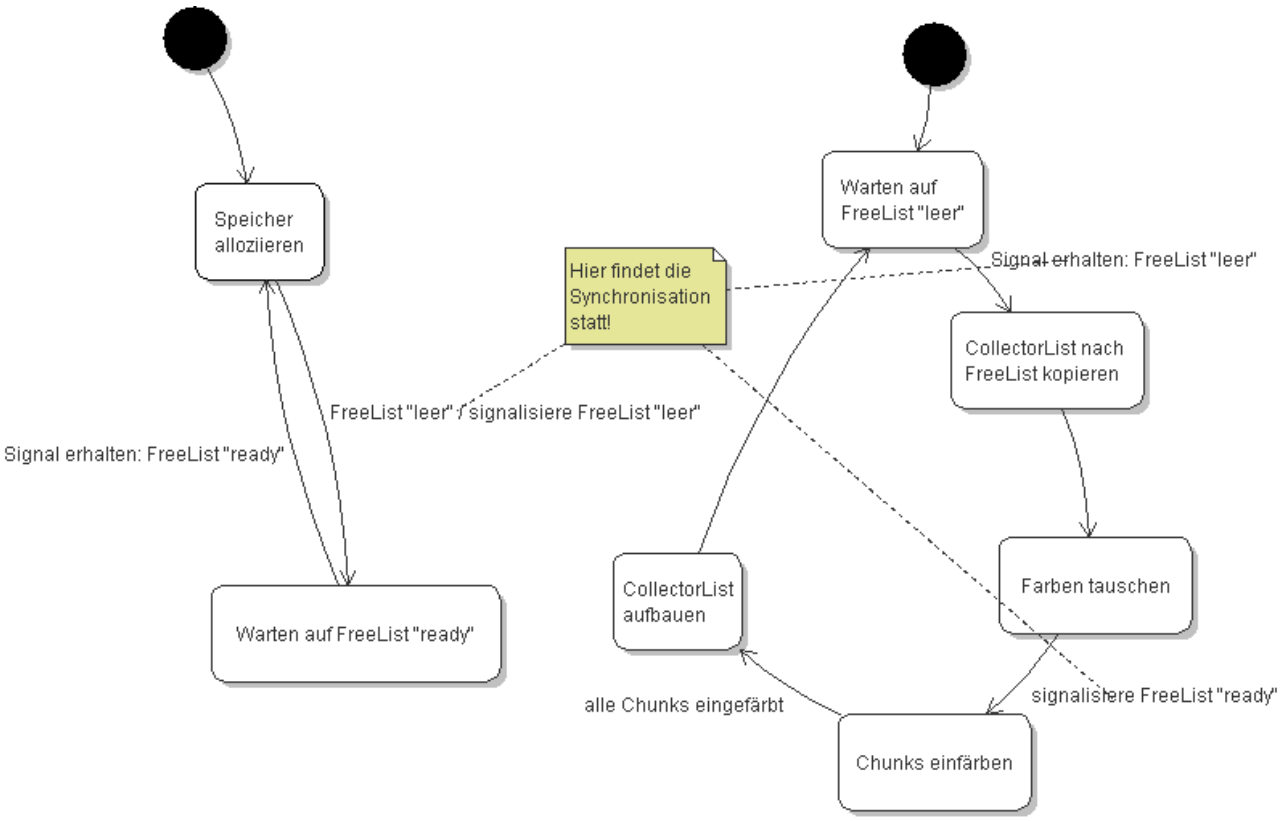
- Die Farben Schwarz und Weiss werden vertauscht. Dies muss zwischen Mutator und Collector synchronisiert werden, passiert allerdings nicht allzu oft.

6.2.1 Synchronisation des Farbtausches und die Free List

Dijkstra lässt ein paar Details explizit offen:

- Wie wird der Farbtausch synchronisiert?
- Wie wird die Free List so verwaltet, dass gleichzeitig der Mutator daraus Chunks entfernen und der Collector Chunks dazu fügen kann?

Ich habe für dieses beiden Probleme eine kleine Abwandlung des Algorithmus vorgenommen, welche beide Fragen gleichzeitig löst. Diese Idee beruht auf der Annahme, dass Parallelität viel einfacher zu erreichen ist, je weniger Daten miteinander geteilt werden: Statt nur einer Free List habe ich zwei implementiert. Der Mutator bedient sich wie gewohnt von der Free List, die beim Start der Sina Virtual Machine die Hälfte der Chunks im Heap enthält, während der Collector mit den restlichen Chunks (die sogenannten Collector List) darauf wartet, bis der Mutator meldet, dass die Free List keine Chunks mehr enthält. Dies geschieht zwingend nur beim Versuch des Mutators, einen Chunk zu alloziieren. Der Mutator wartet dann, bis der Collector ihm meldet, dass die Free List wieder gefüllt ist. Zuerst werden jedoch die Farben Weiss und Schwarz vertauscht (da, während der Mutator wartet, keine Chance besteht, dass dieser einen falschen Wert liest), anschliessend überweist der Collector dem Mutator die Collector List als Free List und legt eine neue (leere) Collector List an. Der Collector meldet dem Mutator, dass er weiter fahren kann und beginnt selber mit den oben beschriebenen Schritten. Sobald der Collector keine grauen Chunks mehr findet, werden die weissen Chunks in die Collector List aufgenommen, eine volle Collector List wird signalisiert und der Collector wartet auf den Mutator. Es ist allerdings möglich, dass der Mutator die Free List leert, bevor der Collector mit dem Aufbau der Collector List fertig ist. In diesem Falle wartet der Mutator bis der Collector die volle Collector List signalisiert und signalisiert erst dann, dass er auf die Free List wartet.



