Parallelisierung eines Software Modellprüfers für nebenläufige C++ Programme

Lehrstuhl für Programmiersysteme, Universität Dortmund, Deutschland

Damian Sulewski
Juni 2007

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Kapitel 1

Einführung

1.1 Vorwort

Immer mehr Bereiche in unserem Leben werden durch Technik bestimmt. Musste man vor einiger Zeit den Wecker noch jeden Abend neu aufziehen, wechselt man heute nur ab und zu die Batterien. Die Zeit kommt, Atom-Uhr genau, per Funk. Baute man im Mittelalter Burgen ca. 30 km voneinander entfernt (dies entspricht der Distanz, die ein Mensch am Tag zu Fuß zurücklegen konnte) können wir heute nahezu jeden Ort der Welt innerhalb von 24 Stunden erreichen.

Nicht nur das Vordringen moderner Technik in immer mehr Lebensbereiche, auch die Weiterentwicklung dieser Technik, ob zur Steigerung der Lebensqualität oder zur Erhöhung der Sicherheit, stellt eine Herausforderung an Prüfmechanismen dar. Klingelt der Wecker morgens aufgrund eines Programmfehlers nicht, so ist dies nicht tragisch. Entscheidet aber eine Software aufgrund eines Modellierungsfehlers bei einer Landung eines Flugzeugs nicht zu bremsen, kann dieses sogar Menschenleben kosten.

Eine bis heute gängige Prüfmethode ist das *Testen* (Gelperin und Hetzel, 1988; Hetzel, 1990). Hierbei werden nach einem vorher bestimmten Ablaufplan nahezu alle Eigenschaften des Modells geprüft, die Prüfungen ausgewertet und gegebenenfalls Fehler eliminiert. Diese Methode ist nicht nur sehr zeitaufwändig, sondern auch nicht unbedingt zuverlässig. Da die Tests von Menschen ausgewählt werden, können unwahrscheinliche, jedoch fatale Fehler übersehen werden. Hinzu kommt der riesige Zeit- und Arbeitsaufwand, der nötig ist, um umfangreiche Software-Projekte zu überprüfen. Natürlich muss der Aufwand in nahezu gleichem Umfang wiederholt werden, falls ein Fehler entdeckt und entfernt wurde.

Bei Software, an die besonders hohe Sicherheitsanforderungen gestellt werden (zum Beispiel Software die in lebenserhaltenden Geräten eingesetzt wird) ist eine andere Vorgehensweise notwendig. Es werden Fehlerbedingungen in einer so genannten *Metasprache* definiert und die Software in diese Sprache übersetzt (Holzmann, 2004; Robby u. a., 2003). Das so entstandene Modell kann nun von einem *Modellprüfer* (Clarke u. a., 1999) gelesen und bezüglich der Fehlerbedingungen verarbeitet werden. Verletzt die Software eine der Bedingungen so wird sie als Fehlerhaft eingestuft. Da die Übersetzung meist von Hand vorgenommen wird, entstehen zwei potentielle Quellen für falsche Testergebnisse. Erstens kann ein Fehler der während einer Prüfung diagnostiziert wurde, beim Übersetzten hinzugefügt worden sein. Zweitens ist es möglich, dass ein Fehler nicht erkannt wird, da dieser bei der Übersetzung unwissentlich korrigiert wurde, in der Software allerdings weiterhin existiert. Des Weiteren kann eine Modellprüfung nur Fehler diagnostizieren die in der Metasprache formuliert werden können.

Software Modellprüfer (Godefroid, 1997; Visser u. a., 2000a) sind hingegen in der Lage, Quelltexte zu prüfen. Sie haben Ihre Wurzeln in der abstrakten Interpretaion (Cousot und Cousot, 1977) und der Datenflussanalyse (Steffen, 1993) Die vom Entwickler erstellte Software wird gelesen in einer kontrollierten Umgebung ausgeführt und auf Fehler untersucht. Fehlerbedingungen können sowohl in den Quelltext, als auch in den Prüfer eingebunden werden. *StEAM*[*] (Mehler, 2006), ein am Lehrstuhl für Programmiersysteme der Universität Dortmund entwickelter Software Modellprüfer, der innerhalb dieser Diplomarbeit erweitert wurde, verarbeitet die Programmiersprache C++ (Stroustrup, 2000). Bei dieser Sprache handelt es sich um eine höhere Programmiersprache, die sowohl eine maschinennahe, als auch eine Programmierung auf hohem Abstraktionsniveau ermöglicht. Das Programm wird in einer virtuellen Maschine (Visser u. a., 2000b) ausgeführt, die von StEAM kontrolliert wird.

Automatische Verifikationsmethoden kämpfen mit einem Problem: Der Anzahl der möglichen *Programmzustände*. Ein Programmzustand ist ein Abbild des Programms zu einem bestimmten Zeitpunkt. Beim Testen werden zumeist so viele unterschiedliche Programmzustände erzeugt, dass die Ressourcen (Zeit oder Systemspeicher) nicht ausreichen, um komplexe Programme zu überprüfen. Bei der Software Modellprüfung gibt es mehrere Ansätze, um diesem Phänomen zu begegnen , z.B. (Jard und Jéron, 1992; Lluch-Lafuente u. a., 2002; Holzmann, 1998). Ein möglicher Ansatz ist die in dieser Arbeit beschriebene Externalisierung und Parallelisierung. Dabei wird der Zustandsraum auf mehrere Rechner verteilt, um so Zugang zu größeren Ressourcen zu erhalten. Da die einzelnen Knoten unterschiedliche, unabhängige Programmzustände untersuchen, erwarten wir so eine schnellere Antwort. Kombiniert mit der Vereinigung von Hauptspeicher können auch komplizierte Software Modelle untersucht werden. Diese Methode klingt verlockend, da sie auf den ersten Blick leicht skaliert. Falls die gegebene Anzahl an Rechenknoten nicht ausreicht, fügt man weitere hinzu. Dabei muss man aber eines beachten. Wendet der Algorithmus zu viel Zeit für die Kommunikation zwischen Rechenknoten auf, so geht der Zeitvorteil, der durch die Benutzung mehrerer Rechner erlangt wurde, verloren.

Ziel dieser Diplomarbeit war demnach den Software Modellprüfer StEAM zu parallelisieren und mehrere Rechner, die über ein Netzwerk verbunden sind, zur Überprüfung des Modells zu verwenden. Dabei sollte der zu entwickelnde Algorithmus, über geeignete Verteilungsfunktionen, nicht zu viele Ressourcen für die Kommunikation verwenden, aber auch sicherstellen, dass Zustände nicht mehrfach expandiert werden. Der Modellprüfer muss natürlich weiterhin in der Lage sein, einen Fehlerzustand zu finden und den Fehlerpfad korrekt auszugeben. Des Weiteren sollte analysiert werden, ob eine striktere Trennung, zwischen StEAM und der virtuellen Maschine, weitere Möglichkeiten zur Modellprüfung eröffnet.

1.2 Annahmen

In dieser Arbeit werden folgende plausiblen Annahmen getroffen.

1. Die Zustände besitzen eine globale Repräsentation, da ein Rechner sonst die Zustände anderer Rechner nicht interpretieren kann.
2. Die Zustände fordern Speicher dynamisch an und geben diesen wieder frei.
3. Die maximale Größe eines Programmzustands ist beschränkt durch die Größe des Hauptspeichers des Rechenknoten mit minimalem Hauptspeicher.
4. Die zu prüfenden Programme können automatisch in die Sprache der virtuellen Maschine übersetzt werden.

[*] State Exploring Assembly level Model Checker (http://steam.cs.uni-dortmund.de)

5. Die Verteilungsfunktion ist global und wird während der Prüfung nicht verändert.

6. Ein Programmzustand enthält alle zur Berechnung der Verteilungsfunktion benötigten, Informationen.

1.3 Struktur

Die Arbeit ist wie folgt gegliedert:

Kapitel 2 stellt den verwendeten Software Modellprüfer StEAM vor. Es wird die Funktionsweise erläutert, und die unterstützten Suchalgorithmen vorgestellt. Weiterhin wird die Speicherstruktur näher beschrieben und StEAM mit weiteren Software Modellprüfern verglichen.

Kapitel 3 beschreibt eine Erweiterung von StEAM nicht direkt benötigte Daten auf ein sekundäres Medium (zum Beispiel eine Festplatte) auszulagern. Die Externalisierung ist Grundlage für die Parallelisierung.

Kapitel 4 beschreibt zwei Erweiterungen, die für die Parallelisierung notwendig sind: Ein Interface, über welches auf die virtuelle Maschine zugegriffen wird und virtuelle Speicheradressen.

Kapitel 5 erläutert detailiert den Algorithmus zur Parallelisierung. Beschrieben wird die Kommunikation, die Funktionen zur Verteilung der Zustände und die Implementierung.

Kapitel 6 zieht ein Fazit und zeigt Möglichkeiten auf, StEAM zu erweitern.

Im **Anhang** werden die Modelle, im Quelltext, vorgestellt, die bei der Durchführung der Experimente verwendet wurden.

1.4 Kontributionen

Diese Arbeit erweitert den Software Modellprüfer StEAM um folgende Eigenschaften.

Interface zur Steuerung einer virtuellen Maschine
Der Zugriff auf die virtuelle Maschine wurde über ein Interface realisiert. Dieses separiert die virtuelle Maschine von dem Modellprüfer und ermöglicht einen Austausch der selben. Mit dem Wechsel der virtuellen Maschine wäre StEAM in der Lage, Programme in beliebigen Programmiersprachen zu prüfen.

Virtuelle Speicheradressen
Das ausgeführte Programm greift nicht auf reale, sondern auf virtuelle Speicheradressen zu. Diese stellen eine Verbindung zwischen der Speicherverwaltung des Betriebssystems und dem Modell her. Durch Anpassung der virtuellen Adressen wird eine Übertragung auf andere Rechner und eine Verifikation auf weitere Fehlerarten möglich.

Parallelisierung
Es werden mehrere Rechner, die durch ein Netzwerk verbunden sind, eingesetzt, um Modelle zu prüfen. Die gemeinsame Nutzung der Ressourcen, aller Rechner, ermöglicht es komplexe Modelle in kürzerer Zeit zu untersuchen.

Modelle

Weitere Probleme wurden in C++ implementiert. Diese demonstrieren sowohl die Fehlerarten, als auch die Komplexität der Modelle, die StEAM verarbeiten und prüfen kann.

Kapitel 2

Der Software Modellprüfer StEAM

> Der schlimmste aller Fehler ist,
> sich keines solchen bewußt zu sein.
>
> *Thomas Carlyle*

2.1 Entstehung

Der Software Modellprüfer StEAM (Mehler, 2006) prüft nebenläufige Programme, die in der Programmiersprache C++ geschrieben wurden. Hierzu verwendet StEAM die "Internet C++ Virtual Machine"* (ICVM), die um spezielle Methoden zur Software Verifikation erweitert wurde.

Die ICVM wurde aufgrund ihrer Performance ausgewählt. Diese virtuelle Maschine wurde mit dem Ziel programmiert, aktuelle Spiele, ohne diese neu kompilieren zu müssen, auf verschiedenen Plattformen auszuführen. Um die Geschwindigkeit der Maschine zu demonstrieren, wurde das kommerzielle 3D Spiel "DOOM" portiert.

Ein Programm, welches geprüft werden soll, wird zuerst durch einen Compiler in Maschinensprache, den sogenannten *Objektcode*, übersetzt. Für die ICVM wird ein spezieller Compiler benötigt, der den Quelltext in Objektcode für einen Motorola 86K Chip übersetzt. Ein so kompilierter Objektcode kann von der ICVM ausgeführt werden. Da StEAM Programme auf der Ebene des Objektcode prüft, ist es nötig, diese in einer kontrollierten Umgebung auszuführen. Eine solche Umgebung stellt eine virtuelle Maschine zur Verfügung. Wird ein Programm in einer virtuellen Maschine ausgeführt, so ist es möglich, Fehler zu erkennen. Hierfür werden Fehlerzustände definiert, die nicht von der virtuellen Maschine erreicht werden dürfen.

StEAM ist in der Lage, ein kompiliertes C++-Programm in die virtuelle Maschine zu laden, und diese, auf Fehler zu untersuchen.

2.1.1 Fehlertypen

Folgende Fehler wedern von StEAM diagnostiziert:

Deadlocks beschreiben einen Zeitpunkt an dem alle Prozesse eines Programms blockiert sind. Ein Prozess ist blockiert wenn er auf eine Ressource wartet.

*http://ivm.sourceforge.net/

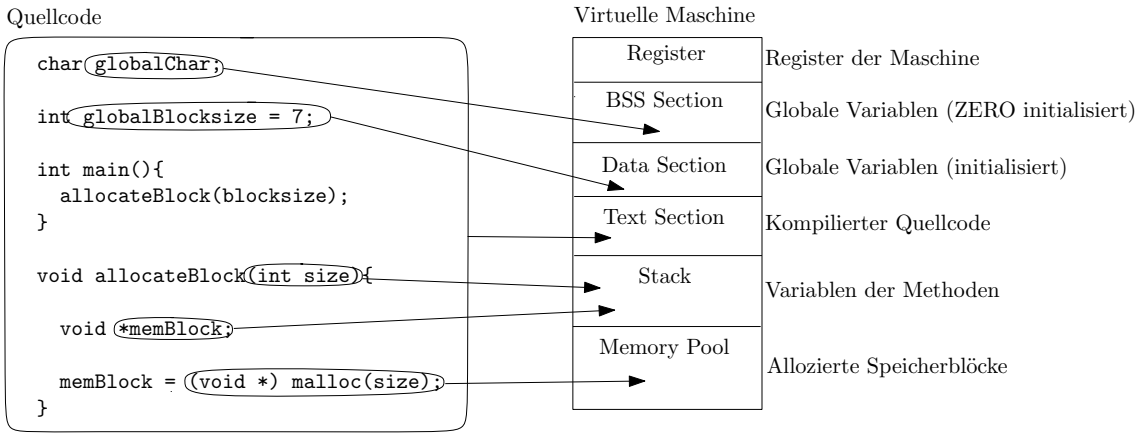

Abbildung 2.1: Abbildung der Variablen in einem Zustand

Assertion Violation werden erkannt indem die vom Programmierer gegebene Zusicherung überprüft wird. Den Variablen der boolschen Formel werden entsprechende Werte zugewiesen. Ist das Ergebnis der Formel negativ so handelt es sich um einen Fehlerzustand.

Segmentation Fault entstehen beim Zugriff eines Modells auf einen nicht angeforderten Speicher.

2.2 Funktionsweise

2.2.1 Zustände und Abschnitte

StEAM arbeitet mit Zuständen in einem *Zustandsraum* S. Bei einem *Zustand* (engl. *state*) $s \in S$ handelt es sich um ein Abbild der gesamten virtuellen Maschine zu einem bestimmten Zeitpunkt. Die virtuelle Maschine besteht aus den in Abbildung 2.1 gezeigten *Abschnitten* (*Reg, BSS, Data, Text, Stack, MemPool*) (engl. *sections*).

Wird ein Zustand s als Abbild der virtuellen Maschine erzeugt, so wird von jedem Abschnitt eine Kopie im Speicher abgelegt.

Enthält ein Modell mehrere Prozesse (engl. *threads*), so muss ein Zustand dies widerspiegeln. Die ICVM ist nicht in der Lage, mehrere Prozesse gleichzeitig auszuführen. In StEAM wird das Programm auf mehreren virtuellen Maschinen simmuliert. Da jedoch nur eine virtuelle Maschine zur Verfügung steht, wird diese mehrfach verwendet. Die Abschnitte *BSS, Data, Text, MemPool* sind bei allen Prozessen gleich, also werden nur die Abschnitte *Reg* und *Stack* einem Prozess zugeordnet.

Ein Zustand enthält demnach einen Vektor aus Prozess-ID's

$$P = \langle p_1, \cdots, p_n \rangle.$$

Jede p_{id}, mit $1 \leq id \leq n$ besteht wiederum aus einer Kopie eines Register- und eines Stack-Abschnittes der virtuellen Maschine, also:

$$p_{id} = \langle Reg, Stack \rangle.$$

Ein Zustand kann nun als Vektor

$$s = \langle P, BSS, Data, Text, MemPool \rangle$$

angesehen werden.

Da P wiedderum ein Vektor ist, der mehrere Abschnitte enthält, beschreibt s die virtuelle Maschine zu $|P|$ unterschiedlichen Zeitpunkten der Prüfung.

Ist s ein Zustand und $p_{id} \in P \in s$ ein Paar aus *Reg* und *Stack*, so ordnet (s, p_{id}) die virtuelle Maschine einem eindeutigen Zeitpunkt zu.

Wird die virtuelle Maschine initialisiert, enthält diese lediglich zwei Abschnitte: Die *Register* und einen *Stack*. Bei dem Stack handelt es sich um einen Kellerspeicher in dem sowohl Variablen als auch Informationen zur Ausführung des Programms abgelegt werden können. Register sind Speicherbereiche, die direkt in dem Prozessor liegen und deshalb sehr kurze Zugriffszeiten haben. Die Größe beider Sektionen ist einmal durch die Anzahl der Register im CPU Modell und einmal durch die Größe des Stacks festgelegt.

Lädt StEAM ein Modell in die virtuelle Maschine so werden die restlichen Abschnitte in der folgenden Reihenfolge erzeugt:

- Der *Text* Abschnitt enthält den Objektcode, dieser besteht aus *Opcodes*, den Maschinensprachebefehlen die das Programm beschreiben.
- Globale Variablen, die bereits vor der Ausführung des Programms mit einem Wert belegt werden, liegen, direkt nach dem Text, in der *Data* Section.
- Die *BSS* Section enthält alle globalen Variablen, denen zum Zeitpunkt der Initialisierung kein Wert zugewiesen wurde.
- Der *Memory Pool* ist der einzige Abschnitt, der eine variable Größe während der Ausführung aufweist. Er wird mit einer Größe von 0 initialisiert und enthält zur Laufzeit Speicherbereiche, die das ausgeführte Programm anfordert.

Zusätzlich zu den Informationen, die die virtuelle Maschine beschreiben, enthält ein Zustand weitere Informationen, die zur Prüfung benötigt werden. So beinhaltet jeder Zustand eine Referenz auf den Vaterzustand; den Dateinamen; die Zeile im Quellcode, die ausgeführt wurde, während der Zustand generiert wurde und einen *Hashwert*. Dabei ist ein Hashwert das Ergebnis der Anwendung einer Funktion $k : S \rightarrow \mathbb{Z}$, die einen Zustand auf eine ganze Zahl abbildet.

Ein neuer Zustand entsteht am Ende einer jeden Zeile im Quelltext; An jeder nichtdeterministischen Verzweigung, oder an einer speziellen, vom Programmierer im Quelltext gesetzten, Anweisung.

2.2.2 Ablauf einer Prüfung

StEAM startet zuerst eine Instanz der virtuellen Maschine und lässt diese das Modell laden. Hierbei wird laufend geprüft, ob die “main” Routine erreicht wurde. Ist dies der Fall, so pausiert die virtuelle Maschine und es wird ein Zustand erzeugt. Der so entstandene Zustand wird *Initialzustand* genannt, und mit $\mathcal{I}$ bezeichnet. In $\mathcal{I}$ gilt $|P| = 1$ da Prozesse erst in der ”main”Methode gestartet werden können.

Erreicht die virtuelle Maschine eine Stelle im Programm, an der eine nichtdeterministische Entscheidung gefällt werden soll, wird erneut pausiert. Erst jetzt greift StEAM signifikant in den Ablauf des Programms ein. Wieder wird ein Zustand s erzeugt und der Programmablauf fortgeführt, aber nur bis zur der Stelle an der ein Zustand s' erzeugt worden ist. Nun entschiedet der gewählte Suchalgorithmus welcher Weg verfolgt wird. Einerseits kann StEAM den Programmablauf, zum Zeitpunkt als s erzeugt wurde, auf einem weiteren Pfad fortführen, andrerseits kann s' weiter ausgeführt werden. Abbildung 2.2 verdeutlicht die Arbeitsweise des Modellprüfers.

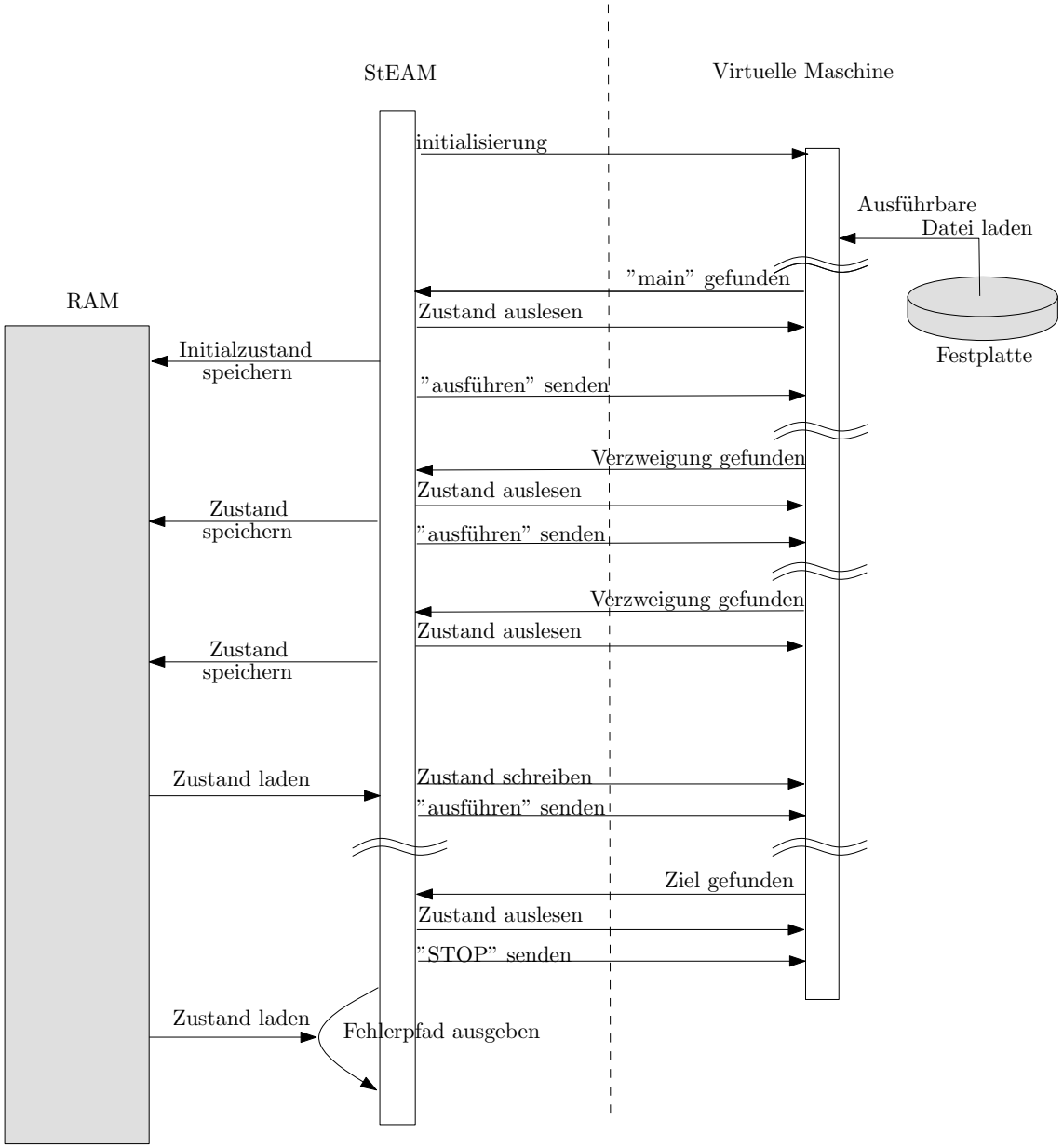

Abbildung 2.2: Ablauf einer Prüfung in StEAM

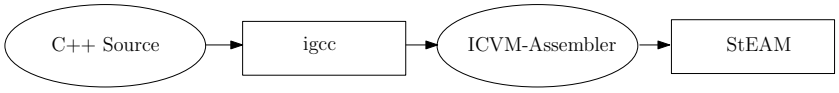

Abbildung 2.3: Prüfen von C++ Source mit Hilfe von StEAM

Bei der Ausführung wird jeder Zustand mit einer vorher gegebenen Fehlerdefinition verglichen. Wird ein *Fehlerzustand* (symbolisch durch ein *E* dargestellt) erreicht, gibt StEAM einen *Fehlerpfad* aus. Dieser enthält alle nötigen Informationen, die der Programmierer braucht, um den Fehler im Quelltext verfolgen zu können.