Eine Bedingung für den Concurrent Garbage Collector ist, dass alle Chunks gleich gross sind. Dies ist nötig, damit der Collector über alle Chunks iterieren kann.

6.2.2 Annahmen

- dieser Algorithmus wird auf einer 32 Bit Maschine ausgeführt. Eventuell kommen andere Architekturen in Frage, wichtig ist aber, dass sizeof(int) und sizeof(void*) gleich gross sind.
- Wird ein int oder ein void* gelesen, kann es zwar vorkommen, dass er inzwischen von einem anderen Prozess verändert wurde und der alte Wert gelesen wird, aber es kann nie ein ungültiger Zwischenwert gelesen werden, bei dem sich erst einige Bits geändert haben: Es wird immer ein disktreter Zustand dieses Wortes (32 Bit) gelesen, der irgendwann im Verlaufe der Berechnungen des Mutators diesen Wert hatte.

7 Probleme bei der Umsetzung

Durch die Art, wie der Sina Interpreter (insbesonders die eingebauten Funktionen der Programmiersprache Sina) funktioniert, befinden sich zu gewissen Zeiten immer wieder Chunks in einem Zustand, in dem sie zwar noch gültig sind, also in absehbarer Zukunft wieder in den Root Set aufgenommen werden, aber für eine Berechnung kurz aus dem Root Set entfernt werden mussten. Ein Beispiel dafür ist die Sina Funktion “add”. Diese wird so interpretiert, dass die zwei obersten Chunks vom Data Stack entfernt werden und ein neuer INTEGER_CHUNK mit dem Wert, der die Summe der Werte der zwei entfernten Chunks entspricht, auf den Data Stack gelegt wird. Dies bedingt einerseits die Allokation eines Chunks für das Resultat und andererseits die Allokation eines Chunks vom Typ LIST_NODE_CHUNK, mit dem das Resultat in den Data Stack eingefügt wird. Falls bei der Allokation des LIST_NODE_CHUNKs der Speicher voll ist, wird der TwoSpace Allgorithmus den Root Set in den Free Heap kopieren und dabei das Resultat (den

INTEGER_CHUNK) übersehen, da dieser noch nicht über das Root Set erreichbar ist.

Um diesem Effekt vorzubeugen wurde ein Register Stack eingeführt, auf das Chunks, die zwar vom Root Set entfernt wurden, aber dennoch gültig bleiben sollen, gelegt werden. Der TwoSpace Algorithmus betrachtet den Register Stack als Teil des Root Set. Jede eingebaute Funktion (siehe sina_builtins.c) muss dafür sorgen, dass noch verwendete Chunks immer mindestens über den Register Stack erreichbar sind. Nach jedem Aufruf, bei dem eine Allokation stattfinden könnte, müssen zwischengespeicherte Zeiger wieder aus dem Register gelesen, da sie vom TwoSpace Algorithmus eventuell verändert wurden. Der Register Stack wird mit den Funktionen allocate_push_register() und allocate_pop_register() bedient.

Dieser Effekt tritt beim Concurrent Garbage Collector auch auf: In der Zeit, in welcher ein Chunk vom Data Stack entfernt und verändert wird, bevor er wieder auf den Data Stack gelegt wird, könnte der Collector seine Suche nach grauen Chunks beenden und die collector list aufbauen. Eventuell hat der (zwar gültige) Chunk die falsche Farbe und wird in die collector list aufgenommen. Da in diesem Fall nicht vorhergesagt werden kann, ab wann ein Chunk ungültig wird, müssen Chunks in einem Schritt sowohl in das Register aufgenommen und vom Data Stack entfernt werden. Dies wird mit der Funktion sinavm_pop_front_to_register() erreicht.

Mit der Implementation des Concurrent Garbage Collectors in den Dateien concurrent_allocator.c und concurrent_allocator.h wurde das Hauptziel dieser Arbeit erreicht.

8 Vergleich TwoSpace vs. Concurrent

Das Sekundärziel dieser Arbeit, die beiden Garbage Collectors zu vergleichen, wurde mittels einer Testreihe erreicht, welche die Laufzeit des Testproblems unter verschiedenen Bedingungen misst. Dabei wurde der Garbage Collector, die Heap-Grösse, die Länge der Inputdatei in Zeilen, die Länge einer Zeile der Inputdatei in Zeichen und die Anzahl der für die Verarbeitung zur Verfügung stehenden Prozessoren varriiert.

Die beiden getesteten Garbage Collectors waren der TwoSpace Algorithmus und der Concurrent Garbage Collector.

Die verwendeten Heap-Grössen waren 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB etc. (Der Einfluss der Heap-Grösse ist nach etwa 2^10KB kaum mehr zu messen).

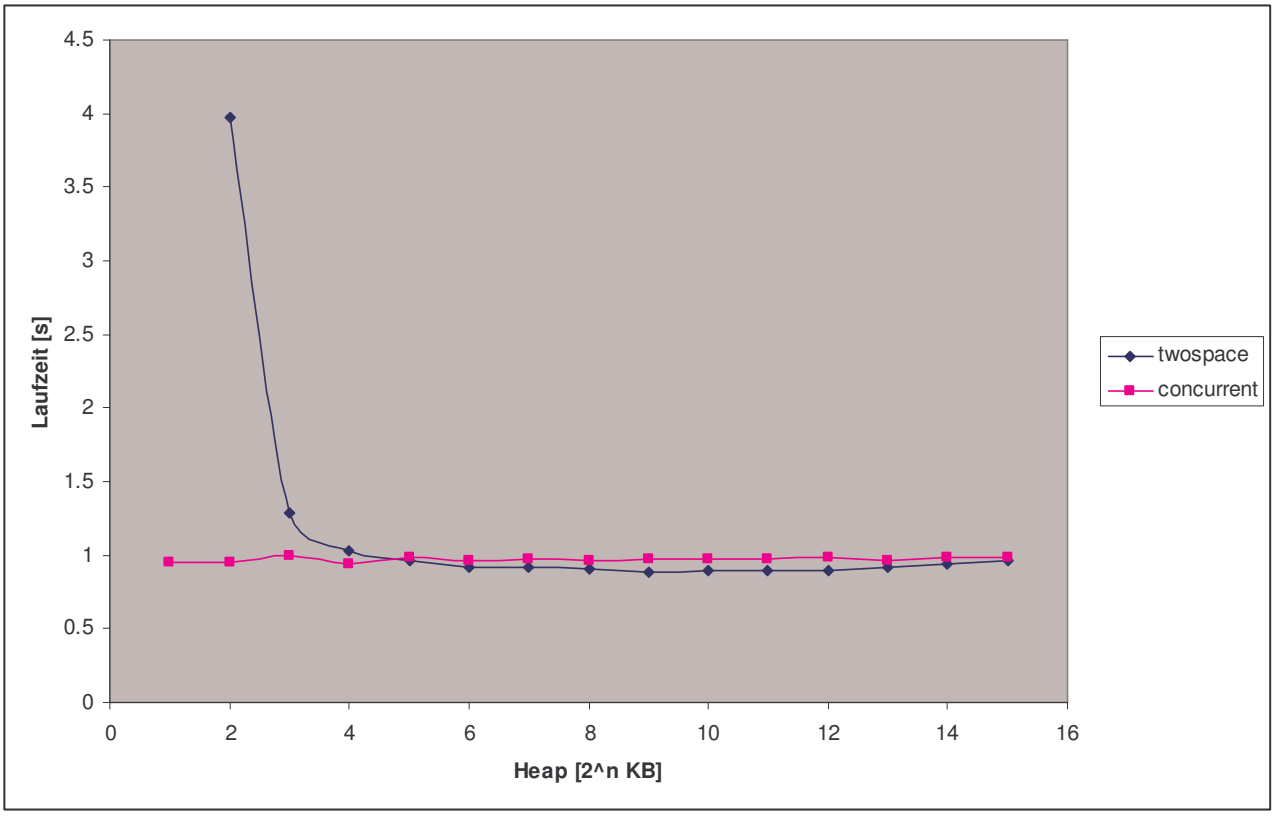
Die Länge der Inputdatei in Zeilen wurde wie folgt variiert: 1'000 Zeilen und 10'000 Zeilen.

Die Länge einer Zeile der Inputdatei wurde wie folgt variiert: 10 Zeichen, 100 Zeichen und 500 Zeichen pro Zeile. Bei 1000 Zeichen pro Zeile wurde die Laufzeit zum derart gross, dass es keinen Sinn machte, dies zu messen.

Die Testreihe wurde einerseits auf einer Maschine mit Dualcore Prozessor und andererseits auf einer Maschine mit einem einzelnen Prozessoren ausgeführt.