Ein Modellprüfer, der Software auf Assemblerebene prüft, liest direkt die Anweisungen, die auch der Prozessor eines Computers verarbeitet. Es ist also nicht nötig, das Modell für die spätere Ausführung unter realen Bedingungen erneut zu übersetzen. Da aber der erzeugte Maschinencode nur auf Maschinen desselben Typs lauffähig ist, ist auch das kompilierte Modell auf diese Maschinen begrenzt.

StEAM verwendet zur Prüfung die ICVM, die intern einen Motorola 68k Prozessor simuliert. Um die Geschwindigkeit der virtuellen Maschine zu erhöhen, wurden einige Opcodes eingeführt. Im Zuge dieser Optimierung entstand der *igcc*, eine Variante des *gcc*[†] (Stallman, 2003), der die speziellen Opcodes der ICVM unterstützt. Programme die mit igcc kompiliert wurden, können sowohl unter Verwendung der ICVM direkt ausgeführt, als auch von StEAM geprüft werden.

Abbildung 2.3 zeigt die einzelnen Schritte einer Prüfung von C++ Quelltext. Der igcc ertellt aus dem C++ Quelltext den *ICVM-Assembler Code*.

2.2.3 Ausgabe des Fehlerpfades

Bei der Prüfung *expandiert* StEAM den Zustandsraum des Programms. Dies bedeutet, StEAM verfolgt alle möglichen Pfade, die eine Ausführung der Assemblerbefehle erlaubt. Bei deterministischen Programmen ist dieser Zustandsraum eine einfach verkettete Liste. Besteht ein Programm aus parallel ausgeführten Funktionen, wie z.B. Threads, so entstehen, da die Reihenfolge bzw. Geschwindigkeit der einzelnen Threads nicht festgelegt ist, Verzweigungen. Eine weitere Möglichkeit für Verzweigungen sind nichtdeterministische Ausdrücke im Quelltext, die einer Variablen mehrere Werte zuweisen.

Gibt StEAM nach der Expandierung die Nachricht aus, das Programm sei fehlerfrei, so ist der gesamte Zustandsraum erforscht worden. Dies bedeutet: Alle möglichen Pfade der Ausführung wurden generiert und alle Zustände haben die Fehlerprüfung bestanden.

Findet StEAM einen Fehlerzustand, so wird ein *Fehlerpfad* erzeugt. Diesem Fehlerpfad kann man entnehmen, welcher Weg durch den Zustandsraum gewählt wurde um einen Fehlerzustand zu erreichen. Abbildung 2.4 zeigt den Fehlerpfad der ausgegeben wurde, nachdem StEAM im *Speisende Philosophen*[‡] Protokoll einen *Deadlock* gefunden hat.

Um einen solchen Fehlerpfad zu generieren, verweist jeder Zustand auf seinen Vorgänger. So ist es möglich, durch Verfolgen der Referenzen von E zu $\mathcal{I}$ zu gelangen. Wurde ein solcher Pfad durchlaufen und abgespeichert, kann nun, angefangen bei $\mathcal{I}$, ein Fehlerpfad aufgezeigt werden. Um den Fehler im Quelltext zu lokalisieren, wird die Zeilennummer und der Dateiname für jeden Zustand ausgegeben, der auf dem Fehlerpfad liegt.

[†]GNU Compiler Collection

[‡]Eine genaue Beschreibung, des Protokolls findet sich im Anhang A.1.

```
Step 1: Thread 1 - Line 44 State ID: 1, source-file: philosophers.c -  main
Step 2: Thread 4 - Line 26 State ID: 5, source-file: Philosopher_err.cc -  Philosopher::run
Step 3: Thread 4 - Line 27 State ID: 15, source-file: Philosopher_err.cc -  Philosopher::run
Step 4: Thread 3 - Line 26 State ID: 63, source-file: Philosopher_err.cc -  Philosopher::run
Step 5: Thread 3 - Line 27 State ID: 158, source-file: Philosopher_err.cc -  Philosopher::run
Step 6: Thread 3 - Line 28 State ID: 431, source-file: Philosopher_err.cc -  Philosopher::run
Step 7: Thread 1 - Line 45 State ID: 790, source-file: philosophers.c -  main
Step 8: Thread 5 - Line 26 State ID: 1607, source-file: Philosopher_err.cc -  Philosopher::run
Step 9: Thread 2 - Line 26 State ID: 2653, source-file: Philosopher_err.cc -  Philosopher::run
Step 10: Thread 5 - Line 27 State ID: 4583, source-file: Philosopher_err.cc -  Philosopher::run
Step 11: Thread 2 - Line 27 State ID: 6923, source-file: Philosopher_err.cc -  Philosopher::run
Step 12: Thread 5 - Line 28 State ID: 10604, source-file: Philosopher_err.cc -  Philosopher::run
Step 13: Thread 1 - Line 45 State ID: 14821, source-file: philosophers.c -  main
Step 14: Thread 4 - Line 28 State ID: 20711, source-file: Philosopher_err.cc -  Philosopher::run
Step 15: Thread 2 - Line 28 State ID: 26972, source-file: Philosopher_err.cc -  Philosopher::run
```

Abbildung 2.4: Fehlerprotokoll der speisenden Philosophen

2.3 Suchalgorithmen

StEAM verwaltet Zustände die noch nicht expandiert wurden in einer Prioritäts-Warteschlange (engl. priority queue) *Open*. Die Reihenfolge, in der die Zustände expandiert werden, wird über eine Prioritätsfunktion festgelegt.

Wurde ein Initialzustand $\mathcal{I}$ erstellt, so beginnt StEAM in der Funktion **check** (Algorithmus 2.1) den Zustandsraum zu prüfen. Hierzu wird der Initialzustand $\mathcal{I}$ in *Open* eingesetzt und eine **while** Schleife gestartet, die terminiert, falls *Open* leer ist, was mit einer kompletten Exploration gleichzusetzen ist. Nachdem ein Zustand s aus *Open* extrahiert wurde [Zeile 3], wird dieser expandiert. Während der Expandierung wird die virtuelle Maschine nacheinander in alle Zustände (s, p_{id}) versetzt und gestartet. Nachdem ein neuer Zustand erzeugt wurde, wird dieser auf Fehler untersucht [Zeile 5] und gegebenfalls der, in der Funktion **genPath**(s) erzeugte, Fehlerpfad zurückgegeben. Ist der erzeugte Zustand fehlerfrei, so wird geprüft, ob es sich um einen neuen Zustand handelt oder ob ein identischer Zustand bereits erzeugt worden ist. Handelt es sich um einen neuen Zustand, wird dieser [in Zeile 9] entsprechend seiner Priorität in *Open* eingesetzt und die **while** Schleife beginnt von vorne. Algorithmus 2.1 beschreibt die Funktionalität von **check** als Pseudocode.

Wurde die **while** Schleife, ohne die Ausgabe eines Fehlerzustandes, beendet, so wurde der gesamte Zustandsraum durchsucht. Alle erzeugten Zustände wurden als Fehlerfrei eingestuft und es kann eine entsprechende Meldung ausgegeben werden.

Die Reihenfolge, in der Zustände expandiert werden, wird durch die Funktion **getNext** bestimmt. Durch Austauschen der Funktion (und, falls nötig, der **insert** Funktion) ist es möglich, verschiedene Suchalgorithmen zu implementieren. Um den Zustandsraum ungerichtet zu durchsuchen, werden die Breitensuche (Cormen u. a., 1990), die Tiefensuche (Tarjan, 1972) und die iterative Tiefensuche (Korf, 1985) unterstützt. Soll gezielt nach einem Fehler gesucht werden, so bietet StEAM die Algorithmen A* (Hart u. a., 1968), Bestensuche (Pearl, 1985) sowie IDA* (Korf u. a., 2001) an.

2.3.1 Ungerichtete Suche

Bei der Breitensuche gleicht die Priorität mit der die Zustände einsortiert werden der Tiefe des Zustandes. So kann sicher gestellt werden das alle Zustände mit einer kleineren Tiefe expandiert wurden. Zusätzlich wird der minimale Tiefenwert abgespeichert, um bei einer **getNext** Anweisung den Zustand, mit der minimalen Tiefe, schneller zu finden.

Algorithmus 2.1 check (Zustandsraum auf Fehlerzustände untersuchen)

Eingabe: Initialzustand $\mathcal{I}$
Ausgabe: Ein Pfad von $\mathcal{I}$ zum Fehlerzustand E, falls ein Fehlerzustand existiert, sonst **true** .
Variablen: *Open*;

```
 1: insert(Open, I)
 2: while Open ≠ ∅ do
 3:    State s ← getNext(Open)
 4:    for all s' ∈ Expand(s) do
 5:       if Errorstate(s') then
 6:          return genPath(s') { Fehlerpfad zurück geben}
 7:       end if
 8:       if not checkForDuplicate(s') then
 9:          insert(Open, s')
10:       end if
11:    end for
12: end while
13: return true {Gesamter Zustandsraum expandiert}
```

Für die Tiefensuche wird ebenfalls die Tiefe verwendet, jedoch wird hier die Tiefe eines Zustandes beim Einsetzen in *Open* von einer Konstanten M subtrahiert. M muss natürlich größer oder gleich der maximal erreichbaren Tiefe sein. Da **getNext** Zustände aus *Open* extrahiert, die eine minimale Priorität haben, werden so Zustände mit größerer Tiefe bevorzugt.

Der Unterschied zwischen der Tiefen- und der iterativen Tiefensuche besteht in der schrittweisen Anpassung der maximalen Tiefe. Die Funktion **getNext** gibt ausschließlich Zustände zurück, deren Tiefenwert kleiner als die, für diese Iteration gesetzte, maximale Tiefe ist. Zustände, deren Tiefenwert größer als die aktuelle Schranke ist, werden nicht gespeichert. Sind keine Zustände im aktuellen Tiefenabschnitt vorhanden, so wird die maximale Tiefe heraufgesetzt, und $\mathcal{I}$ erneut zur *Open* hinzugefügt. Algorithmus 2.2 zeigt die **getNext** Funktion für diesen Fall. MAX_DEPTH muss beim Start mit M initialisiert werden.

Algorithmus 2.2 getNext (nächsten zu expandierenden Zustand ausgeben)

Eingabe: Prioritäts-Warteschlange *Open*;
Ausgabe: Einen Zustand s, dessen Tiefe kleiner als die momentane maximale Tiefe ist
Variablen: Maximale Tiefe MAX_DEPTH

```
1: State s ← getMin(Open)
2: if s.depth ≤ MAX_DEPTH then
3:    return s
4: else
5:    Open ← ∅
6:    insert(Open, I)
7:    MAX_DEPTH ← MAX_DEPTH – 1
8:    return I
9: end if
```

2.3.2 Heuristiken

StEAM unterstützt einige Heuristiken, die eine gerichtete Suche ermöglichen. Wird eine solche beim Start angegeben, so errechnet StEAM aus dem zuletzt generierten Zustand s die Entfernung zum Fehlerzustand E. Anhand dieses *heuristischen Wertes* $h(s)$ wird beim Expandieren immer der Zustand

gewählt, der die kleinste Entfernung zu E aufweist. Die Heuristiken werden hier nur kurz erläutert, für eine vollständige Beschreibung siehe (Groce und Visser, 2002; Edelkamp und Lluch-Lafuente, 2001; Lluch-Lafuente, 2003)

MostBocked Der Wert der Heuristik ist die Anzahl der nicht *blockierten* Threads in einem Zustand. Ein Thread ist blockiert wenn er auf eine Ressource wartet.

Lock and Block Eine Erweiterung der MostBlocked Heuristik. Zu der Anzahl der nicht blockierten Prozesse wird die Anzahl der freien Ressourcen addiert.

Beide Heuristiken sind ausschließlich für die Suche nach einem Deadlock geeignet.

2.3.3 Gerichtete Suche

Für die gerichtete Suche nach Fehlerzuständen werden zwei Algorithmen unterstützt.

Bei der Bestensuche soll immer der Zustand zuerst expandiert werden, der die kleinste Entfernung zum Ziel aufweist. Dies wird dadurch erreicht, dass vor dem Einsetzen eines Zustandes in *Open* sein heuristischer Wert errechnet wird, und dieser genutzt wird, um die Zustände in *Open* zu priorisieren.

Wurde der A* Algorithmus ausgewählt, um einen Zustandsraum zu prüfen, wird in der insert-Funktion die Tiefe zum heuristischen Wert addiert und dieser Wert zur Priorisierung benutzt.

2.4 Speicherstruktur

Die generierten Zustände werden im Hauptspeicher abgelegt. StEAM verwendet mehrere Techniken, um die Anzahl der Zustände, die gespeichert werden, zu maximieren.

2.4.1 Hashing

Um ein doppeltes Expandieren eines Zustandes zu verhindern, unterstützt StEAM *Hashing*. Eine Hash-Funktion $k : S \rightarrow \mathbb{Z}$ bildet einen Zustand auf eine ganze Zahl ab. Sie errechnet aus einem Zustand $s \in S$ einen Schlüssel $k(s) \in \mathbb{Z}$. Hierbei kann es zu *Kollisionen* kommen, d.h. zwei Zustände s_1 und s_2 mit $s_1 \neq s_2$ erhalten denselben Schlüssel $k(s)$. Eine gute Hashfunktion minimiert die Anzahl der Kollisionen. Da die Anzahl der Schlüssel begrenzt ist, kommt es zwangsläufig zu Kollisionen. Um diese handzuhaben, müssen in der *Hashtabelle* an der Position $k(s)$ mehrere Zustände gespeichert werden.

Ein Zustand $s \in S$ kann als eine Folge von Zahlen (Bytes) a angesehen werden. Sie $|s|$ die Länge von s so wird mit folgender Formel ein Zustand als eine Zahl aufgefast:

$$s = \sum_{x=0}^{|s|} a_x$$

Fasst man einen Zustand s als eine Ganzzahl auf, wird der Zustandsraum S in Äquivalenzklassen $[0], [1], \cdots, [m-1]$ unter Verwendung folgender Äquivalenzrelation partitioniert

$$z \sim w \textit{ genau dann, wenn } z \bmod m = w \bmod m$$

Um eine gleichmässige Verteilung zu gewährleisten, ist m von großer Bedeutung. Wählt man ein gerades m aus, so ist $k(s)$ gerade wenn x es ist, und nur dann. Wählt man hingegen $m = r^w$ aus, so ist dies ebenfalls nicht optimal, da man für $x = \sum_{i=0}^{l} a_i r^i$

$$x \bmod m = \left(\sum_{i=w}^{l} a_i r^i + \sum_{i=0}^{w-1} a_i r^i \right) \bmod m = \left(\sum_{i=0}^{w-1} a_i r^i \right) \bmod m$$

erhält. In diesem Fall werden nur die letzten w Zeichen zur Verteilung verwendet.

Es bietet sich also an für m eine Primzahl zu wählen, die eine Zahl $r^i \pm j$ für kleine j nicht teilt, denn $m|r^i \pm j$ ist äquivalent zu $r^i \bmod m = \pm j$ und so ergibt sich, für den Fall +

$$x \bmod m = j * \sum_{i=0}^{l} a_i \bmod m$$

Dies bedeutet, dass für Bereiche dessen Summe aller Zeichen gleich ist, auch derselbe *Hashwert* errechnet wird.

Wird nun ein Hashwert für einen Zustand errechnet so wird in der Hashtabelle geprüft, ob bereits Zustände mit dem selben Hashwert existieren und, falls dies der Fall ist, diese mit dem neu erzeugten verglichen. Sobald bei dem Vergleich Unterschiede festgestellt werden, kann abgebrochen werden und der neue Zustand in der Hashtabelle abgelegt werden.

Partielles Hashing

Um eine Prüfung zu beschleunigen unterstützt StEAM *partielles Hashing* (Lerda und Sisto, 1999). Hierbei wird nur ein Teil des Zustandes herangezogen, um den Hashwert zu errechnen. Partielles Hashing wird für alle Abschnitte eines Zustandes unterstützt, aber auch für alle Kombinationen dieser Abschnitte.

Sei $s \in S$ ein Systemzustand, sei weiterhin $|s|$ die Länge des Zustands und sei a_x der Wert des an Stelle x liegenden Bytes. Es werden nun ein Startpunkt i und ein Endpunkt j gewählt, wobei natürlich $0 \leq i < j \leq |s|$ gelten muss. Die Punkte i und j können auch mehrfach gewählt werden, und werden dann als l_i und l_j bezeichnet. Es ergibt sich ein Hashwert aus

$$k(s) = \sum_{y=1}^{l} \sum_{x=l_i}^{l_j} a_x * mod\, m.$$

Einen, aus der Konkatenation einem oder mehrerer einzelner Abschnitte eines Zustandes errechneten, Hashwert bezeichnen wir als *partiellen Hashwert*.

2.4.2 Kompression in StEAM

Aufgrund der Tatsache, dass bei einer Expandierung nur wenige Teile eines Zustandes verändert werden, wurde eine Kompression implementiert. Ändert sich ein Abschnitt nicht, so wird lediglich eine Referenz auf den Abschnitt des Vorgängers gespeichert. Natürlich ist es so immer noch möglich, zwei identische Sektionen im Speicher vorzuhalten, jedoch zeigt bereits diese einfachere Methode eine große Speicherersparnis. Diese Idee wird bei der Externalisierung aufgegriffen, und erweitert.

2.5 Vergleich mit anderen Software Modellprüfern

StEAM bietet eine Reihe von Funktionen, die andere Modellprüfer nicht in diesem Umfang unterstützen. Um dies zu demonstrieren, werden hier ausgewählte Modellprüfer etwas näher betrachtet. Ausgewählt wurden Software Modellprüfer die Quelltext akzeptieren.

2.5.1 Verisoft

Der von den Bell-Labs entwickelte Modellprüfer *Verisoft* (Godefroid, 1997, 2005) gehört ebenfalls wie StEAM zur Klasse der Software Modellprüfer. Verisoft akzeptiert C Code als Eingabe und prüft diesen direkt auf Maschinencodeebene. Der Code und der Modellprüfer werden zu einer ausführbaren Datei kompiliert, diese wird gestartet und so das Modell geprüft. Als Ausgabe erhält man ebenfalls einen Fehlerpfad zu einem Fehlerzustand. Die Datei, die diesen Fehlerpfad enthält, kann dann von einem Simulator gelesen werden, und so das Programm analysiert. Verisoft ist in der Lage Deadlocks, Livelocks und Assertion Violations zu finden.

Leider birgt die Methode der Kompilierung der Software in den Modelliprüfer einige Nachteile. Segementation Faults oder Memory Leaks können so nicht gefunden werden, da das Programm direkt auf dem Rechner ausgeführt wird, nicht in einer virtuellen Umgebung. Trifft demnach ein solcher Fehler auf, hält die Prüfung an, ohne ein Ergebnis zu liefern. StEAM hingegen führt das Programm in einer virtuellen Umgebung aus, hat also sowohl die Kontrolle über die Anforderung von Speicher, als auch über den Zugriff auf nicht angeforderten Speicher.

Ein weiterer Aspekt ist die geforderete Neukompilierung der Software. Sollte Verisoft, was z.Z. noch nicht der Fall ist, andere Suchalgorithmen unterstützen, so muss bei jeder Änderung der Suchattribute, z.B. die Wahl eines anderen Suchalgorithmus, oder einer anderen Heuristik die Software neu kompiliert werden.

StEAM hingegen ist in der Lage, ein einmal kompiliertes Programm mit all seinen zur Verfügung stehenden Algorithmen und Heuristiken zu untersuchen.

2.5.2 Bogor

Bogor (Robby u. a., 2003) ist eine an der Kansas State University entwickelte Software. Sie arbeitet nicht direkt auf C++, sondern auf einer eigens entwickelten Programmiersprache. Die *Bogor Input Repräsentation* (BIR) unterstützt die dynamische Erzeugung von Objekten und Vererbung ebenso wie weitere Merkmale die objektorientierten Sprachen zu eigen sind. Die Software muß demnach erst in BIR übersetzt werden, um sie zu testen. Der große Vorteil von Bogor ist seine leichte Erweiterbarkeit. Da der Quellcode verfügbar ist, kann jeder Module (wie zum Beispiel einen Suchalgorithmus) implementieren und in Bogor einfügen. Aus diesem Grund wurde auch der StEAM Quellcode im Rahmen einer Projektgruppe unter die GPL Lizenz gestellt, so dass jeder ihn erweitern kann. Ein weiterer Vorteil ist eine vollständige Integration in Eclipse, einer Arbeitsumgebung zur Programierung von Software. Es gibt auch für StEAM ein Eclipse-Plugin, welches die Arbeitsumgebung befähigt alle Fähigkeiten des Software Modellprüfers zu nutzen. Leider bleibt der große Nachteil der notwendigen Übersetzung aus der Implementationssprache in BIR. Diese muß natürlich jedesmal durchgeführt werden, falls eine Änderung an der Software vorgenommen wird.

Auch Bogor ist nicht in der Lage, Speicherfehler zu entdecken, da das übersetzte Modell Speicher in einer anderen Art und Weise anfordert als das C++ Programm.

2.5.3 Java Path Finder

Der einzige Modellprüfer der ebenfalls eine virtuelle Maschine benutzt, um Software zu testen, ist der von der NASA entwickelte *Java Path Finder* (Visser u. a., 2000a). Hier wurde eine virtuelle Maschine für Java, zu einem Modellprüfer erweitert. Leider ist dieser Prüfer lediglich für Java geeignet, während die maschinennahe Sprache C++ heute als Standardsprache für Programierung von Softwareprojekten gibt, bei denen es auf Effizienz ankommt.

2.6 Resümee

Mit dem Werkzeug StEAM steht dem C++ Programmierer eine mächtige Software zur Hand. Sie ermöglicht, parallel laufende Programme während der Entstehungsphase zu prüfen, ohne diese vorher in andere Sprachen übersetzten zu müssen. StEAM kann aber nicht nur zur Prüfung von parallelen Programmen eingesetzt werden. Die Möglichkeit unerlaubte Speicherzugriffe und Verletzungen von Zusicherungen zu prüfen macht ihn zu einem voll funktionsfähigen Debugger. Mit dieser Software wurde zum ersten Mal gezeigt, das auch komplexe Sprachen geprüft werden können, ohne durch die Einschränkungen der Hardware Qualitätsverluste hinnehmen zu müssen. Durch die Fähigkeit, den gesamten Zustandsraum zu durchsuchen, lassen sich Fehler in der Software aufspühren, die man beim Testen vielleicht gar nicht gefunden hätte.

Kapitel 3

Externalisierung

Das Ganze ist mehr
als die Summe seiner Teile.

Aristoteles

3.1 Motivation

Ein Hauptproblem der Modellprüfung stellen die begrenzten Ressourcen dar. Sei es die Zeit, die benötigt wird, um ein Problem zu prüfen, oder der Speicher, der erforderlich ist, um alle Zustände, zwecks Duplikatsprüfung zu speichern. Hinzu kommt das Problem der sogenannten "Zustandsexplosion": Exploriert man einen Zustandsraum, so wächst die Anzahl der Zustände üblicherweise exponentiell in der Anzahl der Zustandsvariablen, weshalb man schnell die maximal verfügbaren Ressourcen ausnutzt. Zusätzlich zur der Anzahl der Zustände muss deren Größe in Betracht gezogen werden. Der Speicherbereich, den ein Programm anfordern kann, ist nur durch die Größe des Hauptspeichers begrenzt. StEAM muss also in der Lage sein, große Zustände zu prüfen.