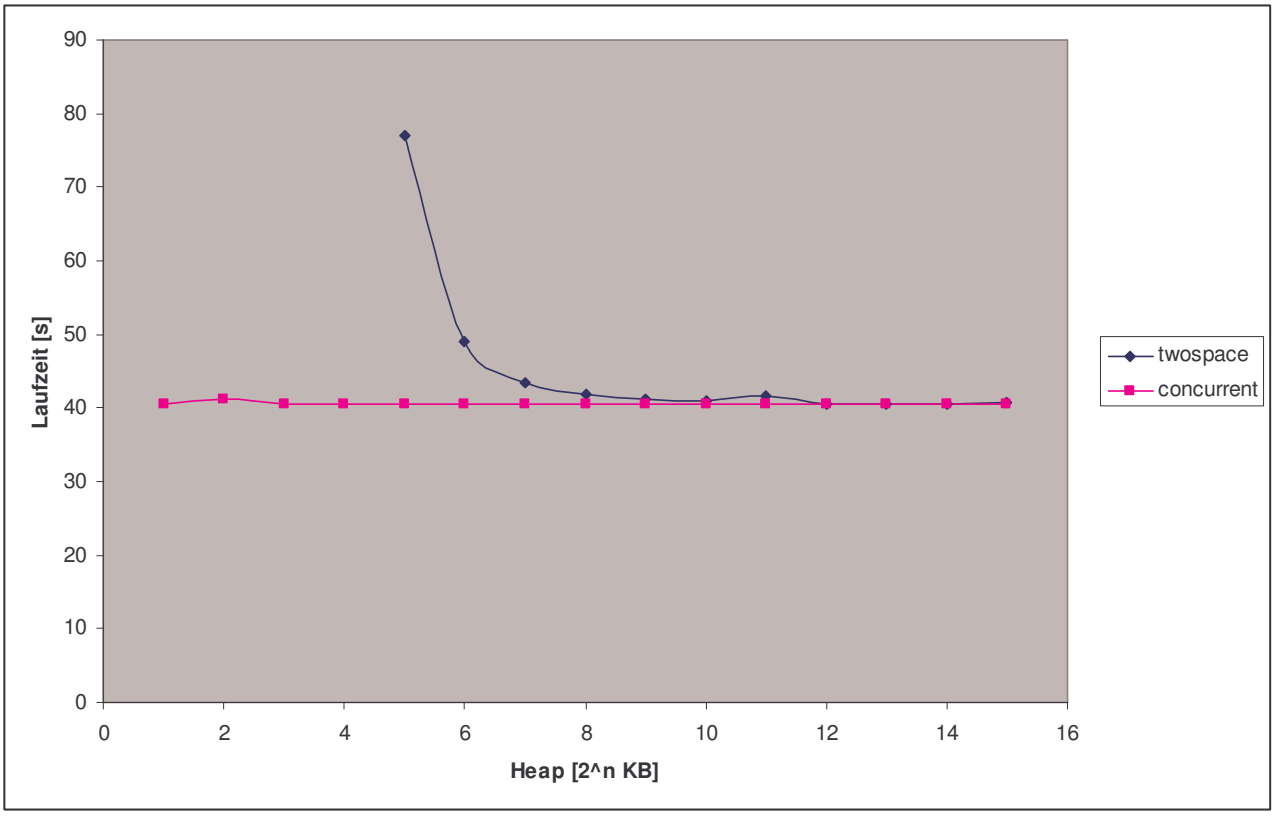
Die Testreihe wurde im Python Script testharness.py festgeschrieben, damit sie wiederholt ausgeführt werden kann. Die so gewonnen Daten wurden dann in eine Spreadsheet Applikation eingelesen und dann daraus die fogenden Graphiken erzeugt (wobei die Achse mit der Heapgrösse logarithmisch dargestellt wird).