Der erste eingeschlagene Weg, um StEAM die Prüfung komplexerer Programme zu ermöglichen, ist eine Externalisierung (Sanders u. a., 2002). Hierbei werden Informationen, die momentan nicht benötigt werden, auf ein sekundäres Medium, z.B. eine Festplatte ausgelagert (Stern und Dill, 1998; Kristensen und Mailund, 2003). Der große Vorteil einer Externalisierung ist die Größe des sekundären Speichermediums. Während der Hauptspeicher selbst bei aktuellen Computern auf 64 Gbytes begrenzt ist, liegt der maximal nutzbare Speicherplatz auf Festplatten bei einigen Terrabytes. Leider haben sekundäre Speichermedien einen entscheidenden Nachteil: Die Zugriffszeit ist weitaus langsamer als beim Hauptspeicher.

Dies wird durch die Art verursacht in der Festplatten konstruiert sind. Intern bewegt sich der Schreib-/Lesekopf einer Festplatte über eine sich drehende Metallscheibe, ähnlich einem Plattenspieler. Möchte man bei dem Plattenspieler ein Lied spielen so muss man die Nadel auf eine Position zwischen zwei Liedern legen, es sei denn es ist das erste. Ein Abspielen der Lieder in einer Reihenfolge, die nicht der gepressten entspricht, würde also ein Umsetzten der Nadel nach jedem Lied voraussetzen. Ebenso geht eine Festplatte vor. Natürlich bewegt sich der Schreib-Lesekopf automatisch und schneller aber die Positionierung, hier *seek* genannt, beim Zugriff auf ein Datum kostet trotzdem Zeit. Hinzu kommt, dass man den Lesevorgang in einer Datei an jeder beliebigen Stelle starten kann. Speichern wir also alle Zustände in eine Datei, so wird es trotzdem zum seek kommen, wenn die Zustände

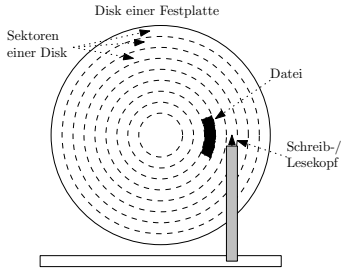

Abbildung 3.1: Interner Aufbau einer Festplatte. Der Schreib-/Lesekopf muss nach links bewegt werden, während sich die Disk dreht, um die Datei lesen zu können.

in einer anderen Reihenfolge als der gespeicherten gelesen werden. Abbildung 3.1 zeigt den internen Aufbau einer Festplatte mit nur einer sich drehenden *Disk*.

Um diesem Problem zu begegnen ist es notwendig, Algorithmen zu entwickeln, die sekundäre Speichermedien nutzen, wobei sich die Zugriffszeit nicht nachteilig auf die gesamte Prüfung auswirkt.

3.1.1 Vorüberlegungen

Zuerst gilt es zu entscheiden, wann ein vollständiger Zustand benötigt wird. Dies ist in zwei Situationen der Fall:

1. falls er expandiert werden soll
2. zur Duplikatsprüfung

Es wir demnach ein Algorithmus benötigt, der die Zustände möglichst nur dann auf eine Festplatte auslagert, wenn sie schon expandiert wurden, oder wenn es unwahrscheinlich ist, dass ein Duplikat entsteht. Natürlich sollte nur ausgelagert werden, wenn also kein Hauptspeicher mehr zu Verfügung steht.

Neben den Problemen, wann und wie ein Zustand externalisiert werden soll, stoßen wir bei der Externalisierung eines Modellprüfers auf folgende Herausforderungen. Ein Zustand, der sich im Speicher befindet, ist eine dynamische Struktur. Es handelt sich dabei um ein Objekt, welches aus mehreren Teilen besteht, die wieder mehrere Teile enthalten können. Der Zustand muß also vor dem Auslagern in eine Struktur übertragen werden, die sequentiell in eine Datei gespeichert werden kann. Dieser Weg muß auch umkehrbar sein, d.h. ein Zustand, der sequentiell aus einer Datei gelesen wird, muss in ein Objekt umgewandelt werden.

Es sollte noch erwähnt werden, dass die Struktur des Suchbaums unbedingt erhalten bleiben muss, da sie zur Konstruktion des Fehlerpfades benötigt wird.

3.1.2 Unterschiedliche Arten der Externalisierung

Grundsätzlich unterschieden wir bei der Externalisierung zwischen einer *vollständigen Externalisierung* und einer *teilweisen Externalisierung*. Während bei der vollständigen Externalisierung gefordert wird, dass der interne Speicherverbrauch konstant gehalten wird, darf ein Algorithmus bei der teilweisen Externalisierung mit zunehmender Rechenzeit mehr Speicher verbrauchen. Dies kann erforderlich sein, falls das zu externalisierdende Programm Speicherbereiche enthält, die nicht externalisiert werden können, z.B. die Position der Daten auf dem externen Medium. Da bei der Überprüfung des

Tabelle 3.1: Aufbau eines “Minizustands”

Name	Beschreibung
pred	einen Zeiger auf den Vater Minizustand
fid	die Nummer der Datei in der sich der Zustand befindet (falls er ausgelagert wurde)
fpt	einen Zeiger auf den Zustand in der Datei (falls er ausgelagert wurde)
h	den heuristischen Wert des jeweiligen Zustandes (zur Priorisierung in *Open*)
d	die Tiefe des Zustandes (ebenfalls zur Priorisierung in *Open*)
tid	eine *child_id*, also die Information welcher Prozess ausgeführt wurde
h_next	einen Verweis auf einen Zustand mit dem selben heuristischen Wert (benötigt von *Open*)
state	einen Verweis auf den Zustand, falls er sich im Speicher befindet

Zustandsraumes sehr viele dynamische Faktoren eine Rolle spielen und zusätzlich die Struktur des Suchbaums erhalten bleiben muss, wurde für StEAM eine relative Externalisierung gewählt.

3.2 Speicherstruktur

Um sowohl dem Problem der Zustandsgröße, als auch der “Zustandsexplosion” zu begegnen, wurde die Idee von *Minizuständen* entwickelt.

3.2.1 Minizustand

Die Minizustände (engl. ministates) verbleiben im Speicher und stellen ein Skelett des Zustandsraums dar. Da die Größe eines Minizustands konstant ist, kann vor einer Prüfung errechnet werden, wie viele Minizustände erzeugt werden, bis der Hauptspeicher aufgebraucht ist. Die konstante Größe der Minizustände ist auch der größte Vorteil dieser Idee. Während die Anzahl der Zustände, die im Speicher gehalten werden können, von deren Größe abhängt, ist es mit Hilfe der Minizustände möglich, sogar Zustände zu untersuchen, die nahezu den gesamten Hauptspeicher eines Rechners einnehmen.

Des Weiteren dient ein Minizustand lediglich als Referenz auf einen Zustand. Dieser kann sich im Hauptspeicher oder auf einem weiteren Medium befinden, der Zugriff über einen Minizustand ermöglicht, ihn jederzeit zu lokalisieren.

Da die einzige Voraussetzung, die ein Minizustand erfüllen soll, eine konstante Größe ist, wurden noch weitere Informationen aufgenommen, die eine Prüfung erleichtern oder beschleunigen. Tabelle 3.1 zeigt den gesamten Aufbau eines Minizustands.

3.2.2 Speicherung der Zustände im Hauptspeicher

Wird ein neuer Zustand erzeugt, so wird dieser nicht sofort auf die Festplatte geschrieben, sondern verbleibt im Speicher. Der Speicherbereich, in dem Zustände aufbewahrt werden, wird *Zustand-Cache* genannt. Zusätzlich enthält der zugehörige Minizustand eine Referenz auf den Zustand. So kann direkt, über den Minizustand, auf einen Zustand zugegriffen werden. Der Zustand-Cache ist eine einfach verkettete Liste, die als Least-Recently-Used (LRU) Struktur gehandhabt wird. Wird ein Zustand der sich im Cache befindet, gelesen - sei es um ihn zu expandieren, oder ihn auf Duplikate zu prüfen - so wird seine Referenz in der Listei auf Position eins verschoben. Auf diese Weise wird sichergestellt, dass Zustände, die nicht gebraucht werden, ans Ende der Liste wandern. Ein Cache wurde so konzipiert, dass die Reihenfolge, in der Zustände ausgelagert werden, frei wählbar ist. So ist es möglich, über die Verwendung unterschiedlicher Techniken (z.B. *Last-In-First-Out (LIFO), First-In-First-Out*

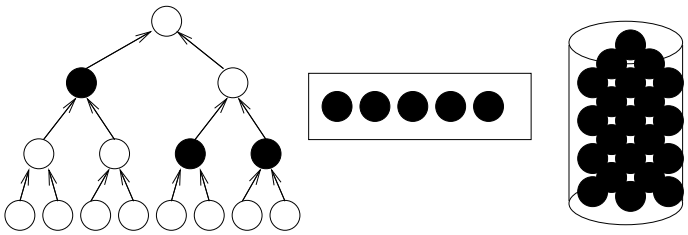

Abbildung 3.2: Externalisierung der Zustände auf Festplatte über einen Cache. Ausgefüllte Minizustände im Suchbaum repräsentieren Zustände im Cache, weiße Minizustände, Zustände die sich auf dem sekundären Medium befinden.

(FIFO), Least-Frequently-Used (LFU), Flush-When-Full (FWF), etc. (Tanenbaum, 1992)) den Cache an den Suchalgorithmus anzupassen.

Die Anzahl der Zustände, die der Zustand-Cache aufnehmen kann, ist, statisch (beim Start durch einen User) oder dynamisch (durch Expandierung) begrenzt. Wird die maximale Größe erreicht, so werden die Zustände, die sich am Ende der Liste befinden, also die die am längsten nicht benötigt wurden, gelöscht. Natürlich erfolgt vor dem Löschen eine Speicherung. Geschrieben wird aber nicht nur der zu löschende Zustand, sondern alle Zustände, die noch nicht gespeichert wurden. Die Tatsache, ob ein Zustand bereits auf die Festplatte abgelegt wurde, kann direkt aus dem jeweiligen Minizustand gelesen werden. Ist die *fid*, also die Nummer der Datei in der sich ein Zustand befindet, positiv, so wurde dieser bereits gespeichert. Wird ein Zustand gelöscht, so wird die Referenz *state*, im entsprechenden Minizustand, auf NULL gesetzt. Abbildung 3.2 zeigt eine schematische Darstellung der Funktionsweise.

3.2.3 Erweiterung der Collapse Kompression

Wie in Kapitel 2 bereits beschrieben, unterstützt StEAM Kompression. Es werden lediglich die Abschnitte der Vorgänger auf Gleichheit geprüft, so konnten identische Abschnitte mehrfach im Speicher existieren. Um den Speichernutzung vor einer Externalisierung zu optimieren, wurde die vollständige Funktionalität der "Collapse Compression" (Holzmann, 1997) implementiert (Edelkamp u. a., 2006) .

StEAM beinhaltet nun mehrere Cache Strukturen, die jeweils einen Abschnitt eines Zustandes aufnehmen können. Wird ein Zustand dem Zustand-Cache hinzugefügt, so wird er vorher in einzelne Abschnitte zerlegt. Jeder Abschnitt wird in den entsprechenden Cache eingefügt, der eine Zahl *id* zurückliefert, die den Abschnitt im Cache eindeutig identifiziert. Die *id* wird im Zustand gespeichert und die direkte Referenz auf den Abschnitt gelöscht. Nachdem dies für alle Abschnitte des Zustandes durchgeführt wurde, besteht der Zustand nun aus einem Ganzzahlvektor

$$\sigma = \langle P, BSS_{id}, Data_{id}, Text_{id}, MemPool_{id} \rangle.$$

Dieser Vorgang wird für jedes $p_{id} \in P$ wiederholt, so erhält man einen Ganzzahlvektor

$$p_{id} = \langle Reg_{id}, Stack_{id} \rangle.$$

Die Größe von σ ist nicht konstant da sich $|P|$ während der Ausführung ändern kann. σ wird dann in den Zustand-Cache eingefügt.

Bevor ein Abschnitt in den Cache eingefügt wird, muss eine *id* berechnet werden. Als *id* wird der nach Kapitel 2.4.1, für den entsprechenden Abschnitt berechnete partielle Hashwert $k(sec)$ mit

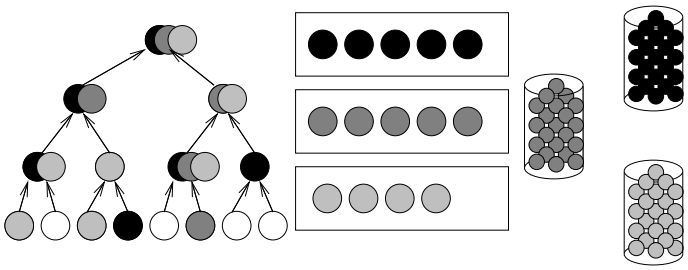

Abbildung 3.3: Externe Collapse Kompression. Gefärbte Zustände im Zustandsbaum repräsentieren Zustandsteile, die sich im Cache befinden. Weiße Zustandsteile wurden externalisiert.

sec ∈ [*BSS, Data, Text, MemPool, Reg, Stack*] verwendet. Tritt eine Kollision auf, so wird der errechnete Hashwert inkrementiert und als *id* verwendet. So ist sichergestellt, dass jeder Abschnitt eines Caches eine *id* erhält. Aufgrund von Kollisionen kann $k(sec)$ nicht als eindeutige Identifikationsnummer verwendet werden. Intern werden die Sektionen in einem AVL-Baum gespeichert, so dauert ein Einfügen höchstens $O(\log n)$, wobei n die Anzahl der Elemente im Cache bezeichnet.

Zusätzlich zu dem AVL-Baum verwaltet jeder Cache eine einfach verkettete Liste, die der im Zustand-Cache entspricht. Ein Hinzufügen läuft nach dem selben LRU-Schema, wie ein Hinzufügen eines Zustandes zum Zustand-Cache. So ist auch hier sichergestellt das Abschnitte die seltener gebraucht werden, in der Liste hinten aufbewahrt werden.

Um den internen Speicherverbrauch zu reduzieren, wurde die Größe dieser Caches beschränkt. Jeder Cache hat eine maximale Größe. Wird diese überschritten, werden Elemente, die sich am Ende der Liste befinden, externalisiert. Der AVL-Knoten verbleibt dabei im Speicher um die Information aufzunehmen, wo sich die Daten des Abschnitts auf der Festplatte befinden. Wird der Abschnitt wieder benötigt, kann er leicht rekonstruiert werden. Abbildung 3.3 zeigt ein Beispiel mit drei Cache Strukturen.

3.3 Der Algorithmus

Um die Nutzung der Minizustände zu ermöglichen, wurde der Algorithmus 2.1, zu dem Algorithmus 3.1 erweitert.

Die wichtigste Änderung ist die Modifizierung von *Open*. Während die priorisierte Liste vor der Änderung Referenzen auf Zustände aufnehmen konnte, werden jetzt Minizustände eingefügt. Dies hat den Vorteil, dass die gesamte Speicherstruktur eine viel geringere Größe aufweist als zuvor. Es wurde eine Funktion (**collapseDecompress**) implementiert, die aus einem gegebenen Minizustand einen Zustand generiert [Zeile 5]. Eine Funktion, die einen Zustand in die einzelnen Abschnitte zerlegt, diese in die entsprechenden Caches einsortiert, und einen Minizustand für diesen Zustand erzeugt ist **collapseCompress**. Wird nun ein Zustand generiert, wird der entsprechende Minizustand erzeugt, der den Zustand eindeutig beschreibt und in *Open* eingefügt. Wird ein Zustand zum Expandieren oder für Vergleiche benötigt, so wird dieser entsprechend seinem Minizustand und σ, in zwei Schritten, aus den einzelnen Abschnitten generiert.

Algorithmus 3.1 collapseCheck(Zustände prüfen unter Verwendung von Kompression)

Eingabe: Initialzustand: $\mathcal{I}$
Ausgabe: Einen Pfad von $\mathcal{I}$ zum Fehlerzustand, falls ein Fehlerzustand existiert.
Variablen: Prioritäts Warteschlange: *Open*;

```
 1: mini-state ms ← collapseCompress(I) {Zustand komprimieren und Minizustand erzeugen}
 2: insert(Open, ms)
 3: while Open ≠ ∅ do
 4:    mini-state ms ← getMin(Open)
 5:    States ← collapseDecompress(ms)
 6:    for all s' ∈ Expand(s) do
 7:       if Errorstate(s') then
 8:          return genPath(s') { Fehlerzustand zurück geben}
 9:       end if
10:       if checkForDuplicate(s') = false then
11:          mini-state ms ← collapseCompress(s')
12:          insert(Open, ms)
13:       end if
14:    end for
15: end while
16: return true {Gesamter Zustandsraum expandiert}
```

3.3.1 Komprimieren und Auslagern der Zustände

Wird ein Zustand erzeugt und nicht als Duplikat eingestuft, wird er, in der Funktion **collapseCompress**, in die einzelnen Abschnitte zerlegt und jeder Abschnitt, im entsprechenden Cache, gespeichert. Jeder Cache kann hierbei Daten auslagern um Platz für neue Abschnitte zu schaffen. Anschliessend wird der Ganzzahlvektor s_{id} des Zustandes in dem Zustand-Cache gespeichert, der wiederum auch Informationen auslagern kann. Zuletzt wird der Minizustand generiert und zurückgegeben. Ein Zustand besteht aus sechs Abschnitten wie sie in der Abbildung 2.1 dargestellt sind. Es sind jedoch nur fünf unterschiedliche Cache Strukturen nötig, da der Text Abschnitt, während der gesamten Prüfung, nicht verändert wird.

3.3.2 Dekomprimieren und Lesen der Zustände

Wird ein Zustand benötigt - sei es um ihn zu expandieren, oder um ihn mit einem anderen zu vergleichen - wird dieser über die Funktion **collapseDecompress** erzeugt. Algorithmus 3.2 zeigt den Pseudocode dieser Funktion. Hierbei ermittelt die Funktion, über die Referenz *state* im Minizustand ob sich der Vektor σ im Zustand-Cache befindet [Zeile 1]. Wurde σ ausgelagert, so wird dieser Vektor von der Festplatte gelesen, im Cache abgelegt und eine Referenz im *ms* erzeugt. Nun wird, anhand jeder einzelnen *sec*, der Zustand aus den Abschnitten zusammengesetzt, indem jede *sec* an den entsprechenden Cache übergeben wird, der den zugehörigen Abschnitt zurück gibt.

3.4 Zugriff auf das sekundäre Medium

Jeder Zustand wird nicht einzeln auf die Festplatte gespeichert. Es existieren zwei Methoden, die dem Benutzer zur Wahl stehen. Welche der beiden Methoden effektiver ist hängt von der Streuung der Hashfunktion, demnach auch vom Modell, ab. Streut die Haschfunktion gut, so müssen nur wenige Zustände miteinander verglichen werden. Diese geringe Anzahl der Zustände passt zumeist in den

Algorithmus 3.2 collapseDecompress (einen Zustand aus einem Minizustand erzeugen)

Eingabe: Minizustand des zu erstellenden Zustandes: *ms*
Ausgabe: Zustand *s* auf den *ms* verweist.

```
 1: if ms.state then
 2:     σ ← Read(ms, ⟨ms.fid⟩) {σ der aus der Datei ⟨ms.fid⟩ lesen}
 3:     insertIntoCache(ßtate",σ) {σ in dem Zustand-Cache ablegen}
 4:     ms.state ← σ
 5: else
 6:     σ ← ms.state {σ aus dem Zustand-Cache lesen}
 7: end if
 8: for all sec ∈ [ BSS, Data, Text, MemPool, Reg, Stack] do
 9:     s ← s ∪ getFromCache(sec, sec_id)
10: end for
11: return s
```

Hauptspeicher, es muss also selten auf die Festplatte zugegriffen werden. Hierbei bietet sich die *Single File Externalisierung* an. Handelt es sich um ein Modell, bei dem die Hashfunktion nur sehr schwach streut, sollte die *Hash File Externalisierung* verwendet werden.

3.4.1 Single File Externalisierung

Diese Methode wurde für das Speichern optimiert. Es wird nur eine Datei angelegt und alle Zustände in diese Datei gespeichert. Übersteigt die maximale Größe einer Datei des Betriebssystems, so werden weitere Dateien erzeugt. Die Datei in die momentan geschrieben wird, bleibt über die gesamte Suchdauer geöffnet. Der Schreib-/Lesekopf muss also nicht bewegt werden, falls zwischen zwei Schreibvorgängen kein Lesevorgang stattfindet. Stehen Zustände zum Schreiben bereit, so werden diese sequentiell ohne Sortierung geschrieben.

Soll ein Zustand zwecks Expandierung gelesen werden oder soll eine Kollision geprüft werden, wird der Schreib-Lesekopf an der richtigen Stelle positioniert und der Zustand gelesen. Da uns das System ermöglicht, die selbe Datei sowohl zum Lesen als auch zum Schreiben offenzuhalten, entsteht kein Zeitverlust durch ein Öffnen, oder Schließen der Datei.

Sobald aber ein Modell geprüft wird, bei dem es zu vielen Kollisionen kommt, ist diese Methode nicht zu empfehlen. Entsteht eine Kollision mit n Zuständen, so muss der Kopf bis zu n mal bewegt werden, um alle Zustände zu lesen. Man kann dem Phänomen natürlich durch eine bessere Haschfunktion entgegen wirken, jedoch braucht diese auch mehr Zeit um einen Hashwert zu errechnen. Ob sich dies rentiert, muss also für jedes Modell experimentell erprobt werden.

3.4.2 Hash File Externalisierung

Für Modelle, bei denen die Hashfunktion nicht gut streut, wurde diese Methode der Externalisierung entwickelt. Hier liegt die Priorität beim Lesen. Der Schreibzugriff ist deswegen um einiges langsamer als bei der ersten Methode, da hier die Zustände, nach Haschwert getrennt, in einzelnen Dateien gespeichert werden. Es ist also möglich, dass beim Schreiben von n Zuständen n Dateiöffnungen und n seek Befehle ausgeführt werden müssen, falls alle Zustände unterschiedliche Hashwerte aufweisen. Die Wahrscheinlichkeit für einen derartigen *worst case* wird durch eine Sortierung vor dem Schreiben reduziert.

Algorithmus 3.3 verdeutlicht die Funktionsweise dieser Methode. Die wichtigste Eigenschaft ist die Fähigkeit zu unterscheiden, ob ein Zustand gelesen werden soll, um ihn zu expandieren, oder ob

alle Zustände mit dem selben Hashwert zur Duplikatsprüfung benötigt werden [Zeile 5].

Wird nur ein Zustand gebraucht, so wird auch nur dieser gelesen. Nötigenfalls muss eine Datei geöffnet werden, was den Lesezugriff etwas langsamer macht als bei der Single File Externalisierung. Müssen aber viele Zustände gleichem Hashwert, gelesen werden, so zeigt sich die Stärke dieser Implementierung. Die Datei, die Zustände mit dem gesuchten Hashwert enthält, wird geöffnet und der Cache mit diesen Zuständen gefüllt [Zeile 15].

Algorithmus 3.3 read (lesen vom sekundären Medium, Hash File Externalisierung)

Eingabe: Minizustand des Zustands der gelesen werden soll: ms
Eingabe: Boolescher Wert, ob eine Duplikatsprüfung vorgenommen wird: $dupcheck$, offene Datei aus der gelesen wird: $file$
Ausgabe: Zustand der von der Festplatte gelesen wurde

```
 1: if getFilename(file) ≠ ms.fid then {falsche Datei offen}
 2:     file.close
 3:     open(file, ms.fid)
 4: end if
 5: if not dupcheck then {nur ein Zustand benötigt?}
 6:     setFilePosition(file, ms.fpt)
 7:     State s ← read(file)
 8: else
 9:     while not endOfFile(file) do
10:         State s'' ← read(file)
11:         collapseCompress(s'') {Zustand in den State-cache einfügen}
12:         if ms.fpt = getFilePosition(file) then {der gewünschte Zustand wird zurückgegeben}
13:             State s ← s''
14:         end if
15:     end while
16: end if
17: return s
```

3.5 Experimente

Es hat sich gezeigt, dass eine Externalisierung die Performance eines Modellprüfers signifikant steigert. Zwar verliert man immer Zeit beim Speichern auf die Festplatte, jedoch ist es möglich, komplexe Probleme zu überprüfen.