8.1 10'000 Zeilen mit je 10 Zeichen mit zwei Prozessoren



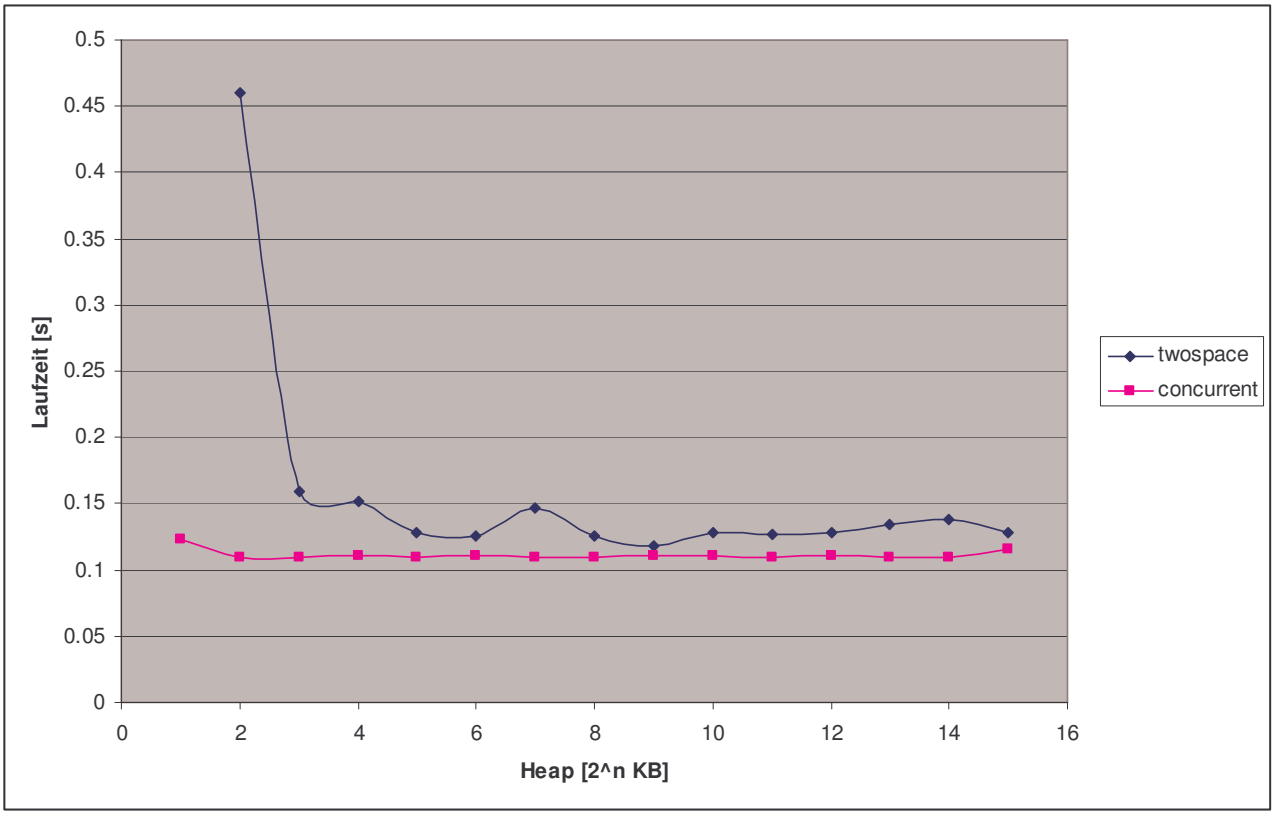
Bei kurzen Zeilen (kleiner statischer Speicherverbrauch) ist der TwoSpace Garbage Collector bei grösser werdendem Heap leicht besser als der Concurrent Garbage Collector. Allerdings ist für kleine Heaps der Concurrent Garbage Collector viel stärker (und kann schon mit dem kleinsten Heap den Testalgorithmus ausführen, während der TwoSpace GC damit erst ab einer gewissen Heap Grösse funktioniert). Interessant finde ich hier auch, wie konstant die Laufzeit beim Concurrent Garbage Collector in Bezug auf Heapgrösse bleibt.

8.2 10'000 Zeilen mit je 500 Zeichen auf zwei Prozessoren



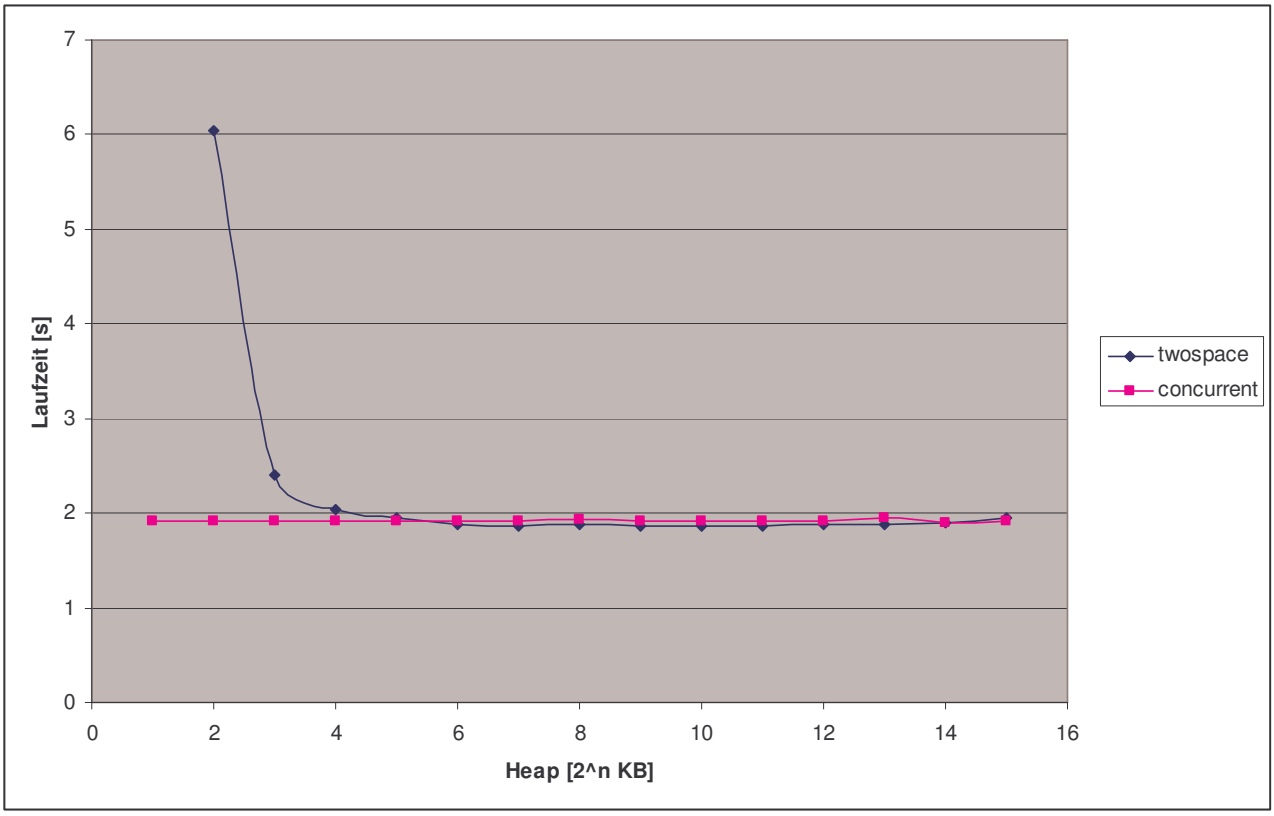
Bei 500 Zeichen pro Zeile (grossem statischem Speicher) ist der TwoSpace GC mit kleinem Heap klar überfordert. Allerdings gleicht sich die Laufzeit mit grösser werdendem Heap an die des Concurrent GC an. Im Gegensatz zum Beispiel mit kleinem statischen Speicher, wird der TwoSpace GC hier nicht schneller als der Concurrent GC.