Hier zeigen wir einige Ergebnisse der Experimente, die durchgeführt wurden, um die Externalisierung zu testen. Beim ersten Experiment wurde das Modell der speisenden Philosophen getestet, beim zweiten das 8-Puzzle. Eine genaue Beschreibung der Modelle findet sich in Kapitel A. Wir präsentieren zwei Tabellen die aufzeigen sollen wie effektiv die Externalisierung arbeitet.

Graph 3.4 zeigt die Effizienz der Minizustände. Selbst bei einem Modell wie dem 8-Puzzle, dessen Zustandsgröße konstant 300 Bytes beträgt, wirkt sich der Effekt der Externalisierung signifikant aus. Während die vollständige Implementierung der Collapse-Kompression schon die Lösung schwerer Probleme ermöglicht, wird erst durch eine Externalisierung ein signifikant kleinerer Speicherverbrauch erzielt.

Tabelle 3.2 zeigt die Resultate, die erzielt wurden im einzelnen. Des Weiteren verdeutlicht diese Tabelle, wie wichtig eine gut streuende Hashfunktion für ein Modell ist. Da die Anzahl der Kollisionen mit der Komplexität des Modells steigt, steigt auch die Anzahl der Festplattenzugriffe, die nötig sind um eine Kollision zu prüfen. Bei dem Modell des 8-Puzzle sieht man, an den letzten beiden Instanzen

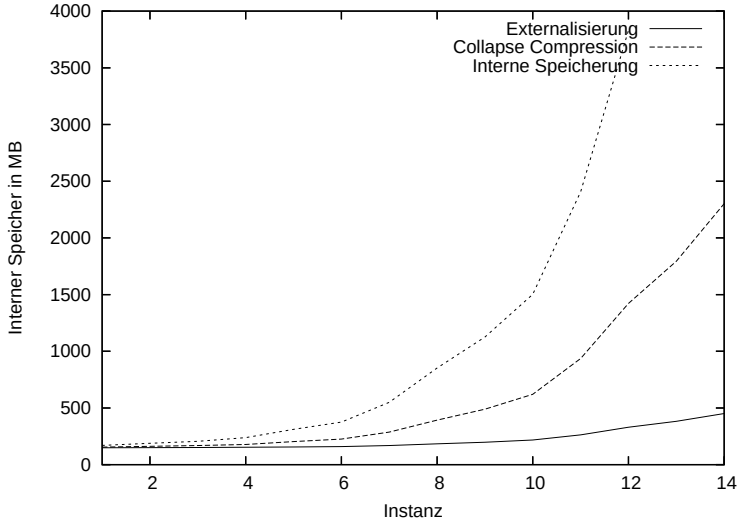

Abbildung 3.4: Interner Speicherverbrauch bei der optimalen Lösung einer 8-Puzzle Instanz

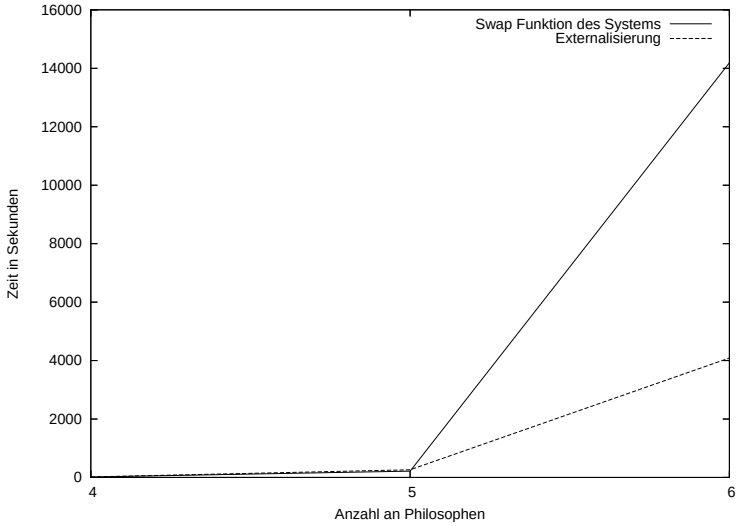

Abbildung 3.5: Vergleich zwischen Swap Funktion und Externalisierung

Instanz	Zustände	Tiefe	RAM	Festplatte	Zeit	Haschwerte	Kollisionen
Original Implementation							
(267105384)	437,400	150	853MB	-	4,049s	2,938	63,637,143
(267150384)	607,331	159	1126MB	-	5,650s	2,997	119,295,029
Collapse Kompression							
(267105384)	437,400	150	393MB	-	4,080s	2,938	63,637,143
(267150384)	607,331	159	489MB	-	5,705s	2,997	119,295,029
(267154380)	840,493	168	620MB	-	7,954s	2,985	222,463,964
(267154308)	1,405,027	177	936MB	-	13576s	2,982	598,825,442
Sinlge File Externalization							
(267105384)	437,400	150	184MB	111MB	6,000s	2,938	63,637,143
(267150384)	607,331	159	198MB	154MB	9,661s	2,997	119,295,029
(267154380)	840,493	168	217MB	213MB	15,293s	2,985	222,463,964
(267154308)	1,405,027	177	262MB	355MB	34,980s	2,982	598,825,442
(267104358)	2,284,022	186	330MB	577MB	80,438s	3,013	1,517,396,795
(267140358)	2,949,907	195	382MB	745MB	195,309s	3,018	2,487,367,370
Hash File Externalization							
(267105384)	437,400	150	184MB	123MB	6,401s	2,938	63,637,143
(267150384)	607,331	159	198MB	166MB	9,316s	2,997	119,295,029
(267154380)	840,493	168	217MB	225MB	13,604s	2,985	222,463,964
(267154308)	1,405,027	177	262MB	368MB	28,768s	2,982	598,825,442
(267104358)	2,284,022	186	330MB	487MB	68,723s	3,013	1,517,396,795
(267140358)	2,949,908	195	382MB	758MB	152,264s	3,018	2,487,367,370
(267148350)	3,898,007	204	451MB	997MB	217,912s	3,025	4,302,171,526

Tabelle 3.2: Vergleich verschiedener Methoden der Externalisierung beim 8 Puzzle.

deutlich, dass selbst wenn die Anzahl der Zustände nur um 30% ansteigt, die Anzahl der Kollisionen bereits um 70% steigen kann.

Des Weiteren sieht man einen Vergleich zwischen der *Single-File-Externalisierung* und der *Hash-File-Externalisierung*. Die erste Methode benötigt für eine Instanz, mit der Lösung in der Tiefe 195 schon fast 55 Stunden, mit der zweiten Methode ist es möglich eine Instanz, deren Lösung in Tiefe 204 liegt, in nur 5 Stunden mehr zu lösen.

Mit Hilfe der Externalisierung war es möglich das Protokoll der Speisenden Philosophen bis zur einer Instanz von 300 Philosophen zu untersuchen. Diese Prüfung dauerte, wie man der Tabelle 3.3 entnehmen kann, 112 Stunden und verbrauchte ca. 19 Gigabyte Speicher auf dem externen Medium.

Graph 3.5 verdeutlicht, dass die Externalisierung effizienter arbeitet als die Swap Funktion des Systems.

3.6 Resümee

Eine Externalisierung ermöglicht nicht nur die Prüfung großer Zustandsräume, sondern auch großer Zustände. Mit dem vorgestellten Ansatz war es möglich, weitaus größere Modelle zu untersuchen. Wie man den Experimenten entnehmen kann, ist dies sogar ohne signifikanten Zeitverlust möglich. Die besonderen Herausforderungen, die ein Software Modellprüfer an eine Externalisierung stellt, wurden durch unterschiedliche Methoden bewältigt. Aufgrund einer Optimierung der Speichernutzung, vor dem Auslagern, findet weniger Zugriff auf die Festplatte statt. Die Anwendung der Collapse-Kompression, in Verbindung mit einer LRU-Strategie, stellt sicher, dass Abschnitte eines Zustands,

die oft benötigt werden, im Hauptspeicher verbleiben. Schließlich kann der Zugriff auf das sekundäre Medium, über eine Auswahl aus den zwei vorgestellten Externalisierungsarten, optimiert werden.

Es ist vorstellbar, dass die Wahl der Externalisierungsart, oder auch der Auslagerungsstrategie eines Cache, automatisch vorgenommen wird. Hierzu würden Statistiken über eine Prüfung geführt und aus diesen, die optimale Methode ermittelt. Weiterhin ist eine Optimierung des Suchalgorithmus denkbar. Dieser könnte aus Informationen der Cache-Strukturen, Zustände bevorzugt expandieren, die im Hauptspeicher vorhanden sind.

Aufgrund der positiven Ergebnisse, die durch die Externalisierung erreicht wurden, baut diese Arbeit im weiteren Verlauf auf diesem Ansatz auf. Durch Auslagern der Zustände auf Festplatte “blockieren” diese keinen Hauptspeicher einzelner Rechenknoten. Ebenso wenig werden die Rechenknoten durch zusätzliches Empfangen und Ablegen ganzer Zustände belastet, sondern empfangen lediglich Minizustände.

Breitensuche

Philos.	Zustände	Tiefe	RAM	Festplatte	Zeit
Original Implementierung					
5	78,173	19	235MB	-	202s
Collapse Kompression					
5	78,173	19	235MB	-	235s
6	642,982	22	945MB	-	2,556s
Single-File-Externalisierung					
5	78,173	19	154MB	-	234s
6	642,982	22	188MB	894MB	9,255s
7	546,995	25	233MB	8,6GB	181,143s

Tiefensuche

Philos.	Zustände	Tiefe	RAM	Festplatte	Zeit
Original Implementierung					
20	48,998	3,898	284MB	-	345s
Collapse Kompression					
20	48,998	3,898	284MB	-	345s
25	76,954	5,123	403MB	-	804s
Single-File-Externalisierung					
20	48,998	3,898	159MB	202MB	365s
25	76,954	5,123	163MB	391MB	852s
50	304,929	10,938	186MB	3GB	12,146s

Bestensuche

Philos.	Zustände	Tiefe	RAM	Festplatte	Zeit
Original Implementierung					
50	9,465	353	232MB	-	351s
Collapse Kompression					
50	9,465	353	204MB	-	349s
100	36,430	703	559MB	-	5185s
Single-File-Externalisierung					
100	36,430	703	176MB	727MB	5,217s
150	80,895	1,053	201MB	2.3GB	25,766s
300	319,290	2,103	269MB	19GB	403,603s

Tabelle 3.3: Ergebnisse der Exploration des speisenden Philosophen Protokolls

Kapitel 4

Virtualisierung

Bei gleicher Umgebung
lebt doch jeder in einer anderen Welt.

Arthur Schopenhauer

In diesem Kapitel werden zwei Erweiterungen vorgestellt, die im Rahmen dieser Arbeit entstanden sind. Sie erweitern die Funktionalität von StEAM und sind für eine Parallelisierung unerlässlich.

4.1 Speicherzugriff

Programme fordern Speicherbereiche an. Findet eine solche Anforderung statt, überprüft das Betriebssystem, bzw. die virtuelle Maschine, ob genügend freier Platz im Hauptspeicher vorhanden ist, und liefert dem Programm die Anfangsadresse eines Speicherbereiches. Dieser Bereich bleibt so lange reserviert, wie er vom Programm benötigt wird. Wird das Programm beendet, oder gibt es den Bereich wieder frei, wird die Reservierung wieder aufgehoben und der Platz wird zur Verfügung gestellt.

Das geprüfte Programm kommuniziert nicht direkt mit dem Betriebssystem, sondern mit der virtuellen Maschine. Hier werden Speicherbereiche angefordert und wieder freigegeben. Da eine zusätzliche Verwaltung virtueller Speicheradressen in der virtuellen Maschine zunächst sowohl Zeit als auch Speicher gekostet hätte, wurde bei der Implementierung der ICVM darauf verzichtet. Dies bedeutet dass die virtuelle Maschine eine Anforderung ungefiltert an das Betriebssystem weitergibt. Reserviert das Programm einen Bereich im Hauptspeicher, so gibt die virtuelle Maschine die Adresse, die sie vom Betriebssystem erhalten hat, unverändert weiter. So verwendet das Programm, obwohl es in einer virtuellen Maschine ausgeführt wird, reale Speicheradressen des Betriebssystems.

Wechselt die virtuelle Maschine von (s, p_{id}) nach (s', p_{id}), mit $s \neq s'$, so muss sichergestellt werden, dass die Inhalte der angeforderten Speicherbereiche ebenfalls angepasst werden. Da das Programm jedoch reale Speicheradressen verwendet, sind die angeforderten Speicherbereiche fest im Hauptspeicher verankert. Deshalb müssen die Inhalte aus dem *MemoryPool* an die entsprechenden Stellen kopiert werden. Abbildung 4.1 verdeutlicht diese Funktionsweise schematisch.

Diese Methode ist, während das Programm ausgeführt wird, sehr effizient und speichersparend, birgt jedoch für eine Parallelisierung einen Nachteil. Überträgt man ein Programm von Knoten i auf Knoten j, mit $i \neq j$, so müsste man auf Knoten j an denselben Stellen Speicher reservieren wie auf Knoten i. Dies ist aber nicht möglich, da die Reservierung von Hauptspeicher allein dem Betriebssystem erlaubt ist.

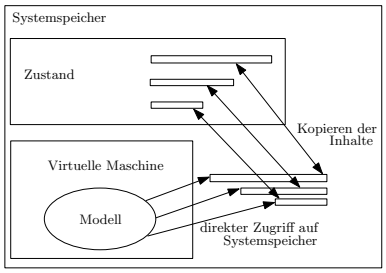

Abbildung 4.1: Direkter Zugriff auf Speicherbereiche. Das Modell greift direkt auf Speicherbereiche ausserhalb der virtuellen Maschine zu.

Während der Entwicklung des Algorithmus wurden zwei Lösungsansätze in Betracht gezogen, die in folgenden erörtert werden.

4.1.1 Konvertieren der Adressen vor der Ausführung

Eine auf den ersten Blick sehr effiziente Lösung, ist, die Adressen im Programm vor der Ausführung auf einem Rechenknoten zu konvertieren. Diese Lösung erscheint deswegen effizient, da sie lediglich bei der Übertragung zwischen zwei Rechenknoten Zeit benötigt; zur Laufzeit werden keine Aktionen vorgenommen.

Das Programm kann die benötigten Speicherbereiche auf Rechenknoten i anfordern und die erhaltenen Anfangsadressen werden von StEAM zwischengespeichert. Wird der Zustand auf Rechenknoten j, mit $j \neq i$, übertragen, so werden benötigte Speicheradressen angefordert und im Programm nach den bisherigen Adressen gesucht. Diese werden ersetzt.

Angenommen, das Programm fordert Speicher an und erhält die Adresse x als Startadresse des zur Verfügung gestellten Speicherbereiches zugewiesen. StEAM speichert diese Adresse im MemoryPool zusätzlich zu den Inhalten im Speicherblock, der bei x beginnt. DIe Adresse x wird aber auch im Programm abgelegt, um diesem Zugriff auf diesen Speicherbereich zu ermöglichen. Falls das Programm auf einen anderen Knoten übertragen wird, wird die Stelle im Programm geändert, an der x liegt.

Da StEAM aber ein Programm ausführt, ändern sich die Adressen in diesem zwangsläufig. Das bedeutet, dass Adressen im Programm verlegt oder Daten an zusätzlichen Stellen im Modell abgelegt werden.

Nachfolgend werden einige Situationen in einem Programm geschildert. Ein Programm fordert Speicher an und arbeitet mit den zugewiesenen Adressen. Nach jedem Arbeitsschritt wird aufgezeigt was getan werden müsste, falls das Programm auf einen anderen Knoten übertragen würde.

1. Das Modell fordert Speicher an, und erhält die Adresse a zugewiesen. Diese Adresse wird im Programm an Position y abgelegt. Das bedeutet für das Programm, es darf auf einen Speicherbereich, der bei a beginnt, zugreifen.

   - Findet nun eine Übertragung auf einen anderen Rechenknoten statt, dann wird dort, der Speicherbereich an einer anderen Adresse $b \neq a$ reserviert. Es ist demnach nötig die Adresse an Position y auf den Wert b zu setzen.

2. Nehmen wir an es findet keine Übertragung statt und im Programm wird eine weitere Variable initialisiert. Diese Variable bekommt zufällig den Wert a zugewiesen und wird im Programm

an Position z abgelegt. Zusätzlich wird die reale Adresse a, an einer weiteren Position im Programm, abgelegt.

- Fände nun eine Übertragung statt, müsste das gesamte Modell nach dem Wert a durchsucht und ersetzt werden. Jedoch darf die Position z nicht modifiziert werden, da diese einen Wert einer Variablen aber keine Adresse darstellt.

3. Bei der weiteren Ausführung, angenommen es fand immer noch keine Übertragung statt, wird der Wert a an Position y modifiziert. Da hier ein Zugriff auf eine bestimmte Stelle im angeforderten Speicherbereich erfolgen soll.

   - Suchen nach a ist nun aussichtslos, es müsste nach allen Werten zwischen a und a+(Größe des angeforderten Bereichs) gesucht werden.

4. Nun wird die Variable an Position z im Modell als Zeiger eingesetzt und auf den Speicherblock der bei Adresse a beginnt zugegriffen.

   - Der Modellprüfer müsste die Verwendung einer Variable, zu jedem Zeitpunkt, kennen und wissen, ob es sich um einen Wert handelt, der unverändert bleibt oder eine Speicheradresse, die angepasst werden muss.

Bei diesem Beispiel handelt es sich um Funktionen die nahezu in jedem Programm vorkommen können. Es wird deutlich das ein Anpassen von Speicheradressen in einem ausgeführten Programm nahezu unmöglich ist.

4.1.2 Virtuelle Adressierung

Die zweite Option ist eine Konvertierung der Speicheradressen, auf die das Programm zugreift, zur Laufzeit. Dadurch wird das Modell in einer “black box” Umgebung ausgeführt und kann so jederzeit auf andere Knoten übertragen werden.

Angeforderte Speicherbereiche

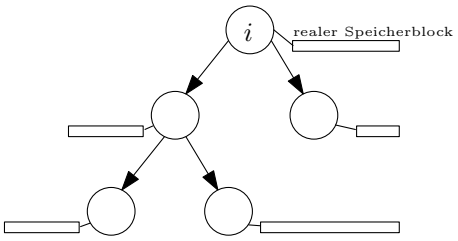

Abbildung 4.2: MemoryPool in StEAM. Sortiert nach virtueller Anfangsadresse. Jeder Knoten enthält eine Referenz auf den realen Speicherblock und die Größe dieses Speicherblockes.

Fordert das Modell Speicher an, gibt die virtuelle Maschine , falls die Anforderung beim Betriebssystem erfolgreich war, eine *virtuelle Adresse* an das Programm weiter. Eine vitruelle Adresse ist eine Zahl v die über eine Funktion $\phi(v)$ einer realen Adresse zugeordnet ist. Beide Adressen werden im MemoryPool gespeichert. Zusätzlich wird auch die Größe l des angeforderten Bereiches im MemoryPool gespeichert. So stellt der AVL-Baum die Funktion ϕ dar.

Eine Überprüfung einer Adresse verläuft nun in zwei Schritten. Da die virtuellen Adressen fortlaufend durchnummeriert sind, wird im ersten Schritt geprüft, ob die Adresse, auf die zugegriffen wird, im Bereich der virtuellen Adressen liegt. Hardwareadressen werden also in Zeit O(1) erkannt. Ist dies der Fall, so wird der MemoryPool nach der entsprechenden realen Adresse durchsucht. Der AVL-Baum ist nach den virtuellen Startadressen sortiert. Falls für einen Knoten mit der virtuellen Startadresse i gilt:

$$i \leq v \leq (i + l),$$

so enthält dieser Knoten die reale Adresse zu v. Abbildung 4.2 stellt einen MemoryPool dar.

Hierzu ein Beispiel mit realen Zahlen:

1. Sei $v = 23$
2. Im AVL-Baum wird ein Knoten gefunden mit $i = 20$ und $l = 5$ (dieser Knoten enthält einen Speicherbereich, dessen virtuelle Startadresse 20 ist und der die Länge 5 hat
3. Sei die reale Startadresse 1705, so ist die gesuchte reale Adresse $(23 - 20) + 1705 = 1708$

Die v zugehörige reale Adresse wird in Zeit $O(\log n)$ erkannt wird, wobei n die Anzahl der angeforderten Speicherbereiche darstellt.

Ein großer Vorteil dieser Methode ist die Tatsache, dass Speicherinhalte nicht mehr Kopiert werden müssen. Greift ein Programm auf reale Adressen zu, so muss, vor der Ausführung, dafür gesorgt werden, dass die Speicherinhalte richtig sind. Virtuelle Adressen können aber auf den gesamten Hauptspeicher abgebildet weden, demnach auch direkt auf die Inhalte im MemoryPool.

Globale Variablen und der Programmzähler

Neben dem MemoryPool ist es nötig, den Zugriff auf die restlichen Speicherbereiche zu kontrollieren. Hiebei wurde jedoch die Funktionsweise der virtuellen Maschine ausgenutzt.

Bei der Initialisierung legt die virtuelle Maschine die ausführbare Datei in einem Speicherblock ab. Darin liegen die 3 Abschnitte *Text*, *Data* und *BSS* direkt nebeneinander. Der Programmzähler wird mit der ersten Adresse dieses Speicherblocks initialisiert. Während der Ausführung wird der Programmzähler nur relativ verändert. Dies bedeutet, er wird um einen bestimmten Betrag erhöht oder verringert. Der Zugriff auf Variablen, die sich in den Abschnitten befinden, erfolgt relativ zum Programmzähler.

Beginnt der *Text* Abschnitt auf Rechenknoten i an Position x und auf Rechenknoten j, mit $i \neq j$ an Position y, so wird zu dem Wert des Programmzählers, bei einer Übertragung von i nach j, $(y - x)$ addiert.

Der Stack

Ein weiterer Speicherbereich auf den kontrolliert zugegriffen werden muss, ist der Stack. Jedoch reicht es nicht, den Wert des Stackzählers anzupassen, da auf dem Stack Variablen abgelegt werden. Die Position dieser Variablen muss dem Programm natürlich bekannt sein. Auch hier wurde eine Zwischenschicht geschaffen, ähnlich dem MemoryPool.

Wird eine virtuelle Maschine initialisiert, wird ein Speicherblock angefordert. Dieser Speicherblock hat die Größe des Stacks. Bei der ICVM wird nun der Stackpointer auf die größte Adresse dieses Speicherblocks initialisiert. Da der Stackpointer, bei jedem Ablegen von Daten auf dem Stack, dekrementiert wird, belegt der Stack nun Adressen in diesem Speicherblock. Die Änderung besteht darin,

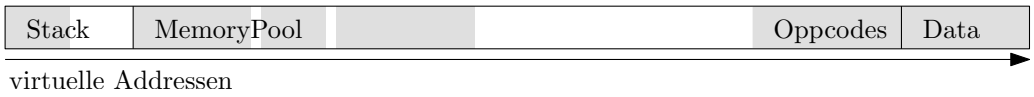

Abbildung 4.3: Aufteilung der virtuellen Adressen. Grau unterlegte Abschnitte kennzeichnen belegten Speicher, in den einzelnen Abschnitten.

den Stackpointer nicht als Hardwareadresse zu initialisieren, sondern diesen direkt auf die Größe des Stacks zu setzten. Nun wird, bei jedem Zugriff auf den Stack, zu dessen virtueller Adresse die Startadresse des Speicherbereichs addiert, an der sich der Stack befindet. Abbildung 4.3 verdeutlicht die Anordnung der einzelnen Sektionen.

4.1.3 Implementierung und Analyse

Bei der Implementierung wurde der vorhandene Quelltext der virtuellen Maschine erweitert. Einerseits wird eine virtuelle Adresse in eine reale übersetzt, dies geschieht im Algorithmus 4.1, andererseits werden, falls das Programm Speicher anfordert, reale Adressen angefordert und mit den virtuellen Adressen im MemoryPool gespeichert.

Bei einem Zugriff auf den Stack wird geprüft, ob dieser auch gültig ist. Findet ein Zugriff unterhalb des Stackpointers statt, so muss es sich um einen ungültigen Zugriff handeln, ebenso beim Zugriff auf den Bereich zwischen der letzten angeforderten Speicherstelle und dem Data-Abschnitt.

Algorithmus 4.1 getAddress (reale Adresse ermitteln)

Eingabe: Adresse die geprüft werden soll: a ; Größe des Stack: *StackSize* ; Reale Anfangsadresse des Stack: *realStack* ; Maximale virtuelle Adresse im MemoryPool: *maxMem*
Ausgabe: Reale Adresse an der die Daten zu finden sind, sonst Fehler

```
1: if a ≤ stackPointer then {Unzulässiger Stack Zugriff}
2:     return ILLEGAL STACK ACCESS
3: else if a ≤ StackSize then {Adresse auf dem Stack?}
4:     a ← a + realStack
5: else if a ≤ maxMem then {Adresse im MemoryPool}
6:     return getRealAddressFromPool(a)
7: else if a ≥ startData then {Adresse im erlaubten Bereich}
8:     return a
9: end if
10: return ILLEGAL MEMORY ACCESS
```

Erkennen von unzulässigen Speicherzugriffen im MemoryPool

Fordert das Modell Speicher an, so erhält es die nächstmögliche virtuelle Adresse. Hierzu wird der Wert der größten virtuellen Adresse um 2 inkrementiert und zurückgegeben. So entsteht zwischen den einzelnen Bereichen eine *Lücke*. Versucht das Modell nun auf eine dieser Lücken zuzugreifen, ergibt dies einen Fehler, da es sich hierbei um einen nicht angeforderten Speicherbereich handelt. So kann man erkennen, ob ein Arrayindex außerhalb des ihm zugewiesenen Bereichs liegt. Algorithmus 4.2 zeigt die Funktion zu Ermittlung einer realen Adresse aus dem MemoryPool.

Algorithmus 4.2 getRealAddressFromPool (reale Adresse im Memory Pool ermitteln)

Eingabe: Adresse die geprüft werden soll: a
Ausgabe: Reale Adresse an der die Daten zu finden sind, sonst Fehler

```
1:  avlNode n ← root
2:  while n ≠ NULL do
3:      if n.key > a then
4:          n ← n.left
5:      else if n.key + n.size < a then
6:          n ← n.right
7:      end if
8:      return n.start + (a − n.key) {n.start enthält die erste reale Adresse des Speicherblocks}
9:  end while
10: return SEGMENTATION FAULT {Adresse zwar im Pool, jedoch nicht angefordert}
```

4.1.4 Analyse der Zugriffszeit

Die ICVM erlaubt, in der Originalimplementation, einen Zugriff in $O(1)$ auf jeden vom Programm benötigten Speicherbereich.

Die Zugriffszeit auf den Stack wurde um $O(1)$ erhöht. Hier ist lediglich eine Addition und ein Vergleich nötig, um die reale Adresse aus der virtuellen zu errechnen. Zugriffe auf die Sektionen sind ebenfalls um einen Vergleich verlangsamt. Die größte Verzögerung findet beim Zugriff auf Adressen im MemoryPool statt. Hierzu wird zuerst ein Vergleich durchgeführt, der feststellt, dass sich die dieser virtuellen Adresse zugeordnete reale Adresse im MemoryPool befindet. Nun muss der AVL-Baum traversiert werden, um den entsprechenden Speicherknoten zu finden. Dies dauert $O(\log n)$, da der MemoryPool als AVL-Baum balanziert ist, wobei n die Anzahl der angeforderten Speicherbereiche darstellt. Wurde der Knoten gefunden, so ist nun noch eine Addition nötig, um die reale Adresse zu ermitteln.

Somit ergibt sich für die größtmögliche Verzögerung, die Formel

$$V = O(1) + O(1) + O(\log n) + O(1) = O(\log n).$$

4.1.5 Experimente

Abbildung 4.4 zeigt die Geschwindigkeit einer Expandierung. Untersucht wurde die 8-Puzzle Instanz ⟨2 7 0 1 6 5 3 8 4 ⟩. Das Experiment wurde auf einem Rechner mit AMD Opteron MP 852 (2.6 GHz) CPU, 16 Gigabytes Hauptspeicher und einer Maxtor 160GB SATA (gespiegelt/ RAID 1) Festplatte durchgeführt. Wie wir sehen, entsteht durch die virtuelle Adressierung eine Beschleunigung in der Exploration des Zustandsraumes. Dies kann mit der Tatsache erklärt werden, dass die Speicherbereiche für die virtuelle Adressierung nicht kopiert werden müssen.

Werden reale Adressen benutzt, so werden bei einem Wechsel des Zustandes alle angeforderten Speicherbereiche in den Zustand kopiert, während Speicherinhalte aus dem neuen Zustand in die virtuelle Maschine kopiert werden. Dies gewährleistet das Daten an der richtigen Stelle vorhanden sind, falls das Programm auf sie zugreift. Greift das Modell aber nicht auf angeforderten Speicher zu. So müssen die Inhalte trotzdem kopiert werden. Bei der virtuellen Adressierung verbleiben die Informationen im Zustand; das Kopieren entfällt. Greift das Modell nicht auf angeforderten Speicher zu, so entsteht keine Verzögerung.

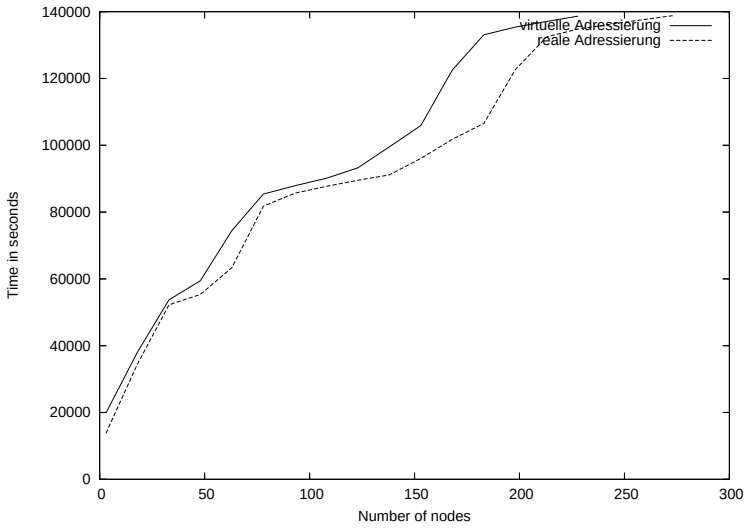

Abbildung 4.4: Geschwindigkeit der Expandierung mit und ohne virtuelle Adressierung.

4.2 Zugriff auf eine virtuelle Maschine

Soll Maschinencode ausgeführt werden, so wird eine Einheit benötigt, die diesen ausführen kann. Im Fall von StEAM handelt es sich um die ICVM. Eine virtuelle Maschine führt jedoch nur Code aus, für den sie konzipiert wurde. Theoretisch ist es jedoch irrelevant in welcher Sprache das Modell entwickelt wurde. Denn ein Modellprüfer durchsucht lediglich den Zustandsraum nach vordefinierten Fehlerzuständen. Trennt man also die virtuelle Maschine so weit von dem eigentlichen Prüfer ab, dass er nur über vordefinierte Funktionen mir der virtuellen Maschine kommuniziert, kann man die virtuelle Maschine und damit die Sprache, die der Modellprüfer akzeptiert, austauschen.

In dieser Arbeit wurde zwischen StEAM und der ICVM ein Interface geschaffen über das StEAM die virtuelle Maschine steuert und Informationen über die Maschine erhält. Somit ist es möglich, eine virtuelle Maschine, die ein anderes CPU Modell simuliert, um die benötigten Methoden zu erweitern und so Maschinencode für unterschiedliche CPU Typen zu prüfen.

4.2.1 Zweck einer virtuellen Maschine

StEAM ist in der Lage, ein nichtdeterministisches Programm auszuführen und in dessen Ausführung nach unerwünschten Zuständen zu suchen. Um das Verhalten von Programmen zu untersuchen, in denen mehrere Prozesse laufen, deren Ausführung voneinander abhängig ist, muss man die Prozesse in allen möglichen Reihenfolgen ausführen. Das Programm sollte also - möglichst in einer kontrollierten Umgebung - ausgeführt und bei jedem Durchlauf die Reihenfolge in der die Prozesse ausgeführt werden, geändert. Es entsteht ein Zustandsraum S, der untersucht wird. In diesem Zustandsraum entspricht jeder Zustand $s \in S$, einem Zustand des Programms im Speicher, also einem Abbild des Speichers des jenigen Rechners, auf dem das Programm ausgeführt wird. Wird das Programm in einer virtuellen Maschine ausgeführt, wie es bei StEAM der Fall ist, so entspricht ein Zustand s einem Abbild der virtuellen Maschine zu dem Zeitpunkt, an dem s erzeugt wurde. Jede Kante entspricht einer *Übergangsfunktion*. Die Reihenfolge, in der die Prozesse ausgeführt werden, wird durch die Auswahl der Übergangsfunktion bestimmt, die zu einem Prozess gehört. Um mehrere die-

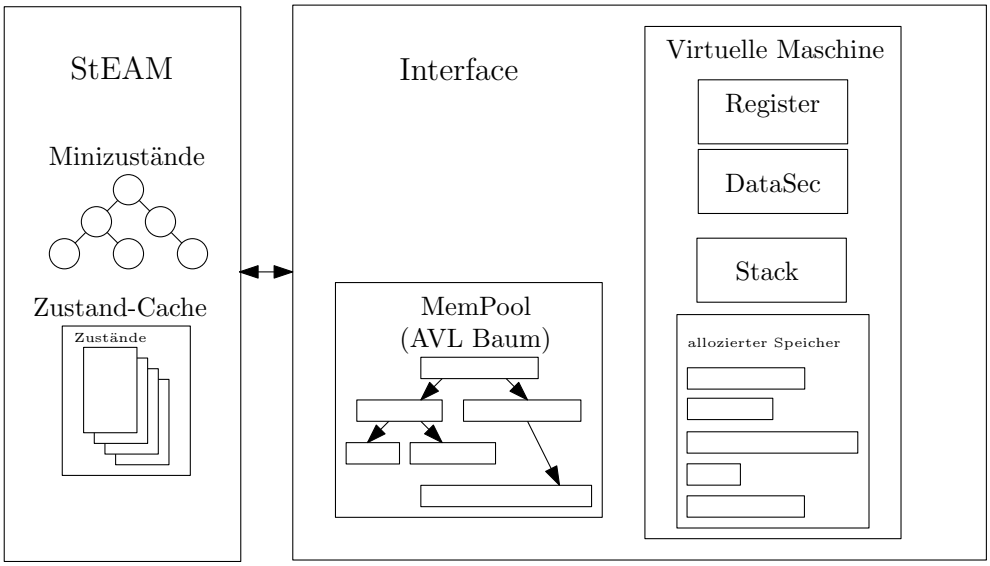

Abbildung 4.5: **Kontrolle der Virtuellen Maschine über ein Interface.** Alle Informationen werden über ein Interface an die virtuelle Maschine übergeben. Das Interface erkennt Fehlerzustände und meldet diese an StEAM. Real angeforderte Speicherbereiche befinden sich im Interface, die virtuelle Maschine greift lediglich auf diese zu.

ser Übergangsfunktionen an einem Verzweigungspunkt ausführen zu können, muss das Programm wieder an dieser Stelle gestartet werden. Dies ist nur möglich, falls das Programm in einer virtuellen Maschine ausgeführt wird.

4.2.2 Identifikation der benötigten Methoden

Um ein Modell zu prüfen, kontrolliert StEAM die virtuelle Maschine. Diese Kontrolle erfolgt über eine Schnittstelle.

Wir definieren zuerst eine *Fehlerklassifikation* als eine Zahl $1 \leq e \leq n$, wobei jedem e eine Verletzung einer Fehlerdefinition zugeordnet wird. Dabei stellt n die Anzahl aller Fehlerarten dar, die von StEAM erkannt werden.

Startet man die virtuelle Maschine, so lädt diese eine ausführbare Datei und startet die Ausführung. Ohne einen Eingriff von außen würde diese Maschine nun den Code ausführen, die Auswahl der einzelnen Prozesse einem internen Algorithmus überlassen, und, falls kein Fehler auftritt, zu einem Endzustand gelangen. Hieraus ergeben sich drei Funktionen, die die Maschine StEAM zur Verfügung stellen muss:

pause() : StEAM wird in der Lage sein, die virtuelle Maschine jederzeit anzuhalten. Dies ist insbesondere dann nötig, falls ein neuer Zustand erzeugt werden soll.

resume() : Die virtuelle Maschine setzt die Ausführung fort und gibt den Wert **true** zurück falls sie erfolgreich war, sonst eine Fehlerklassifikation

setProzess(*p*) : An jeder Verzweigung muss StEAM einen Prozess bestimmen, der ausgeführt werden soll.

Eine weitaus komplexere Kommunikation zwischen dem Prüfer und der Maschine ist die Erstellung eines Zustandes. Der Modellprüfer muss sowohl Schreib-, als auch Lesezugriff zu sämtlichen Teile der virtuellen Maschine haben. Hierzu enthält das Interface zwei Methoden:

setState(s, p_{id}) : Diese Methode versetzt die virtuelle Maschine in einen Zustand s, der übergeben wurde und kopiert die Register und den Stack des Prozesses p_{id}.

getState() : Diese Methode gibt den aktuellen Zustand der virtuellen Maschine zurück.

Es besteht jedoch auch eine Kommunikation zwischen dem Programm und dem Modellprüfer, die über die virtuelle Maschine hergestellt wird. StEAM unterstützt sogenannte *Tags*. Dies sind spezielle Befehle zur Steuerung des Modellprüfers, die im Quelltext vorhanden sind. Wird das Modell nun kompiliert, werden diese Befehle in Maschinencode umgewandelt, der natürlich von der verwendeten virtuellen Maschine abhängt.

handleTag() ist eine Methode von StEAM, die einen Tag ausführt, sie gibt 0 zurück falls erfolgreich, sonst eine Fehlerklassifikation.

Diese Tags müssen also dem Interface bekannt sein, um die virtuelle Maschine anzuhalten und StEAM über ein Auftreten eines solchen Tags zu benachrichtigen. Eine Funktion die einen solchen Befehl, welcher eventuell aufgetreten ist, an StEAM weitergibt, ist:

getTag() : gibt ein StEAM Befehl zurück, falls aufgetreten, sonst **false** .

Im Zuge der in Kapitel 4.1 entstandenen Erweiterung, wurde StEAM ebenfalls um weitere Methoden ergänzt. Da die virtuelle Maschine auf virtuelle Adressen zugreift, ist es nötig, den MemoryPool an das Interface zu übergeben. StEAM stellt einige Methoden zur Speicherverwaltung, zur Verfügung. Nachfolgend werden Methoden aufgelistet, die die virtuelle Maschine während einer Ausführung aufruft.

steamMalloc(n) : gibt die erste virtuelle Adresse zurück, fordert beim Betriebssystem einen realen Speicherbereich der Größe n an, und weist diesen im MemoryPool der zurückgegebenen virtuellen Adresse zu.

steamFree(a) : gibt den angeforderten Bereich, der der virtuellen Adresse a zugeordnet ist frei. Ist a keine virtuelle Adresse wird ein Fehler ausgegeben.

steamGetAddress : siehe Algorithmus 4.1.

4.2.3 Implementierung des Interfaces

Aus diesen Überlegungen entstand eine Klasse, die alle nötigen Methoden für eine Steuerung der virtuellen Maschine enthält. Algorithmus 4.3 generiert aus einem Zustand die Nachfolger und prüft ob diese fehlerfrei sind. Nachdem die virtuelle Maschine in (s, pid) versetzt wurde, wird p_{id} als nicht iteriert vermerkt. Nun wird die virtuelle Maschine gestartet. Diese führt die Opcodes, die einer Zeile Quelltext entsprechen, aus. Ist die Ausführung erfolgreich [Zeile 5] so wird geprüft, ob ein *Tag* aufgetreten ist [Zeile 6]. Falls bei der Ausführung des *Tags* eine Fehlerklassifikation zurückgegeben wurde, wird abgebrochen, sonst p_{id} als iteriert markiert. Nun wird ein neuer Zustand erzeugt, der zur Rückgabeliste hinzugefügt wird.

Zusätzlich zu den im Algorithmus 4.3 genannten Methoden wird die virtuelle Maschine durch diese Klasse auch initialisiert. Dies bedeutet, die virtuelle Maschine lädt das ausführbare Modell und führt es aus bis die *main* Methode des Modells gefunden wurde. Der erzeugte Zustand wird als Initialzustand $\mathcal{I}$ an StEAM übergeben.

Algorithmus 4.3 expand (Ausführen von Opcodes in der virtuellen Maschine)

Eingabe: Zustand der expandiert werden soll: s
Ausgabe: Liste aller Nachfolger, falls erreichbar, sonst eine Fehlerklassifikation

```
 1: list ← ∅ {Liste für die Zustände initialisieren}
 2: for all p_id ∈ s do
 3:    setState(s, p_id)
 4:    p_id.iterated = false
 5:    if resume() then
 6:       e ← handleTag(getTag())
 7:       if e > 0 then
 8:          return e
 9:       end if
10:       p_id.iterated = true
11:       list ← list ∪ getState()
12:    end if
13: end for
14: if ∀p_id ∈ s : p_id.iterated = false then
15:    return DEADLOCK
16: end if
17: return s
```

4.3 Resümee

Die Implementierung dieses Interfaces ermöglicht es, die virtuelle Maschine gegen beliebige andere Maschinen auszutauschen. Jedoch sollten die Änderungen, die an einer solchen Maschine durchgeführt werden müssten, nicht unterschätzt werden. Das Interface an eine virtuelle Maschine anzupassen, erfordert sehr detailiertes Wissen über den Aufbau dieser Maschine und ihrer Funktionsweise. Allerdings zeigt die Implementierung eines solchen Interfaces, dass es möglich ist, einen Modellprüfer von der virtuellen Maschine zu trennen, und dass der Austausch der virtuellen Maschine von jemandem vorgenommen werden kann, der kein Wissen über den Modellprüfer selbst besitzt. Dies erhöht den Wert des Modellprüfers, da Entwickler lediglich ihr Wissen über eine virtuelle Maschine benötigen, um diese an StEAM anzupassen. Sie werden nicht gezwungen, sich mit den Interna eines Modellprüfers zu beschäftigen.

Die Methode der virtuellen Adressen bringt ebenfalls sehr viele Vorteile mit sich. In dieser Arbeit wurde sie hauptsächlich implementiert, um einen Zustand auf einen anderen Rechenknoten zu übertragen. Sie ermöglicht des Weiteren, ein Pausieren der Ausführung über einen Neustart von StEAM hinaus. Speichert man die Zustände dauerhaft auf Festplatte und terminiert den Modellprüfer, so braucht man nun nur noch die Struktur des Minizustandsbaumes zu speichern, und diesen bei einem erneuten Start wiederherzustellen.

Ebenso findet diese Methode weitere Fehler. Durch die Abstände, die künstlich zwischen angeforderten Bereichen erzeugt werden, wird nicht mehr Speicher angefordert als nötig, aber es ist möglich, Indexüberläufe abzufangen.

Erweiterungen der virtuellen Adressierung erlauben, weitere Fehlersituationen zu überprüfen. So ist es z.B. möglich, zu prüfen, ob ein Lesezugriff an einer Stelle stattfindet, an die zuvor Daten geschrieben wurden. Hierzu müsste lediglich eine Speicherstruktur geschaffen werden, in der durchgeführte Speicherzugriffe abgelegt werden. Um eine Verzögerung durch diese Prüfung zu minimieren, bietet sich hierfür ein Bitvektor an.

Zugriffe auf den Stack können genauer untersucht werden indem, es verboten wird, auf Speicher-

stellen zuzugreifen, auf die nur die virtuelle Maschine zugreifen darf. So könnte in einem Modell überprüft werden, ob es dem Programm möglich ist, Rücksprungadressen zu überschreiben - eine der Ursachen für *Buffer Overflow* Attacken.

Denkbar wäre auch eine Einführung von den im MemoryPool verwendeten *Lücken* im Stack, so wäre es möglich, Zugriffe auf Arrays zu überprüfen, die sich auf dem Stack befinden.

Kapitel 5

Parallelisierung

> Nothing is particularly hard
> if you divide it into small jobs.
>
> Henry Ford

Die Prüfung komplexer Programme benötigt sowohl viel Rechenzeit, als auch Speicherplatz. Die im Kapitel 3 genante Methode begegnet zwar dem Problem des Speicherbedarfs, stellt jedoch keinerlei Verbesserung der Prozessorleistung zur Verfügung. Möchte man demnach auch die Dauer einer Prüfung optimieren, kommt man nicht umhin, mehrere Prozessoren, oder sogar mehrere Rechner, zu verwenden.

Für eine Parallelisierung stehen zwei Systeme zur Auswahl. Während in einem *Multicore* System lediglich die Prozessoren mehrfach vorhanden sind, werden im *Cluser System* mehrere Rechner über ein Netzwerk verbunden. Ein Cluster-System kann leichter und weiter skaliert werden als ein Multicore-System, da es praktisch keine Begrenzung für die Anzahl an Rechnern in einem Netzwerk gibt. Die Anzahl an Prozessoren, die in einem Rechner verbaut werden können, ist jedoch begrenzt. Eine Parallelisierung auf einem Multicore-System ist deshalb zwar denkbar, jedoch nicht so leicht skalierbar wie eine Parallelisierung auf einem Cluster-System. In der Abbildung 5.1 werden beide Systeme, bildlich verglichen.

Auf Grund der Skalierbarkeit, aber auch auf Grund dessen, dass eine Parallelisierung für ein Cluster-System auf ein Multicore-System übertragbar ist, wurde innerhalb dieser Arbeit ein Cluster-

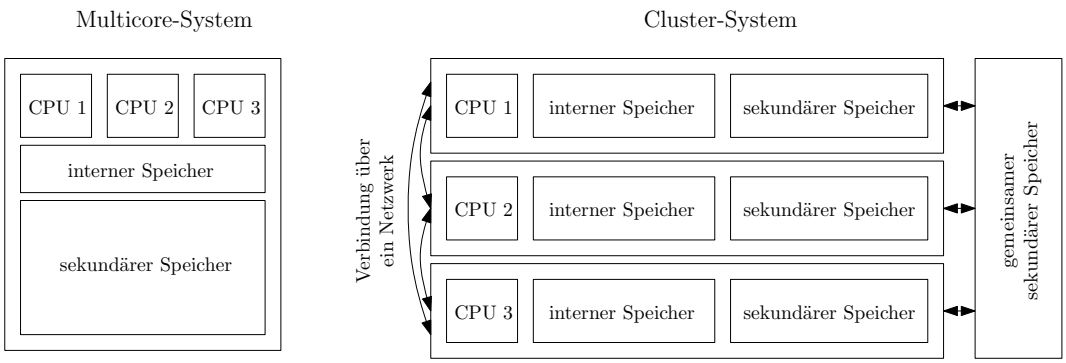

Abbildung 5.1: Vergleich zwischen einem Multicore (links) und einem Cluster-System (rechts).

System verwendet.