8.3 1'000 Zeilen mit je 10 Zeichen auf zwei Prozessoren



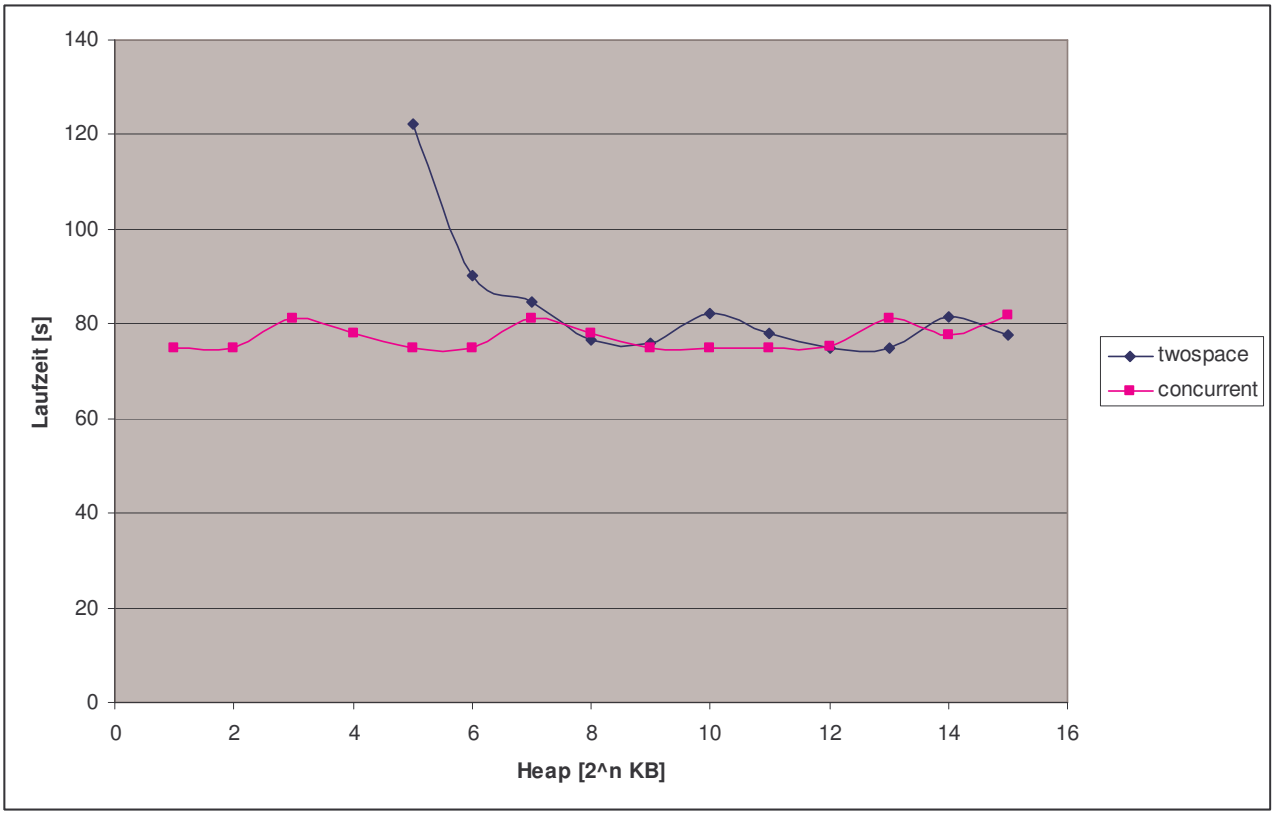
Hier zeigt sich ein ähnliches Verhalten wie beim Beispiel mit 10'000 Zeilen zu 10 Zeichen auf zwei Prozessoren. Ich gehe davon aus, dass die Ausreisser für den TwoSpace GC für grössere Heaps Messfehler (Prozessor mit Systemprozessen kurzfristig ausgelastet) darstellen.