Durch die gemeinsame Nutzung mehreren Rechner, steht mehr Hauptpeicher zur Verfügung, als in einem einzelnen verwendet werden könnte. Des Weiteren steht natürlich mehr Prozessorzeit zur Verfügung, da diese mehrfach vorhanden sind. Jedoch sollte man bedenken, dass die Kommunikation zwischen unterschiedlichen *Knoten*, wie jeder Rechner im einem Cluster genannt wird, im Vergleich zur internen Kommunikation eines Rechners sehr langsam ist. Transferiert man nun viele Informationen zwischen den Knoten, so kann dies Zeit- und Platzvorteil zunichte machen. Alle Prozessoren eines Multicore-Systems greifen auf denselben Speicher zu. Hier entstehen demnach kleinere Zeitverluste durch Transaktion von Daten als bei einer Transaktion über ein Netzwerk.

Wie bereits erwähnt ist eine Parallelisierung, die für ein Cluster-System entwickelt wurde, auf einem Multicore-System ausführbar. Hierzu werden mehrere Instanzen der parallelisierten Software gestartet und das Betriebssystem weist jeder Instanz Speicherbereich und Prozessor zu. Eine Methode, die für einen Multicore Computer entwickelt wurde, ist jedoch zumeist nicht ohne weiteres auf ein Cluster-System portierbar, da diese auf Speicherbereichen beruht, auf den beide Prozessoren Zugriff haben.

Die Erweiterungen, die an StEAM vollzogen wurden, wurden variabel gehalten. So ist es möglich, die Verteilungsfunktion zu variieren. Eine *Verteilungsfunktion* ist eine Abbildung $d : S \rightarrow \mathbb{Z}$, die einen Zustand auf eine natürliche Zahl abbildet, die einem Knoten entspricht.

5.1 Einbindung der Externalisierung

Aufgrund der positiven Erfahrungen, die mit der Externalisierung gemacht wurden, wurde diese zur Grundlage für die Parallelisierung. Wie wir im Kapitel 3 gesehen haben, konnten komplexere Probleme geprüft werden, nachdem nicht benötigte Informationen auf ein sekundäres Medium ausgelagert wurden. Des Weiteren würde ohne die Externalisierung die Anzahl oder Größe der Zustände stärker begrenzt werden.

Erfahrungen aus der Externalisierung sind auf zwei Wegen in die Parallelisierung eingeflossen.

1. Jeder Knoten ist in der Lage, Systemzustände, die gerade nicht benötigt werden, auf ein internes, oder externes sekundäres Medium, z.B. eine Festplatte, auszulagern.

2. Die Möglichkeit, Zustände auf ein gemeinsames sekundäres Medium auszulagern, wird genutzt um eine Zwei-Wege-Kommunikation zu schaffen. Hierbei belegen Zustände in der Kommunikationswarteschlange keinen Hauptspeicher.

5.1.1 Effiziente Nutzung interner Festplatten

Zur Steigerung der Effizienz eines jeden Knoten, ist dieser in der Lage, Zustände auf eine Festplatte zu speichern. Dies geschieht nach der Methode der externen Kompression, wie sie in Kapitel 3.2.3 beschrieben wurde. Der hier vorgestellte Algorithmus unterstützt sowohl Cluster Systeme, bei denen auf jedem Knoten mehrere interne Festplatten, installiert sind, als auch Systeme auf denen Festplatten lediglich über ein Netzwerk zugänglich sind. Besitzen jedoch die Knoten eines Clusters interne Festplatten so wirkt sich das positiv auf die Dauer der Prüfung aus, da auf diese Informationen ausgelagert werden können, die nur lokal von einem Knoten benötigt werden.

Verbleibt ein neuer Zustand auf dem Knoten, werden dessen Abschnitte in dem entsprechenden Cache gespeichert. Erreicht ein Cache die maximal zulässige Anzahl an Elementen, so lagert dieser Elemente auf Festplatte aus. Da der Speicherort für jeden Cache separat konfiguriert werden, ist

es möglich, jedem Cache eine eigene interne Festplatte zuzuweisen. Dies steigert die Zugriffszeit. Natürlich kann auch hier ein Speicherort verwendet werden, der im Netzwerk liegt.

Wird bei der externen Kompression ein gemeinsames, über ein Netzwerk angebundenes Medium verwendet, verlangsamt sich die Zugriffszeit auf dieses Medium mit steigernder Anzahl an Knoten. Dieser Effekt tritt nicht auf, falls interne Festplatten verwendet werden. Des Weiteren haben interne Festplatten den Vorteil, dass ein Auslagern der Zustände aufgrund eines Cache Überlaufs die Kommunikation zwischen den Knoten nicht verlangsamt.

5.2 Kommunikation zwischen den Knoten

Selbst wenn es möglich ist, den Hauptspeicher mehrerer Knoten zu benutzen, so ist dieser dennoch begrenzt. Sollen jedoch komplexe Programme untersucht werden, so benötigt man viel Speicher. Einerseits, um viele Zustände zu speichern, anderseits, um große Zuständen zu expandieren. Sei M die Größe des Hauptspeichers eines Knoten, und M_s die Anzahl Bytes, die StEAM benötigt, so ist die maximale Anzahl Bytes die ein Programm, während der Ausführung belegen darf $M - M_s$. Diese Größe konnte natürlich nur durch Auslagern der Zustände auf ein sekundäres Medium erreicht werden.

Bei den Vorüberlegungen zur Parallelisierung war eines der wichtigen Kriterien, diesen Vorteil nicht zu verlieren. Dies bedeutet, dass weder Zustände, die von einem Knoten empfangen werden, noch Zustände, die gesendet werden, Speicher belegen dürfen. Die Herausforderung hierbei ist klar ersichtlich: Die Zustände, die empfangen werden, belegen keinen Speicher. Hierzu wurden zwei Taktiken durchdacht. Die erste bestand darin, einen Zustand zu empfangen und dann, falls nötig, diesen Zustand auszulagern. Die zweite Möglichkeit, die auch implementiert wurde, war die Kommunikation auf zwei Wegen. Hierbei werden Minizustände über eine Netzwerkverbindung verschickt, während Zustände auf ein gemeinsames sekundäres Medium geschrieben werden. Wird nun ein Zustand benötigt, muss der Knoten auf das gemeinsame sekundäre Medium zugreifen und diesen von dort lesen. Auf einem Multicore-System wäre dies gemeinsame Medium, entweder die Festplatte, oder, falls genügend Speicher vorhanden ist, ein s.g. *Ramdrive*, eine virtuelle Festplatte, die im Speicher abgelegt wird. Im folgenden werden beide Methoden detailiert verglichen.

5.2.1 Verschicken von Zuständen

Die Methode, den Zustand über eine Netzwerkverbindung zu übertragen, mag auf den ersten Blick effizient erscheinen, da sie keinen Zugriff auf das gemeinsam genutzte sekundäre Medium benötigt.

Die Zustände werden empfangen und stehen sofort zur weiteren Verarbeitung zur Verfügung, um sie auf Duplikate zu prüfen oder zu expandieren.

Ein Knoten, der Zustände empfängt, ist jedoch keinesfalls untätig. Im idealen Fall ist jeder Knoten gleichermaßen damit beschäftigt, Zustände zu expandieren und den Zustandsraum zu durchsuchen. Das Empfangen von Zuständen verlangsamt diese Expandierung lediglich. Bei einer größeren Menge an Rechenknoten steigt der Aufwand für den Empfang von Zuständen und deren Prüfung auf Duplikate. Diese muss zwangsläufig stattfinden, um Speicher nicht unnötig zu verbrauchen.

Hinzu kommt die Tatsache, dass StEAM in der Lage ist, Zustände zu expandieren, die den gesamten freien Speicherbereichs füllen. Würden nun Zustände zwischen den Knoten transferiert werden, die sehr groß sind, wäre ein Knoten nur dann in der Lage zu arbeiten, wenn er vorher alle empfangenen Zustände auslagert.

Auch eine verzögerte Duplikatseliminierung (engl. *delayed duplicate detection*) (Korf, 2003) ist in einer solchen Umgebung nur schwer möglich. Da bei dieser Art der Duplikatsprüfung zeitweise

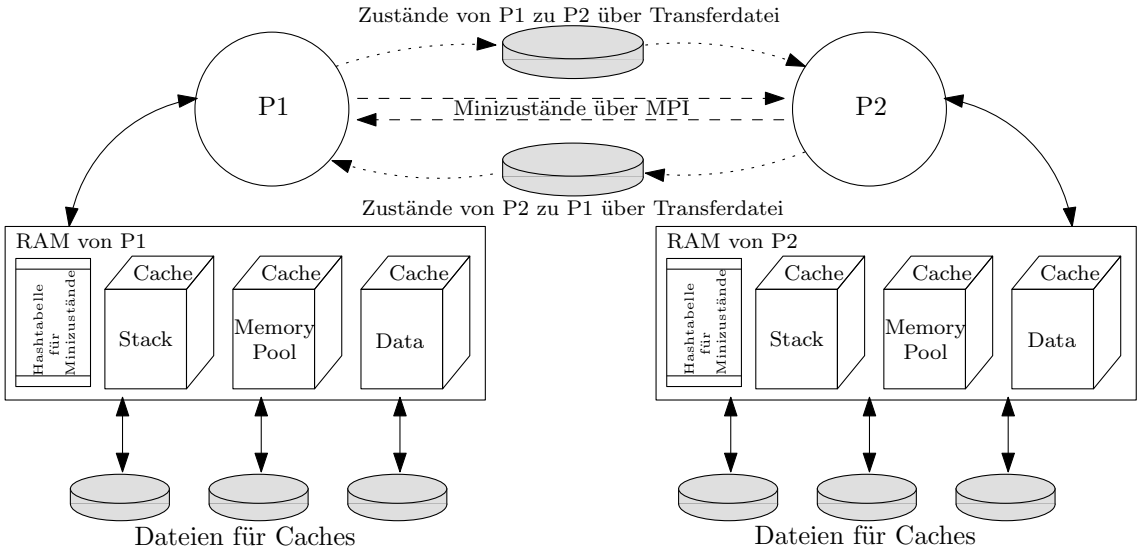

Abbildung 5.2: Zwei-Wege-Kommunikation mit internen Cache Strukturen. Zwei Knoten versenden Zustände und Minizustände über getrennte Kanäle.

identische Zustände vorhanden sind, wird unnötig Hauptspeicher belegt. Es ist also notwendig, Duplikate direkt nach dem Empfang zu eliminieren, selbst wenn es sich hierbei um Zustände handelt, die in einem Bereich des Zustandsraums liegen, der zu diesem Zeitpunkt nicht expandiert wird.

Ein weiteres Argument für diese Methode ist die nicht vorhandene Notwendigkeit, auf ein gemeinsames, im Vergleich zur Kommunikation über ein Netzwerk, langsames Medium zugreifen zu müssen. Die Zustände werden direkt im Speicher empfangen und stehen zur Verfügung. Dies ist aber nur so lange von Vorteil, bis der Hauptspeicher des Empfängerknotens gefüllt ist. Erreicht der Speicherverbrauch die Grenze an der ein empfangener Zustand s nicht mehr in den Speicher passt, so wird folgender Vorgang gestartet:

1. Ein vorhandener Zustand s' wird ausgelagert (Zugriff auf die interne Festplatte).
2. Der empfangene Zustand s wird einer Duplikatsprüfung unterzogen.
3. Möglicherweise werden mehrere Zustände, zum Vergleich, von der Festplatte gelesen. Diese verdrängen weitere Zustände aus dem Speicher.
4. Durch den Empfang wurde mindestens ein Zustand ausgelagert. Wird einer dieser ausgelagerten Zustände nun benötigt (z.B. s') so wird dieser wieder gelesen. (Zugriff auf die interne Festplatte)

5.2.2 Zwei-Wege-Kommunikation

Bei dieser Methode werden nur die Minizustände direkt über ein Netzwerk geschickt. Während der Minizustand im Hauptspeicher empfangen wird, liegt der entsprechende Zustand, in einer sogenannten *Transferdatei*, auf der gemeinsamen Festplatte. Hier bietet sich natürlich die verzögerte Duplikatseliminierung geradezu an. Zustände werden nur zu dem Zeitpunkt auf Duplikate geprüft, zu dem sie expandiert werden. Auf der gemeinsam genutzten Festplatte kann demnach ein Zustand mehrmals vorhanden sein, sei es, da er von verschiedenen Knoten expandiert wurde oder sei es, weil ein Knoten einen Zustand mehrmals erreicht hat.

Da der Minizustand alle nötigen Informationen enthält, um ihn im Suchraum einzuordnen, werden Zustände nur gelesen, wenn sie expandiert werden sollen. Der Empfang eines Minizustands verbraucht aufgrund seiner Größe und einfachen Beschaffenheit weitaus weniger Ressourcen als der Empfang eines Zustands. Ein Minizustand wird auch direkt in *Open* eingefügt und verbleibt dort bis der zugehörige Zustand s expandiert werden soll. Nun wird s in den Speicher geladen und geprüft, ob ein identischer Zustand schon mal expandiert wurde. Falls dies nicht der Fall ist, wird er expandiert.

Empfängt ein Knoten, dessen Hauptspeicher vollständig gefüllt ist, einen Zustand s, so wird folgender Vorgang gestartet.

1. Der Minizustand wird empfangen und in *Open* einsortiert (kein Zugriff auf eine Festplatte).
2. Erst wenn s expandiert werden soll, wird er von der gemeinsamen Festplatte gelesen und auf Duplikate geprüft.
3. Möglicherweise werden mehrere Zustände, zum Vergleich, von der Festplatte gelesen.Diese verdrängen weitere Zustände aus dem Speicher.
4. Da s expandiert werden soll, wurde er nicht ausgelagert und kann nun verwendet werden (kein Zugriff auf eine Festplatte nötig).

Auf diese Weise wird die Anzahl der Zugriffe auf die Festplatte minimiert. Weiterhin wird der Knoten nicht durch unnötige Zustandsvergleiche verlangsamt. Zustände, die im Suchraum in Bereiche fallen, die später expandiert werden sollen, werden in *Open*, mit einer kleineren Priorität einsortiert, und so später, wenn überhaupt, in den Speicher geladen. Abbildung 5.2 veranschaulicht die Zwei-Wege-Kommunikation.

5.2.3 Duplikatseliminierung

Wird aus *Open* ein Minizustand gelesen, der von einem anderen Knoten stammt, so wird der zugehörige Zustand aus der Transferdatei geladen. Ob es sich hierbei um ein Duplikat handeln kann, wird vor dem Expandieren überprüft. Hierzu wird in der Hashtabelle nachgesehen, ob bereits ein Zustand mit dem gleichen Hashwert vorhanden ist. Die Hashwerte werden beim Erzeugen eines Zustands errechnet, um den Vorteil, den das *inkrementelle Hashing* (Mehler und Edelkamp, 2006) bietet, nicht zu verlieren. Ist bereits ein Minizustand mit dem selben Hashwert vorhanden, werden die Zustände verglichen. Falls sie gleich sind, wird der Neue verworfen und der nächste Zustand aus *Open* verwendet.

5.3 Verteilungsfunktionen

Sämtliche Suchalgorithmen blieben unverändert. Die Parallelisierung dient lediglich zur Verteilung der Zustände auf einzelne Knoten. Diese empfangen einen Zustand und expandieren ihn nach einem zuvor festgelegten Algorithmus. Um eine möglichst flexible Umgebung zu schaffen, können alle Informationen die ein Zustand enthält, zur Verteilung genutzt werden. Es wurden mehrere Verteilungsfunktionen $d : S \rightarrow \mathbb{Z}$ getestet, die dem Index eines Zustand $s \in S$ auf eine natürliche Zahl abbilden, die einem Knoten entspricht. Nachfolgend sei K die Anzahl der verfügbaren Knoten.

Eine gute Verteilungsfunktion stellt sicher, dass in einer "Kette" $s_1 \rightarrow s_2 \rightarrow s_3$, wobei s_2 aus s_1 und s_3 aus s_2 entsteht, nur eine Übertragung zwischen den selben Knoten stattfindet. Wird Zustand s_1 auf Knoten k_1 expandiert, dann Zustand s_2 auf Knoten k_2, gefolgt von Zustand s_3 auf Knoten k_1, so ist

dies eine denkbar schlechte Konstellation. Aufgrund dieser Tatsache sollte eine Verteilungsfunktion nur einen Teil des Zustandes in Betracht ziehen. Dieser Teil sollte so gewählt sein, dass bei einer Expandierung nur dann Zustände übertragen werden, wenn es unbedingt nötig erscheint.

5.3.1 Verteilung entsprechend einem Hashwert

Bei dem ersten Versuch den Zustandsraum zu partitionieren, wurde der Hashwert des gesamten Zustandes genutzt. Um die Zustände einzelnen Knoten zuzuweisen, wurde die Funktion

$$d = k(s) \mod K$$

genutzt, wobei $k(s)$ der nach Kapitel 2.4.1 errechnete Hashwert des Zustandes s ist.

Diese Art der Verteilung erwies sich in der Praxis als äußerst ineffizient. Wie die Experimente zeigen, übersteigt bei dieser Methode der Zeitverlust, der durch die Kommunikation entsteht, den Zeitgewinn in der Expansion. Dies ist relativ einfach zu erklären: Da in die Verteilung weder Informationen über die Entstehung des Zustandes einfließen, noch berücksichtigt wird, dass Zustände möglichst selten übertragen werden sollten, erfolgt hier die Verteilung nahezu zufällig.

5.3.2 Verteilung entsprechend einem partiellen Hashwert

Um ein effizienteres Verteilen der Zustände zu erreichen, schränken wir die Berechnung des Hashwertes nun auf einen Abschnitt des gesamten Zustands ein. Die Verteilungsfunktion sei dabei die selbe wie in Kapitel 2.4.1.

Wählt man nun i und j so aus, dass sich der dazwischen liegende Bereich nur ändert, wenn signifikante Änderungen am Gesamtzustand stattfinden, so findet weitaus weniger Kommunikation zwischen den Knoten statt. Um diese Arbeit zu untermauern, wurden Experimente durchgeführt, in denen unterschiedliche Teilbereiche eines Zustands zur Errechnung des Hashwertes herangezogen wurden. Besonders positive Ergebnisse werden erzielt, falls man den MemoryPool zur Verteilung der Zustände nutzt.

5.3.3 Verteilung entsprechend der Tiefe

Die beschriebenen Ansätze betrachten lediglich einen Zustand und errechnen aus diesem den entsprechenden Knoten. Eine weitere Möglichkeit der Verteilung ergibt sich, falls man die Struktur des Zustandsraums in Betracht zieht. So ist es bei der Tiefensuche von Vorteil, die Zustände entsprechend ihrer Tiefe zu verteilen.

Diese Methode wurde für Multicore-Systeme bereits in (Holzmann und Bosnacki, 2006) vorgestellt. Sie musste an ein Cluster-System angepasst werden. Dies bedeutet, dass die Art der Verteilung übernommen wurde, die Kommunikation findet jedoch auf zwei Wegen statt.

Sei m die maximale Höhe eines Tiefenabschnitts, der von einem Knoten bearbeitet werden darf, so ergibt sich k aus

$$k = (t * m) \mod K,$$

Wobei t darstellt, in welcher Tiefe sich der Zustand befindet.

Erreicht Knoten i die maximale Tiefe t mit $((t * m) \mod K) = i$ so sendet er Minizustände, die tiefer im Zustandsraum liegen, an Knoten $i + 1$ weiter. Während Knoten $i + 1$ weiter in die Tiefe expandiert, expandiert Knoten i nur Zustände für die $((t * m) \mod K) = i$ gilt. Hierdurch wird nicht mehr in die Tiefe gesucht, sondern in die Breite und es kann zu anderen Ergebnissen kommen als

bei der Tiefensuche. Jedoch sind die Fehlerpfade, die entstehen können, höchstens kürzer als die der Tiefensuche.

Des Weiteren eignet sich diese Methode nur für die Tiefensuche. Wird Breitensuche benutzt, so würde immer nur ein Knoten expandieren.

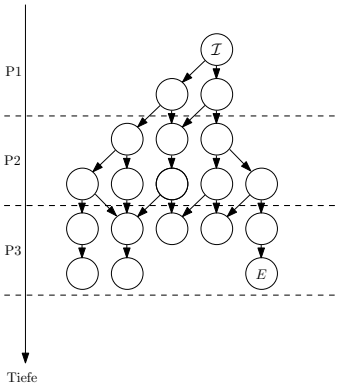

Abbildung 5.3: Die Zustände werden, entsprechend ihrer Tiefe im Zustandsraum, einzelnen Knoten zugewiesen. $\mathcal{I}$ ist der Initial Zustand, E ein Fehler Zustand.

5.3.4 Verteilung entsprechend einem heuristischen Wert

Eine weitere Art, die Zustände einzelnen Knoten zuzuweisen, ergibt sich durch das Heranziehen des heuristischen Wertes. Denn hier wird die Position im Zustandsraum berücksichtigt. Diese Verteilungsfunktion ist nur bei gerichteter Suche möglich. Wurde die Entfernung zum Fehlerzustand errechnet, so benutzt man die Formel aus dem vorherigen Kapitel, um den Knoten zu ermitteln, an den der Zustand gesendet wird. Abbildung 5.4 zeigt die Aufteilung des Zustandsraums, der, im Gegensatz zur Verteilung entsprechend der Tiefe, vertikal aufgeteilt wird.

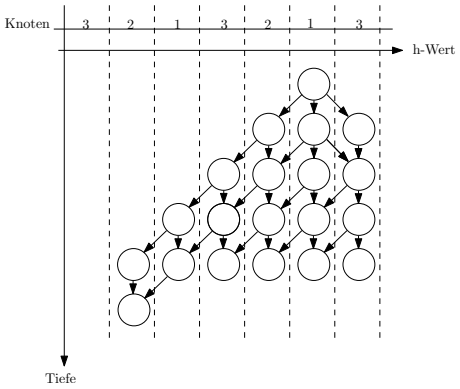

Abbildung 5.4: Vertikale Aufteilung des Suchbaums entsprechend dem heuristischen Wert

5.4 Implementierung

5.4.1 Netzwerkkommunikation

Für die Übertragung eines Minizustands zwischen zwei Knoten wurde Open-MPI* (Burns u. a., 1994; Squyres und Lumsdaine, 2003) gewählt. MPI ist ein effizientes Interface um Nachrichten, zwischen Knoten in einem Netzwerk, zu übermitteln. Nachdem man die Datei `mpi.h` mit Hilfe des `#include` Befehls hinzugefügt hat, stehen verschiedene Funktionen zum Senden oder Empfangen von Nachrichten zur Verfügung. Die Knoten werden anhand ihrer *MPI-ID's* identifiziert, die automatisch von MPI zugewiesen werden. Der erste Knoten hat die MPI-ID 0, bei jedem weiteren Knoten wird die MPI-ID um eins inkrementiert. Ermöglicht wird sowohl ein sogenanntes *blockierendes*, als auch ein *nicht blockierendes* Senden oder Empfangen. Empfangene Nachrichten werden in einer *MPI-Queue* gespeichert.

Beim blockierenden Senden bzw. Empfangen wartet der Knoten bis eine Übermittlung statt gefunden hat. Der Vorteil dieser Methode ist ihre Geschwindigkeit. Sobald eine Nachricht in der MPI-Queue vorhanden ist, steht diese zur Verfügung. Blockierenden Empfang nutzen alle Knoten, bis zur Ankunft des ersten Zustandes der expandiert werden soll.

Stehen Zustände bereit, wird die nicht blockierende Kommunikation verwendet. Dies bedeutet das im Hintergrund, während der Hauptprozess den Zustand s expandiert, ein Minizustand empfangen und nach der Expandierung von s, in *Open* eingefügt wird.

Gesendet wird natürlich immer nicht blockierend, um ein Weiterarbeiten zu ermöglichen.

5.4.2 Verteilung der Zustände

Um zu prüfen, welcher Knoten einen Zustand expandieren soll, wird die Funktion **checkOwner**(s) verwendet. Wobei lediglich vorausgesetzt wird das diese Funktion einen Zustand s erhält und eine Zahl zurückliefert, die einem Knoten entspricht. So ist es sehr einfach die Zustände, auf unterschiedliche Arten, auf die Knoten zu verteilen. Um eine neue Strategie bei der Verteilung der Zustände zu testen, muss lediglich diese Funktion angepasst werden.