8.4 10'000 Zeilen mit je 10 Zeichen auf einem Prozessor



Interessanterweise unterscheidet sich die Grafik kaum von der entsprechenden Variante für zwei Prozessoren. Die Laufzeit ist zwar insgesamt länger, dies liegt aber sicher nicht zuletzt an der älteren Hardware. Ich hätte erwartet, dass der TwoSpace Algorithmus bei Maschinen mit nur einem Prozessor klare Vorteile bietet und dass der Concurrent GC mit nur einem Prozessor einen grossen Overhead aufweist. Es könnte sein, dass die Wahl des Testproblems, welches mit vielem blockierenden I/O funktioniert, die Nachteile eines einzelnen Prozessors beim Multitasking aufwiegt, da während die I/O im Mutator blockiert, der Collector seine Arbeit verrichten kann.

8.5 10'000 Zeilen mit je 500 Zeichen auf einem Prozessor



Ich bin mir nicht sicher, wie die Schwankungen in diesem Graphen zu interpretieren sind. Möglicherweise sind dies Messfehler infolge von Systemlast. Die grundsätzlichen Formen der Graphen blieben jedoch erhalten. Die gleiche Zeilenlänge mit 1'000 Zeilen produziert keine solchen Schwankungen, sondern den fast gleichen Graphen wie auf zwei Prozessoren. Dies erhärtet die Vermutung der Systemlast als Quelle von Messfehler.

8.6 Interpretation und Schlussfolgerung

Grundsätzlich lässt sich sagen, dass für grosse Heaps der Unterschied in der Laufzeit zwischen den beiden Algorithmen verschwindet. Erfreulich für den Concurrent Garbage Collector ist, dass die Laufzeiten recht unabhängig von der Grösse des Heaps bleiben. Beim TwoSpace GC gilt dies jedoch für genügend grosse Heaps auch.

Die Implementation von TwoSpace ist im Vergleich zum Concurrent Garbage Collector trivial, was vielleicht dessen Beliebtheit bei industriellen Runtimes erklärt: Je einfacher ein Algorithmus ist, desto einfacher (und billiger) kann er gewartet werden und desto weniger Fehler enthält er.

Um der Forderung des Concurrent Garbage Collectors, dass alle Chunks gleich gross sein müssen, nachzukommen, musste im Falle der Sina Virtual Machine für die einfachen Datentypen INTEGER, SYMBOL, ESCAPED_SYMBOL und NATIVE zusätzlich Speicher “verschenkt” werden. So war bei diesen Datentypen 1/3 des Speichers (Chunkgrösse abzüglich Metadaten) sogenannter “Padding”, also unbenutzt. Bei industriellen Programmiersprachen sind viele weitere Datentypen, sowie (z.B. bei Feldern) variierbare Speichergrössen erforderlich. Diese können nicht mit dem Concurrent Garbage Collector zusammen verwendet werden, da der GC den Heap unabhängig von der effektiven Verwendung der einzelnen Speicherpositionen durchwandern können muss.

9 Literaturverzeichnis

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M: *On-The-Fly Garbage Collection: An Exercise in Cooperation.* Communications of the ACM, 21(11):965-975.

Brian W. Kernighan / Dennis M. Ritchie: *The C Programming Language. Second Edition.* Prentice Hall, 1988.

Alfred V. Aho / Ravi Sethi / Jeffrey D. Ullman: *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

„POSIX Threads Programming“. URL: http://www.llnl.gov/computing/tutorials/pthreads/ [Stand 10. August 2007].

10 Funktionsreferenz der Programmiersprache Sina

Im Folgenden werden die eingebauten Funktionen, wie sie in der Datei sina_builtins.c definiert sind, beschrieben. Alle Funktionen brechen die Verarbeitung mit einer Fehlermeldung ab, falls der falsche Datentyp auf dem betrachteten Stack vorgefunden wird oder der Stack nicht genügend Werte aufweist. Data Stack und Code Stack sind eigentlich Listen. In den folgenden Beschreibungen ist der Anfang der Liste jeweils “oben” und das Ende der Liste jeweils “unten”. Dies liegt daran, dass Stack Operationen auf einzeln verketteten Listen nur am Listenkopf gemacht werden können. Um die Funktion “append” (nötig für den Cesar Shift) zu implementieren, enthält jede Liste auch einen Verweis auf ihr Ende (eine PUSH-Operation am Listenende wäre also möglich), allerdings können vom Ende der Liste keine Elemente entfernt werden (die POP-Operation ist somit nicht implementierbar).

Funktion	*Beschreibung*
add	Addiert zwei Zahlen. Diese werden nacheinander vom Data Stack entfernt und ein neue Zahl mit der Summe wird auf den Data Stack gelegt.
append	Entfernt das oberste Element vom Data Stack und fügt es hinten an eine Liste, welche nun zuoberst auf dem Data Stack liegt, an.
bind-symbol	Entfernt das oberste Symbol vom Data Stack. Entfernt das nächste Element vom Data Stack und verknüpft es mit dem Symbol, sodass fortan, wenn dieses Symbol interpretiert werden soll, das verknüpfte Element interpretiert wird.
break	Entfernt das oberste Element vom Code Stack (der aktuelle Block).
call	Entfernt den obersten Block auf dem Data Stack und führt ihn aus (legt ihn auf den Code Stack).
char-is-alpha	Entfernt die oberste Zahl auf dem Data Stack. Falls diese Zahl gemäss ASCII als Buchstabe interpretierbar ist, wird eine neue Zahl mit dem Wert 1 auf den Data Stack gelegt. Ansonsten wird eine neue Zahl mit dem Wert 0 auf den Data Stack gelegt.
char-to-upper	Entfernt die oberste Zahl auf dem Data Stack. Dessen Wert wird als Zeichen interpretiert. Falls diese Zahl ein Kleinbuchstabe repräsentiert (zwischen dem ASCII Wert für 'a' und 'z' liegt), wird der entsprechende Grossbuchstabe auf den Data Stack gelegt. Ansonsten wird eine neue Zahl mit dem gleichen Wert auf den Data Stack gelegt.
drop	Entfernt das oberste Element auf dem Data Stack.
dup	Legt eine Kopie des obersten Elementes auf dem Data Stack auf den Data Stack. Die beiden Elemente sind identisch, es wird also eine Referenzkopie (im Gegensatz zu

	einer Wertekopie) gemacht.
equals	Entfernt die obersten zwei Zahlen vom Data Stack und legt eine neue Zahl auf den Data Stack mit dem Wert 1, falls die beiden entfernten Zahlen den gleichen Wert aufweisen, ansonsten 0.
if	Entfernt zuerst eine Zahl und anschliessend entweder ein Symbol oder ein Block vom Data Stack und interpretiert das Symbol oder den Block, falls die Zahl ungleich 0 war.
list-head	Entfernt das erste Element aus der Liste zuoberst auf dem Data Stack und legt es auf den Data Stack. Auf diese Weise kann über eine Liste iteriert werden, die Liste wird allerdings dabei verändert.
list-is-empty	Entfernt die Liste zuoberst auf dem Data Stack und legt eine neue Zahl auf den Data Stack, dessen Wert 1 ist, falls die Liste leer war, ansonsten 0.
list-new	Legt eine neue, leere Liste auf den Data Stack.
list-prepend	Entfernt das oberste Element vom Data Stack und fügt es vorne in die Liste, welche nun zuoberst auf dem Data Stack liegt, ein.
loop	Der aktuelle Block (zuoberst auf dem Code Stack) wird wieder von Vorne interpretiert (die Referenz auf das nächste auszuführende Element wird wieder auf das erste Element des Blocks gesetzt).
mod	Der ganzzahlige Rest zweier nacheinander vom Data Stack entfernten Zahlen wird als neue Zahl auf den Data Stack gelegt. Dabei ist die erste vom Data Stack entfernte Zahl der Dividend und die zweite entfernte Zahl der Divisor.
not	Entfernt eine Zahl vom Data Stack und legt eine neue Zahl auf den Data Stack mit dem Wert 1, falls die entfernte Zahl den Wert 0 hatte, ansonsten 0.
print-int	Entfernt die Zahl zuoberst auf dem Data Stack und schreibt das entsprechende Zeichen auf STDOUT.
print-string	Entfernt die Liste zuoberst auf dem Data Stack und schreibt für jede darin enthaltene Zahl ein Zeichen auf STDOUT. Falls die Liste Elemente enthält, die keine Zahlen sind, wird die Verarbeitung mit einer Fehlermeldung abgebrochen.
read-line	Legt eine neue Liste auf den Data Stack, an die nacheinander Zahlen angehängt werden, die jeweils den Wert des nächsten Zeichens in STDIN haben, bis das Zeilenendzeichen (welches auch an die Liste angehängt wird) oder EOF gelesen wird.
roll	Manipuliert die obersten 3 Elemente des Data Stack wie folgt: Das oberste Element wird unter das dritte Element gelegt. Anschliessend befindet sich das ursprünglich zweite Element an erster Stelle, das ursprünglich dritte Element an zweiter Stelle und das ursprünglich erste Element an dritter Stelle.
sub	Subtrahiert zwei Zahlen. Diese werden nacheinander vom Data Stack entfernt und eine neue Zahl mit der Differenz dieser beiden Zahlen wird auf den Data Stack gelegt.
swap	Vertauscht die Positionen der zwei obersten Elemente auf dem Data Stack.
trace-off	Schaltet das in trace-on eingestellte Verhalten wieder aus.
trace-on	Setzt einen Flag in der Sina Virtual Machine, der dazu führt, dass der Interpreter während der Ausführung für jeden Schritt den Zustand der Sina Virtual Machine ausgibt.