5.4.3 Der Algorithmus

Der Algorithmus unterscheidet zwischen einem *ROOT* und den restlichen *Client* Knoten. ROOT startet die Expandierung, indem er seine virtuelle Maschine initialisiert und einen Initialzustand $\mathcal{I}$ erzeugt.

Algorithmus 5.1 veranschaulicht die gesamte Prüfung unter der Verwendung mehrerer Knoten. Wieder wurde Algorithmus 3.1 erweitert, um, zusätzlich zur Externalisierung, auch die Parallelisierung zu unterstützen. Nachfolgend werden die einzelnen Ansätze, anhand des Pseudocodes, näher erläutert.

Minizustände zur Übermittlung von Status Nachrichten

Alle Informationen werden über Minizustände übertragen. Um einen Minizustand als Nachricht, von denen drei unterschiedliche verwendet werden, zu kodieren wird ein negativer Wert als *fid* verwendet.

Ein Knoten ist *idle*, falls *Open*und die MPI-Queue leer sind. Der ROOT Knoten verwaltet eine Liste aller *idle* Knoten. Es ist nötig eine solche Liste zu verwalten, da hieran zu erkennen ist, ob der

*http://www.open-mpi.org

Algorithmus 5.1 search (Verteilte Suche mit zwei Kanal Kommunikation)

Eingabe: Initialzustand: $\mathcal{I}$; Process ID: *PID*; Anzahl Minizustände, die gelesen werden darf: MAX_MSG.
Ausgabe: Ein Pfad von $\mathcal{I}$ zum Fehlerzustand, falls ein Fehlerzustand existiert.
Variablen: Priorisierte Warteschlange *Open*; Set *idle_nodes*; Boolean *idle_reported*; Minizustand *ms*;

```
 1: if PID = ROOT then
 2:    idle_nodes ← ∅
 3:    insert(Open, I) {Initialzustand in Open einsetzen}
 4: else
 5:    idle_reported ← false
 6: end if
 7: while true do
 8:    if MPI.queue ≠ ∅ then
 9:       counter ← 0
10:       for all mini-state ms ∈ MPI.queue do
11:          if ms.msg = TERMINATE then
12:             return true
13:          else if ms.msg = IDLE then {Ein Knoten meldet er sei unbeschäftigt}
14:             idle_nodes ← idle_nodes ∪ ms.creator
15:          else if ms.msg = BUSY then {Ein Knoten, der unbeschäftigt war, meldet, er ist nun beschäftigt}
16:             idle_nodes ← idle_nodes \ ms.creator
17:          else
18:             if (PID ≠ ROOT) and (idle_reported = false ) then
19:                send (ms.msg ← BUSY to ROOT) {ROOT informieren, Knoten ist wieder beschäftigt}
20:                idle_reported ← true
21:             end if
22:             insert(Open, ms) { Minizustand in Open einsetzen }
23:          end if
24:          if counter++ > MAX_MSG break { Schleife abbrechen}
25:       end for
26:    end if
27:    if Open = ∅ then
28:       if (PID = ROOT) and (idle_nodes enthält alle Knoten) then
29:          broadcast (ms.msg ← TERMINATE) an alle Knoten
30:          return true
31:       else if (PID ≠ ROOT) and (idle_reported = false ) then
32:          send (ms.msg ← IDLE to ROOT)
33:          idle_reported ← true {Verhindern das wiederholt IDLE Nachrichten gesendet werden}
34:       end if
35:    else
36:       if (State s ← getNext(Open) = NULL) continue {Open wurde durch getNext geleert}
37:       for all s′ ∈ Expand(s) do
38:          if Errorstate(s′) then
39:             broadcast (ms.msg ← TERMINATE) an alle Knoten
40:             return genPath(s′) { Fehlerpfad zurück geben}
41:          end if
42:          new_owner ← checkOwner(s′)
43:          if new_owner ≠ PID then
44:             Append s′ to File ⟨PID, new_owner⟩ {s′ in Transfer Datei von PID zu new_owner speichern}
45:             send (s′.mini-state) to new_owner process {Minizustand an entsprechenden Knoten senden}
46:          else if checkForDuplicate(s′) = false then
47:             mini-state ms ← collapseCompress(s′) {Zustand komprimieren und Minizustand erzeugen}
48:             insert(Open, ms) {Minizustand in Open einfügen}
49:          end if
50:       end for
51:    end if
52: end while
```

Zustandsraum bereits komplett expandiert wurde. Sendet ein Knoten eine *idle* Nachricht, an ROOT [Zeile 32], so legt dieser die MPI-ID des Knotens in seiner Liste ab. Ist *Open* auf ROOT leer, so prüft dieser nach, ob alle MPI-ID's in der *idle*-Liste vorhanden sind. Falls ja, kann die Prüfung erfolgreich beendet werden [Zeilen 26 bis 29]. Jeder Knoten verwaltet eine *idle_reported* Variable, um ein wiederholtes Senden der *idle*-Nachricht zu vermeiden.

Empfängt ein Knoten, der idle war, einen neuen Minizustand, dann sendet er eine *busy*-Nachricht an ROOT [Zeile 19] und dieser entfernt die MPI-ID des Knotens aus der *idle*-Liste.

Alle Knoten können die Prüfung jederzeit terminieren. Wurde ein Fehlerzustand gefunden, sendet der Knoten eine *terminate*-Nachricht an alle übrigen Knoten und die Suche terminiert [ZEILE 39].

Empfangen und Expandieren der Zustände

Da die **while** Schleife nun nicht stoppen darf, falls *Open* leer ist, wurde diese zur einer Endlos Schleife, die aktiv terminiert wird.

In dieser wird zuerst geprüft, ob die MPI-Queue Minizustände enthält. Ist dies der Fall, so müssen die Minizustände in *Open* einsortiert werden [Zeile 10 bis 25]. Beim Einsortieren der Minizustände wird ein Zähler (*counter*) hochgezählt [Zeile 24], um ein endloses Empfangen von Minizuständen zu unterbinden. Jeder empfangene Minizustand wird darauf geprüft, ob es sich um eine Nachricht, oder einen echten Minizustand handelt. Während ROOT alle drei Arten von Nachrichten nach dem oben beschriebenen Schema abarbeitet, ist *terminate* die einzige Nachricht, die ein Client empfangen kann, durch deren Empfang die Prüfung beendet wird.

Handelt es sich nicht um eine Nachricht, so wird der Minizustand in *Open* eingefügt, der *counter* inkrementiert und, falls das Maximum erreicht worden ist, die Schleife abgebrochen.

Erst nachdem die MPI-Queue überprüft worden ist, wird ein Zustand aus *Open* expandiert. Ist *Open* leer, so wird gewartet, bis entweder Zustände zum expandieren vorhanden sind, oder terminiert werden soll [Zeile 27]. Falls bei einem Client die MPI-Queue und *Open* leer sind, sendet dieser eine *idle* Nachricht an ROOT und wartet dann auf einen Minizustand.

Enthält *Open* Zustände, die expandiert werden sollen, werden diese weiterhin mittels der Funktion 5.2 ermittelt und expandiert. Jedoch wird für die expandierten Zustände nun geprüft, ob diese verschickt werden sollen [Zeile 42 bis 45]. Ist dies der Fall wird der Zustand in eine Transferdatei geschrieben und dann der Minizustand verschickt. Um gleichzeitiges Lesen und Schreiben zu vermeiden, existieren für je zwei Knoten immer zwei Transferdateien. Jeweils eine für das Senden von Knoten i zu Knoten j , mit $i \neq j$ und eine für die umgekehrte Richtung. Des Weiteren erlaubt diese Methode ein dauerhaftes offenhalten der Dateien, sowohl fürs lesen, als auch fürs schreiben. Dies wirkt sich positiv auf die Geschwindigkeit der Expandierung aus.

Zustände zur Expansion ermitteln

Algorithmus 5.2 veranschaulicht wie Zustände aus *Open* extrahiert werden. Handelt es sich um einen Zustand, der auf dem selben Knoten expandiert wurde, so wird dieser lediglich dekomprimiert und wiedergegeben. Handelt es sich allerdings um einen Zustand, der auf einem andern Knoten erzeugt wurde, wird vor der Rückgabe geprüft, ob ein identischer Zustand bereits expandiert wurde. Ist dies der Fall, wird der Zustand verworfen, und mittels Rekursion der nächste aus *Open*gelesen [Zeile 5 und 6].

Algorithmus 5.2 getNext (nächsten aus der MPI Queue oder *Open* ausgeben)

Eingabe: Priorisierte Warteschlange *Open*.

```
 1: mini-state ms ← getMin(Open)
 2: if ms = NULL return NULL
 3: if ms.creator ≠ PID then
 4:    State s ← Read(ms) from File< ms.creator, PID > {Zustand, auf den ms.fpos zeigt,
       aus der Transferdatei von ms.creator zu PID lesen.}
 5:    if checkForDuplicate(s) = true then
 6:       return getNext(Open) {Rekursiver Aufruf von getNext() bis kein Duplikat mehr gefunden wird.}
 7:    else
 8:       return s
 9:    end if
10: else
11:    s ← collapseDecompress(ms) {Zustand dekomprimieren}
12: end if
13: return s
```

5.5 Experimente

Um die gewonnene Zeit bei der Expandierung zu demonstrieren, wird der *SpeedUp* wie folgt definiert. Sei m die Zeit, die eine Expandierung mit einem Knoten benötigt, dann errechnet sich der SpeedUp sp für Experimente mit mehreren Knoten aus $sp = m/t$, wobei t die Zeit darstellt, die das Experiment gedauert hat.

Anhang A beinhaltet eine genaue Beschreibung aller verwendeten Modelle.

5.5.1 Benutzte Hardware

Alle Experimente wurden auf einem ClusterVision Cluster-System durchgeführt. Dieses System besteht aus 224 einzelnen Knoten, die, insgesamt 464 Prozessoren besitzen. Als Betriebssystem wird Suse 10.0 verwendet. Für die Experimente wurden ausschließlich Knoten mit 2 CPUs verwendet, bei denen es sich jeweils um AMD Opteron DP 250 (2,4 GHz) Prozessoren handelt. Dieses System benutzt jeden Prozessor als einen Knoten. Für den Transfer werden externe Festplatten auf einem Fileserver benutzt.

Die Übertragung der Minizustände geschieht über ein InfiniBand-Netzwerk. Die Kommunikation mit dem Fileserver, über eine GBit Ethernet-Verbindung.

Zur Zuteilung der Knoten und Ausführung der Programme wird ein PBS System[†] verwendet. Experimente werden in eine so genannte *PBS-Queue* einsortiert und von dort automatisch gestartet. Die maximale Laufzeit für eine Prüfung mit mehreren Knoten beträgt 48 Stunden, eine Restriktion des Cluster-Systems. Obwohl die PBS-Queue eine größere Anzahl an Knoten zulässt, wurden alle Experimente mit maximal 14 Knoten durchgeführt. Auf eine größere Anzahl an Knoten wurde aufgrund der sehr großen Auslastung des Cluster-Systems, durchgehend 99%, und der Notwendigkeit die Experimente mehrmals laufen zu lassen, verzichtet.

5.5.2 Ergebnisse

Graph 5.5 zeigt den SpeedUp, der bei der Verwendung mehrerer Rechenknoten erziehlt wurde Die Instanz, die untersucht wurde, ist ⟨1 2 3 7 8 4 0 11 12 5 6 10 13 14 9 15⟩. Eine optimale Lösung ist in

[†] http://www.openpbs.org

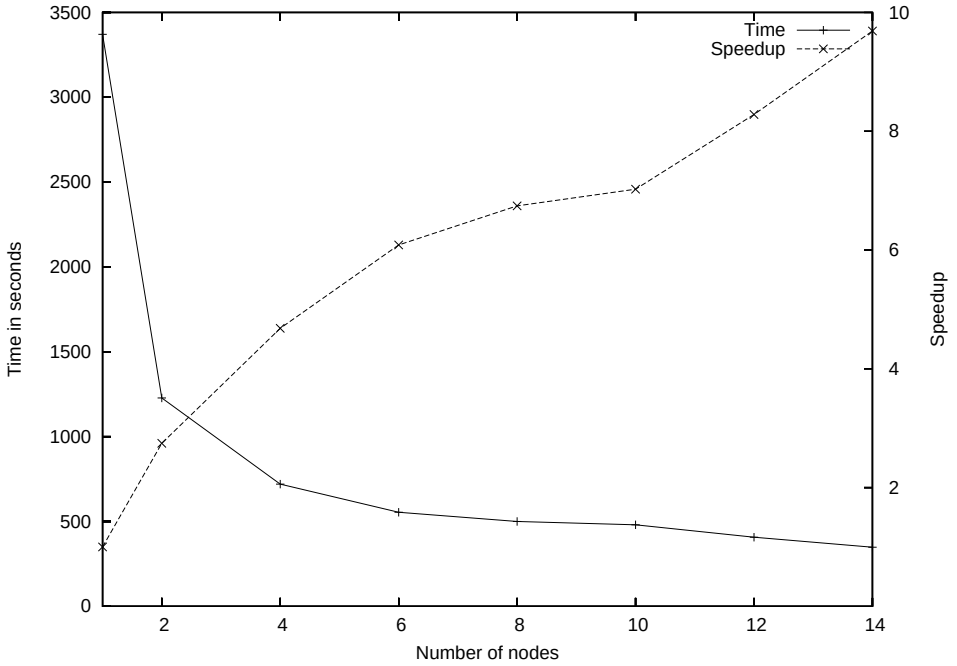

Abbildung 5.5: SpeedUp bei der Lösung einer 15-Puzzle Instanz. Gesucht wurde mit dem BFS Algorithmus. Die Zustände wurden unter Verwendung der Teilhash Verteilung unter den Knoten aufgeteilt. Als Sektion für den Teilhash wurde der MemoryPool benutzt.

Tiefe 140 zu finden. Als Verteilungsfunktion wurde partielles Hashing benutzt. Als Hash-Abschnitt wurde der MemoryPool ausgewählt. Besonders hier zeigt sich, wie wichtig die Auswahl des partiellen Hashing ist. Verwendet man, bei der gleichen Instanz, als Verteilungsfunktion den gesamten Hashwert, so benötigen bereits 2 Rechenknoten die doppelte Zeit, um das Problem zu untersuchen.

Graph 5.6 zeigt die benötigte Zeit, für die Prüfung des Protokolls der Speisenden Philosophen. Als Instanz wurden 200 Philosophen gewählt. Verwendet man einen oder zwei Knoten, ist diese Instanz nicht in der vorgegebenen Zeit lösbar, weshalb die Daten erst bei vier Knoten beginnen. Zur Verteilung der Zustände wurde Ihre Tiefe benutzt. Die maximale Höhe eines Tiefenabschnitts beträgt 100. Auch hier zeigt sich ein nahezu linearer SpeedUp.

Graph 5.7 zeigt den von Holzman in (Holzmann und Bosnacki, 2006) vorgestellten ”Badewannen-Effekt”. Untersucht wurde eine Instanz von 50 Philosophen auf zwei Knoten. Die Ergebnisse stellen einen Mittelwert aus fünf Durchläufen dar. Wir sehen, dass sich sowohl eine zu kleine als auch eine zu große Höhe eines Tiefenabschnitts negativ auf die Geschwindigkeit auswirkt.

Graph 5.8 demonstriert den SpeedUp auf einem Multicore System. Zu Anfang wurde behauptet, dass eine Methode, die für ein Cluster-System entwickelt wurde, sehr einfach auch ein Multicore-System übertragen werden kann. Um dies zu Untermauern, wurde ein Versuch auf einem Multicore-System mit 4 AMD Opteron MP 852 (2.6 GHz) Prozessoren durchgeführt. Das System hat 16 Gigabytes Hauptspeicher und zwei Maxtor 160GB SATA (gespiegelt/ RAID 1) Festplatten, die für die Transferdateien benutzt wurden. Untersucht wurde das Bakery Protokoll mit 3 Kunden. Wir sehen deutlich, dass ein nahezu linearer SpeedUp vorhanden ist.

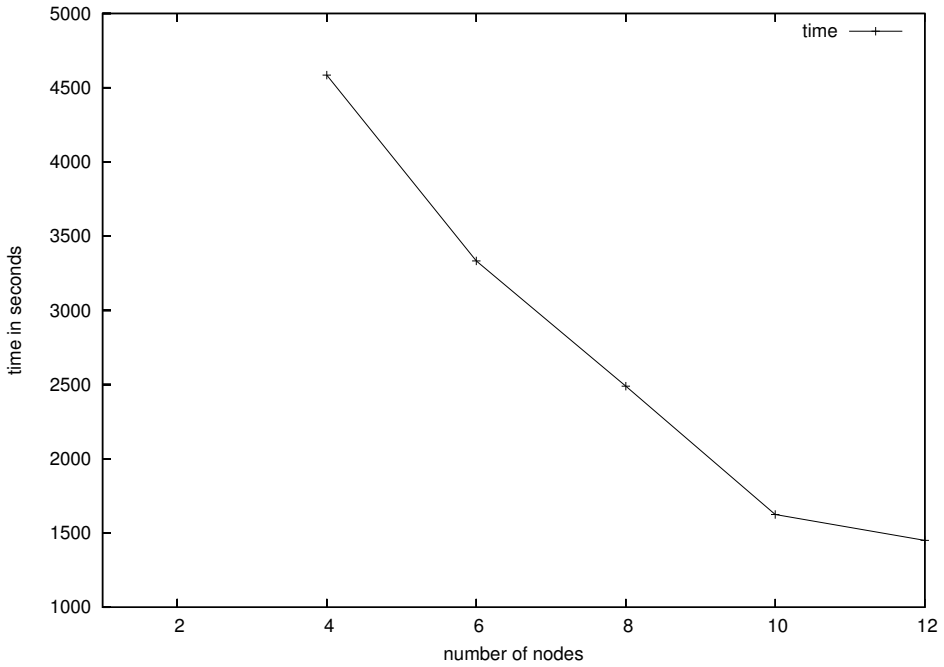

Abbildung 5.6: SpeedUp mit mehreren Knoten, um einen Deadlock im speisende Philosophen Problem, unter Verwendung von DFS, zu finden.

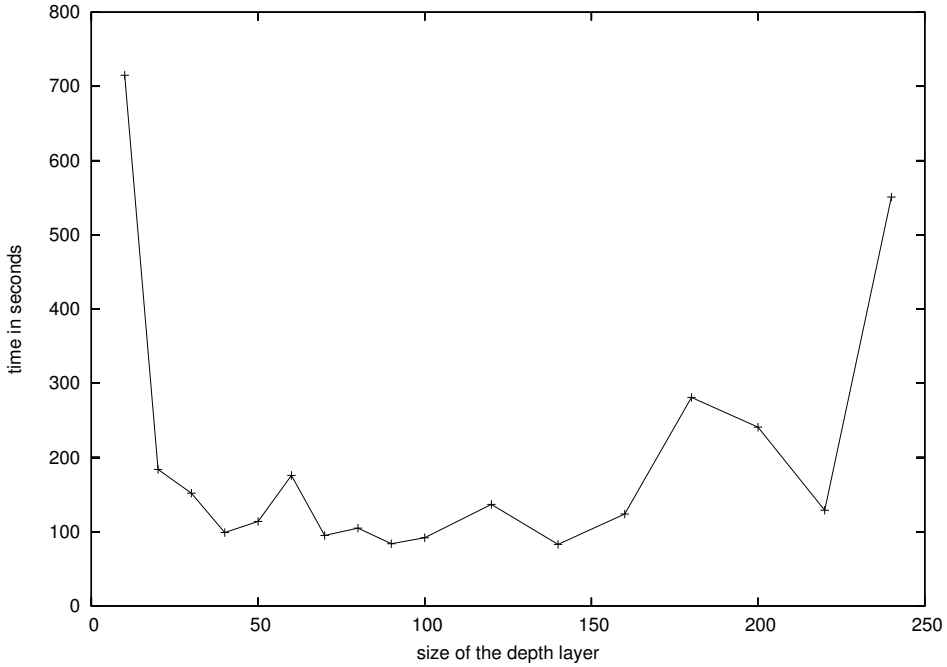

Abbildung 5.7: Badewannen Effekt bei Verteilung entsprechend der Tiefe

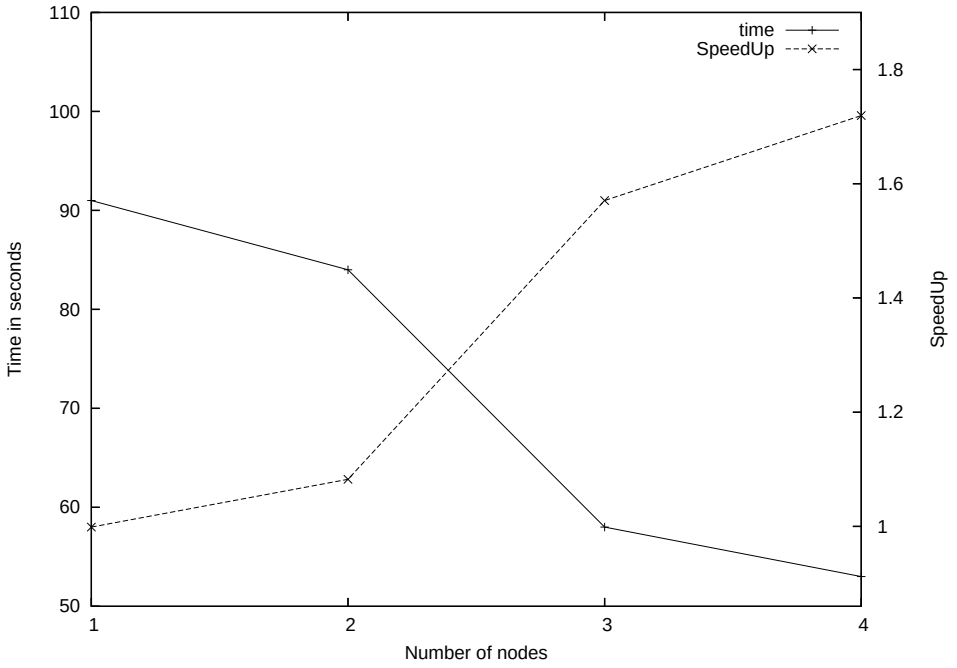

Abbildung 5.8: Prüfung des Bakery Modells auf einem Multicore-System

5.6 Resümee

Der in dieser Arbeit parallelisierte Software Modellprüfer ist in der Lage, komplexe Programme zu untersuchen. Eine Parallelisierung beschleunigt die Prüfung über eine Verteilung der Zustände auf mehrere Knoten.

Eine ähnliche Art der Parallelisierung wurde in Blom u. a. (2007) vorgestellt. Es werden Zustände, ähnlich der hier verwendeten ”Collapse-Kompression” aufgeteilt gespeichert und übertragen. Jedoch findet keine Zwei-Wege-Kommunikation statt.

Programme, die nahezu den gesamten Hauptspeicher eines Rechners einnehmen, könnten, dank der Zwei-Wege-Kommunikation, geprüft werden. Die Zwei-Wege-Kommunikation benutzt sowohl MPI als auch NFS, um Zustände zu transferieren.

Aufgrund der variablen Implementierung der Verteilungsfunktion ist eine dynamische Verteilung der Zustände in Zukunft möglich. Dies würde eine bessere Auslastung jedes einzelnen Knoten bedeuten, aber auch weniger Kommunikation zwischen diesen Knoten erzwingen.

Des Weiteren wäre eine Prioritätsfunktion möglich, bei der Zustände auf einem Knoten präferiert werden, die auf diesem Knoten expandiert wurden.

Kapitel 6

Fazit

In dieser Arbeit wurde der Software Modellprüfer StEAM erweitert. StEAM ist in der Lage, nebenläufige C++ Programme direkt auf der Ebene vom Maschinencode zu prüfen. Benutzt wird dafür die ICVM, eine virtuelle Maschine, die *ferngesteuert* wird. Prüft StEAM ein nebenläufiges Programm, so expandiert er den Zustandsraum des Programms. Die dabei entstehenden Zustände werden entweder im Hauptspeicher (in Cache-Strukturen) oder auf der Festplatte gespeichert. Dies erlaubt eine Untersuchung großer Modelle, die nahezu den gesamten Hauptspeicher eines Rechners einnehmen.

In dieser Arbeit wurden virtuelle Speicheradressen eingeführt, die es ermöglichen ein Programm anzuhalten und auf einen anderen Computer zu übertragen. Die virtuellen Adressen werden ebenfalls benutzt, um Fehler zu finden, die durch unzulässige Speicherzugriffe entstehen.

Des Weiteren wurde StEAM durch ein Interface von der virtuellen Maschine getrennt. Soll der Modellprüfer unterschiedlichen Maschinencode akzeptieren, z.B. im ”Embeded System” Bereich, so kann die virtuelle Maschine nun ausgetauscht werden. Für den Austausch wird nur detailliertes Wissen über die virtuelle Maschine benötigt, nicht über StEAM. Während mit der Entwicklung von StEAM gezeigt wurde, dass eine virtuelle Maschine zu einem Modellprüfer erweitert werden kann, so wurde in dieser Arbeit gezeigt, dass ein Software Modellprüfer mehrere virtuelle Maschinen benutzen kann.

Der Hauptaugenmerk dieser Arbeit lag in der Parallelisierung des Modellprüfers. Diese benutzt eine Zwei-Wege-Kommunikation um die Größe der Programme, die geprüft werden, nicht einzuschränken. Der Ansatz wurde in (Edelkamp u. a., 2007) erläutert und wird bei der ”Parallel and Distributed Methods in verifiCation” Konferenz vorgestellt. Die Tatsache, dass das Papier akzeptiert wurde stuft diese Arbeit als neu ein.

Minizustände, die eine geringe konstante Größe aufweisen, werden über das Netzwerk geschickt, während der Zustand auf einem gemeinsamen Medium abgelegt wird. So belegen Zustände, die von einem Knoten nicht benötigt werden, keinerlei Hauptspeicher. Die verzögerte Duplikatseliminierung ermöglicht es, die Prüfung weiterhin zu beschleunigen, da die nicht expandierten Zustände, nicht vom gemeinsamen Medium gelesen werden müssen. Der in dieser Arbeit vorgestellte Algorithmus ermöglicht einen nahezu linearen SpeedUp, sowohl auf Cluster-, als auch auf Multicore-Systemen.

Eine Erweiterung der virtuellen Speicheradressen würde es möglich machen, weitere Speicherzugriffsfehler zu finden. Über einen Bittvektor wäre ein Lesen von Adressen, an die keine Daten geschrieben wurden, als Fehler zu erkennen. Nach einer Modifizierung des virtuellen Stacks wäre StEAM in der Lage zu erkennen, ob ein ”Buffer Overflow” in einem Programm möglich ist.

Die Parallelisierung kann erweitert werden um eine dynamische Verteilung der Zustände zu unterstützen. So würden einzelne Knoten gleichmäßig ausgelastet und es könnten Knoten zur Laufzeit entfernt oder hinzugefügt werden. Weiterhin könnte die Speicherung der Zustände, die transferiert

werden sollen, optimiert werden. Würde die Collapse-Kompression weiter in die Parallelisierung eingebunden, so könnten einzelne Cache-Strukturen Inhalte untereinander austauschen und es müssten nicht immer Zustände transferiert werden.

Die Transferdateien werden momentan auf einem NFS Server abgelegt. In der Zukunft wäre es möglich andere, schnellere Medien zu finden. Auch eine Auslagerung in eine Datenbank wäre denkbar.

Anhang A

Die verwendeten Modelle

A.1 Speisenden Philosophen

Das Problem der Speisenden Philosophen (Dijkstra, 2002) stammt von Edsger Dijkstra aus dem Jahr 1971, der ein Problem formulierte, bei dem mehrere Computer exklusiven Zugriff auf mehrere Festplatten erhalten sollten. Kurze Zeit darauf formulierte Tony Hoare daraus das Problem der speisenden Philosophen.

Bei dem Problem nimmt man an, dass eine Anzahl von Philosophen an einem Tisch sitzen. Vor jedem Philosophen liegt ein Teller Spagetti. Diese Speise kann nur mir zwei Gabeln von einem Philosophen gegessen werden. Jedoch liegen nur so viele Gabeln auf dem Tisch wie anwesende Philosophen samt Teller. Demnach liegt zwischen je zwei Philosophen eine Gabel, die diese sich teilen müssen.

Implementiert wurde das Protokoll mit Hilfe von Threads und einem Bool'schen Array, welches die Gabeln darstellt. „Greift" ein Philosoph eine Gabel wird in dem Array eine Stelle blockiert, so das kein anderer darauf zugreifen kann. Dies geschieht über das StEAM Tag `VLOCK`. Legt ein Philosoph seine Gabel wieder an den ursprünglichen Platz, so wird die Adresse über `VUNLOCK` wieder freigegeben. Abbilbung A.2 zeigt die Ininitialisierung der Philosophen im Modell. Abblibung A.1 stellt die *run* Methode eines Philosphen dar. Die run Methode wird gestartet nachdem ein Thread instanziiert wurde.

Funktion A.2 und A.1 zeigen die Implemetierung bei der ein Deadlock gefunden werden muss.

```
void Philosopher::run() {
  int i;
  while(1) {
                    // BEGINATOMIC;
    VLOCK(leftfork);
    VLOCK(rightfork);
    VUNLOCK(rightfork);
    VUNLOCK(leftfork);
                    // ENDATOMIC;
  }
}
```

Abbildung A.1: C++ Quellcode eines Philosophen (run-Mathode der Klasse)

```
Philosopher ** p;
short forks[255];

void initThreads (int n) {
  p=(Philosopher **) malloc(n*sizeof(Philosopher *));
  for(int i=0;i<n;i++) {
    p[i]=new Philosopher(&forks[i], &forks[(i+1) % n]);
    p[i]->start();
  }
}

int main(int argc, char ** argv) {
  int n;
  BEGINATOMIC;
  n=atoi(argv[1]);
  initThreads(n);
  ENDATOMIC;
}
```

Abbildung A.2: C++ Quellcode des speisende Philosophen Protokolls(main-Routine)

Dieser entsteht, wenn jeweils ein Philosoph die erste Gabel in der Hand hält und auf die zweite wartet. Um diesen Deadlock zu umgehen muss man einen Philosophen zwingen, beide Gabeln gleichzeitig zu greifen. Dies kann durch einen *atomaren Block* erreicht werden, der von StEAM, ohne Unterbrechung, ausgeführt wird. Hierzu nutzt man die, auskomentierten StEAM Tags `BEGINATOMIC` und `ENDATOMIC`, die um den Bereich in der **while** true Schleife gesetzt werden.

Tabelle A.1: Erläuterung des Fehlerpfades

Schritt	Source	Thread	Erläuterung
1	`ENDATOMIC;`	1	Die Philosophen wurden initialisiert.
2	`while(1) {`	4	beginnt seinen Lebenszyklus, essen und denken
3	`VLOCK(leftfork);`	4	**hält** linke Gabel
4	`while(1) {`	3	beginnt seinen Lebenszyklus, essen und denken
5	`VLOCK(leftfork);`	3	**hält** linke Gabel
6	`VLOCK(rightfork);`	3	**wartet** auf die rechte Gabel
7	`}`	1	terminiert (wird pausiert)
8	`while(1) {`	5	beginnt seinen Lebenszyklus, essen und denken
9	`while(1) {`	2	beginnt seinen Lebenszyklus, essen und denken
10	`VLOCK(leftfork);`	5	**hält** linke Gabel
11	`VLOCK(leftfork);`	2	**hält** linke Gabel
12	`VLOCK(rightfork);`	5	**wartet** auf die rechte Gabel
13	`}`	1	pausiert
14	`VLOCK(rightfork);`	4	**wartet** auf die rechte Gabel
15	`VLOCK(rightfork);`	2	**wartet** auf die rechte Gabel

Tabelle A.1 interpretiert Fehlerpfad auf Quelltext Ebene. Aufgrund der Threadnummer und der zugehörigen Zeile sieht man leicht das alle noch aktiven Threads auf eine Resource “warten”. Dieser Zustand entspricht der Definition eines Deadlock Zustandes.

A.2 N-Puzzle

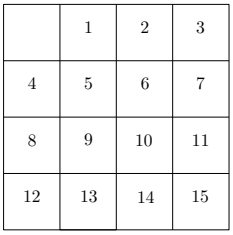

Abbildung A.3: Beispiel eines 16-Puzzels

Beim N-Puzzle handelt es sich um ein Schiebe Puzzle. Der Name wird durch die Anzahl der Plätchen im Puzzle festgelegt. Abbildung A.2 zeigt ein 15-Puzzle. Eine Instanz eines N-Puzzles besteht aus einer vorgegebenen Anordnung aller *Plättchen*. Ein Plättchen ist ein verschiebbares Element des Puzzles, und trägt eine Nummer. Natürlich kann ein Plättchen nur verschoben werden, falls sich neben diesem ein freies Feld befindet. Verschiebt man ein Plättchen, so entsteht eine neue Instanz. Die gesuchte Instanz ist ⟨0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ⟩ und entspricht der, die in der Abbildung A.2 dargestellt worden ist. Hier sind alle Plättchen in steigender Reihenfolge angeordnet.

Obwohl es beim N-Puzzle keinerlei Nichtdeterminismus gibt, wurde das Modell implementiert, um das StEAM Tag `RANGE` zu demonstrieren. An diesem Modell lassen sich sehr gut Optimierungen der nicht gerichteten Suche untersuchen. Abbildung A.4 zeigt die *main* Methode in der ein N-Puzzle instaziert wird. Für die Plätchen engl. *tiles* wird ein **short** Array benutzt. Um ein Plätchen zu bewegen wird `RANGE` in einer **while** Schleife aufgerufen, bis die `VASSERT` Zusicherung verletzt wurde. Bricht die Schleife, aufgrund der Verletzung der Zusicherung, ab, entspricht die Instanz der Zahlenfolge
$\langle 0, 1, ..., ((n^2) - 1), (n^2) \rangle$
.

```
void ShiftUp() {
  BEGINATOMIC;
  if(blank>=n) {
    tmp=tiles[blank-n];
    tiles[blank-n]=0;
    tiles[blank]=tmp;
    blank-=n;
  }
  ENDATOMIC;
}
```

Abbildung A.4: Funktion um ein Plätchen im nPuzzle zu verschiebenn

Funktion A.5 zeigt eine Schiebemethode, von denen es insgesamt vier gibt. Je nach Wert der Variable *i* wird ein Plätchen aus der Umgebung des freien Feldes an dessen Position verschoben.

```
void main(int argc, char **argv) {

  int i=-1;

  BEGINATOMIC;
  tiles=(short *) malloc(n*n*sizeof(short));
  for(i=0;i<n*n;i++){
    tiles[i]=atoi(argv[i+2]);
    if(tiles[i]==0) blank=i;
  }
  ENDATOMIC;

  while(1){
    for(i=0;i<n*n && tiles[i]==i;i++);
    VASSERT(i<n*n);
    RANGE(i,0,3);
    switch(i) {
    case 0:
      ShiftUp();
      break;
    case 1:
      ShiftDown();
      break;
    case 2:
      ShiftLeft();
      break;
    case 3:
      ShiftRight();
      break;
   }
  }
}
```

Abbildung A.5: C++ Quelltext des NPuzzle Modells (main-Methode)

A.3 Bakery Protokoll

Lamport's Bakery Algorithmus (Lamport, 1974) gehört zu der Gruppe der „mutual exclusion" Algorithmen. Diese Algorithmen werden verwendet, um Prozessen exklusiven Zugang zu einer Ressource zu gewähren. Exklusiv bedeutet hierbei, dass niemals zwei oder sogar mehr Prozesse gleichzeitig auf die Ressource zugreifen. Greifft ein Prozess auf die Ressource zu, so befindet er sich in der *kritischen Zone*.

Dr. Lamport benutzt die Idee einer Bäckerei in der mehrere Kunden in der Schlange stehen. Diese

Kunden benötigen exklusiven Zugriff auf den Bäcker, um ihre Ware zu bestellen. Am Eingang der Bäckerei stehen Automaten, die Nummern an jeden Kunden vergeben, der die Bäckerei betritt. Zugewiesen wird immer eine Nummer, die um eins größer ist als die größte bis dato vergebene Nummer. Da aber mehrere Automaten vorhanden sind, kann aber eine Nummer mehrmalls vergeben werden. Prozesse, die Zugriff auf die Ressource beantragen, bekommen ebenfalls eine Nummer und warten dann auf die Erlaubnis, die kritische Zone betreten zu dürfen. Bei der Ausführung des Algorithmus kann es ebenfalls zu einer Situation kommen in der zwei Prozesse dieselbe Nummer zugewiesen bekommen. Deshalb muss, falls ein Prozess auf Zugriff wartet nicht nur die vergebene Nummer, sondern auch die ID des Prozesses überprüft werden. Einlass wird dem Prozess gewährt, der die kleinere ID vorweist. Funktion A.6 zeigt wie die einzelnen Threads initialisiert werden. Implementiert wurde dieses Modell mit Hilfe von `p_threads`.

```
int main(int argc, char ** argv) {
        BEGINATOMIC;
  N=atoi(argv[1]);
        Entering = (int*) malloc(N * sizeof(int));
        Number = (int* ) malloc(N * sizeof(int));
        for (int i = 0;i < N; i++) Entering[i]=1;
        for (int i = 0;i < N; i++) Number[i]=0;
        //initialize threads
        pthread_t threads[N];
        for (int i = 0;i < N; i++){
                pthread_create(&threads[i],NULL,thread,(void *) i);
        }
        ENDATOMIC;
}
```

Abbildung A.6: Initialisierung des Bakery Algorithmus als C++ Quelltext

Funktion A.8 demonstriert wie eine Zahl zugewiesen wird.

Um diesen Algorithmus zu prüfen, wurde eine weitere Variable eingeführt, die die Anzahl der Prozesse in der kritischen Zone angibt. Mit Hilfe des StEAM Tags `VASSERT` kann nun die Größe dieser Variable geprüft werden. Erreicht diese einen Wert größer eins, so wird ein Fehlerzustand gemeldet. Weiterhin wurde die While Schleife in Zeile 20 begrenzt um einen endlichen Zustandsraum zu erhalten .

Obwohl es eine Reihe von Fehlern gibt, die in diesen Algorithmus eingebaut werden können, wurde er als fehlerfreies Modell implementiert, um eine Expandierung des gesamten Zustandsraums zu ermöglichen.

```
void * thread(void * ID){
        int myID = (int) ID;
        int loops = 0; // just to make it finite
        while (loops < 5) {
                lock(myID);
                //check if i am alone (StEAM code)
                VASSERT(inCS == 0);
                inCS=1;
                //do whatever on my own
                inCS=0;
                unlock(myID);
                loops++;
        }
}
```

Abbildung A.7: Kunde einer Bäkerei als Thread in C++ Quelltext

```
void lock(int i) {
  Entering[i] = 1; // getting a number
  int max = 0;
  for (int j = 0; j < N; j++) {
    if (max < Number[j]) max = Number[j];
  }
  Number[i]=max+1; //got the biggest possible number
  Entering[i] = 0; // got a number

  for(int j = 0; j < N; j++) {
    while (Entering[j]) {
      //wait untill all clients got a number
    }
    while ((Number[j] !=0) && (Number[j] < Number[i] || j < i )) {
      //wait until no client, who has a number, has a lover number
    }
  }
}
```

Abbildung A.8: Zuweisen einer Zahl und warten auf Eintritt im Bakery Protokoll

Literaturverzeichnis

[Blom u. a. 2007] Blom, S. ; Lisser, B. ; Pol, J. van de ; Weber, M.: A Database Approach to Distributed State Space Generation. In: *Parallel and Distributed Methods in verifiCation (PDMC)*, 2007. – To appear

[Burns u. a. 1994] Burns, Greg ; Daoud, Raja ; Vaigl, James: LAM: An Open Cluster Environment for MPI. In: *Proceedings of Supercomputing Symposium*, 1994, S. 379–386

[Clarke u. a. 1999] Clarke, E. M. ; Grumberg, O. ; Peled, D. A.: *Model checking*. MIT Press, 1999. – ISBN 0-262-03270-8

[Cormen u. a. 1990] Cormen, T. H. ; Leiserson, C. E. ; Rivest, R. L.: *Introduction to Algorithms*. The MIT Press, 1990

[Cousot und Cousot 1977] Cousot, P. ; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*, ACM Press, 1977, S. 238–252

[Dijkstra 2002] Dijkstra, Edsger W.: Hierarchical ordering of sequential processes. (2002), S. 198–227. ISBN 0-387-95401-5

[Edelkamp u. a. 2007] Edelkamp, S. ; Jabbar, S. ; Sulewski, D.: Distributed Verification of Multithreaded C++ Programs. In: *Parallel and Distributed Methods in verifiCation (PDMC)*, 2007. – To appear

[Edelkamp und Lluch-Lafuente 2001] Edelkamp, S. ; Lluch-Lafuente, A.: HSF-Spin User Manual, 2001

[Edelkamp u. a. 2006] Edelkamp, Stefan ; Jabbar, Shahid ; Midzic, Dino ; Rikowski, Daniel ; Sulewski, Damian: External Program Model Checking. In: *KI'06 (German Conference on AI) Workshop on New Results in Planning, Scheduling and Design (PuK'06)*, 2006

[Gelperin und Hetzel 1988] Gelperin, D. ; Hetzel, B.: The growth of software testing. In: *Commun. ACM* 31 (1988), Nr. 6, S. 687–695. – ISSN 0001-0782

[Godefroid 1997] Godefroid, P.: Model Checking for Programming Languages using Verisoft. In: *Symposium on Principles of Programming Languages*, 1997, S. 174–186

[Godefroid 2005] Godefroid, P.: Software Model Checking: The VeriSoft Approach. In: *Formal Methods in System Design* 26 (2005), Nr. 2, S. 77–101

[Groce und Visser 2002] Groce, A. ; Visser, W.: Heuristic Model Checking for Java Programs. In: *SPIN Workshop on Model Checking of Software* (2002), S. 242–245

[Hart u. a. 1968] Hart, P. E. ; Nilsson, N. J. ; Raphael, B.: A Formal Basis for Heuristic Determination of Minimum Path Cost. In: *IEEE Transactions on Systems Science and Cybernetics* 4 (1968), S. 100–107

[Hetzel 1990] Hetzel, William C.: *The Complete Guide to Software Testing*. New York, NY, USA : John Wiley & Sons, Inc., 1990. – ISBN 0894351109

[Holzmann 2004] Holzmann, G.: *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004

[Holzmann und Bosnacki 2006] Holzmann, G. ; Bosnacki, D.: The Design of a multi-core extension of the Spin Model Checker. In: *Presented at Formal Methods in Computer Aided Design (FMCAD)'06*, 2006

[Holzmann 1997] Holzmann, G. J.: State Compression in SPIN. In: *Third Spin Workshop*. Twente University, The Netherlands, 1997

[Holzmann 1998] Holzmann, G. J.: An Analysis of Bitstate Hashing. In: *Formal Methods in System Design* 13 (1998), Nr. 3, S. 287–305

[Jard und Jéron 1992] Jard, C. ; Jéron, T.: Bounded-memory Algorithms for Verification On-the-fly. In: *CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification*. London, UK : Springer-Verlag, 1992, S. 192–202. – ISBN 3-540-55179-4

[Korf 1985] Korf, R. E.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. In: *Artificial Intelligence* 27 (1985), Nr. 1, S. 97–109

[Korf 2003] Korf, R. E.: Breadth-First Frontier Search with Delayed Duplicate Detection. In: *Model Checking and Artificial Intelligence (MoChArt)*, 2003, S. 87–92

[Korf u. a. 2001] Korf, R. E. ; Reid, M. ; Edelkamp, S.: Time Complexity of Iterative-Deepening-A*. In: *Artificial Intelligence* 129 (2001), Nr. 1–2, S. 199–218

[Kristensen und Mailund 2003] Kristensen, L. M. ; Mailund, T.: Path Finding with the Sweep-Line Method using External Storage. In: *ICFEM*, 2003, S. 319–337

[Lamport 1974] Lamport, L.: A new solution of Dijkstra's concurrent programming problem. In: *Communications of the ACM* 17 (1974), Nr. 8, S. 453–455. – ISSN 0001-0782

[Lerda und Sisto 1999] Lerda, F. ; Sisto, R.: Distributed-memory Model Checking with SPIN. In: *Proc. of the 5th International SPIN Workshop* Bd. 1680, Springer-Verlag, 1999. – URL `citeseer.ist.psu.edu/lerda99distributedmemory.html`

[Lluch-Lafuente 2003] Lluch-Lafuente, A.: Symmetry Reduction and Heuristic Search for Error Detection in Model Checking. In: *Model Checking and Artificial Intelligence (MoChArt)*, 2003, S. 77–86

[Lluch-Lafuente u. a. 2002] Lluch-Lafuente, A. ; Leue, S. ; Edelkamp, S.: Partial Order Reduction in Directed Model Checking. In: *Workshop on Model Checking Software (SPIN)*, Springer, 2002 (Lecture Notes in Computer Science), S. 112–127

[Mehler 2006] Mehler, T.: *Challenges and Applications of Assembly-Level Software Model Checking*, University of Dortmund, Dissertation, 2006

[Mehler und Edelkamp 2006] Mehler, T. ; Edelkamp, S.: Dynamic Incremental Hashing in Program Model Checking. In: *Electronic Notes in Theoretical Computer Science* 149 (2006), Nr. 2, S. 51–69

[Pearl 1985] Pearl, J.: *Heuristics*. Addison-Wesley, 1985

[Robby u. a. 2003] Robby ; Dwyer, M. B. ; Hatcliff, J.: Bogor: An Extensible and Highly-Modular Software Model Checking Framework. In: *European Software Engineering Conference (ESEC)*, 2003, S. 267–276

[Sanders u. a. 2002] Sanders, P. ; Meyer, U. ; Sibeyn, J. F.: *Algorithms for Memory Hierarchies*. Springer, 2002

[Squyres und Lumsdaine 2003] Squyres, J. M. ; Lumsdaine, A.: A Component Architecture for LAM/MPI. In: *Proceedings, 10th European PVM/MPI Users' Group Meeting*. Venice, Italy : Springer-Verlag, September / October 2003 (Lecture Notes in Computer Science 2840), S. 379–387

[Stallman 2003] Stallman, R.: *Using the GNU Compiler Collection*. 2003

[Steffen 1993] Steffen, B.: Generating data flow analysis algorithms from modal specifications. In: *Science of Computer Programming* 21 (1993), Oct., Nr. 2, S. 115 – 139

[Stern und Dill 1998] Stern, U. ; Dill, D.: Using Magnetic Disk instead of Main Memory in the Murphi Verifier. In: *Computer Aided Verification*, 1998, S. 172–183

[Stroustrup 2000] Stroustrup, B.: *The C++ Programming Language*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000. – ISBN 0201700735

[Tanenbaum 1992] Tanenbaum, A. S.: *Modern Operating Systems*. Prentice-Hall, 1992

[Tarjan 1972] Tarjan, R.: Depth-first search and linear graph algorithms. In: *SIAM Journal of Computing* 1 (1972), Nr. 2, S. 146–160

[Visser u. a. 2000a] Visser, W. ; Havelund, K. ; Brat, G. ; Park, S.: *Java PathFinder - Second Generation of a Java Model Checker*. 2000. – URL `citeseer.ist.psu.edu/visser00java.html`

[Visser u. a. 2000b] Visser, W. ; Havelund, K. ; Brat, G. ; Park., S.: Model Checking Programs. In: *International Conference on Automated Software Engineering (ICSE)*, 2000, S. 3–12