Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 106

Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache

Dennis Przytarski

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. / Harvard Univ. Erhard Plödereder
Betreuer/in:	Dipl.-Inf. Timm Felden
Beginn am:	16. Dezember 2013
Beendet am:	17. Juni 2014
CR-Nummer:	D.3.3, D.4.8, E.2, E.5

Kurzfassung

An der Universität Stuttgart wurde die Sprache Serialization Killer Language (SKilL) mit dem Ziel entworfen, sehr große Datenmengen sprach- und plattformunabhängig serialisieren zu können.

Derzeit existiert ein Scala-Binding für die Serialisierungssprache als Referenzimplementierung. Um die Performance des Scala-Bindings vergleichen zu können, wird ein SKilL Codegenerator für die Programmiersprache Ada entwickelt.

In der vorliegenden Arbeit werden das Softwaredesign des Ada-Bindings sowie die entdeckten Bottlenecks, die nach der Auflösung zur Performanceverbesserung des Ada-Bindings führten, vorgestellt. Anschließend wird die Performance mit der Referenzimplementierung und der in der Programmiersprache Ada vorhandenen sprachabhängigen Serialisierung an verschiedenen Nutzungsszenarien verglichen. Darüber hinaus wird ein Verbesserungsvorschlag zum Serialisierungsformat erläutert.

Inhaltsverzeichnis

1 Einleitung

An der Universität Stuttgart wurde die Sprache SKilL mit dem Ziel entworfen, sehr große Datenmengen sprach- und plattformunabhängig serialisieren zu können [Fel13]. Derzeit existiert nur ein Binding für die Serialisierungssprache SKilL in der Programmiersprache Scala [Ode11]. Scala ist eine funktionale, objektorientierte Programmiersprache und läuft auf der Java Virtual Machine (JVM), die einen Garbage Collector (GC) und Just-In-Time (JIT)-Compiler nutzt [Ora14]. Um die Performance des Scala-Bindings vergleichen zu können, wird in dieser Arbeit ein zweites Binding entwickelt. Dafür wurde die Programmiersprache Ada [TDB+12] gewählt, weil sie eine systemnahe Programmiersprache mit hohem Abstraktionsniveau ist, aber keinen GC und JIT-Compiler nutzt. Außerdem wird die Programmiersprache Ada an der Universität Stuttgart für Softwareanalysen mit großen anfallenden Datenmengen verwendet [RVP06].

1.1 Aufgabenstellung

Das erste Ziel dieser Arbeit ist zunächst ein Binding für die Programmiersprache Ada zu entwickeln. Es existiert bereits ein Front-End für die Serialisierungsprache SKilL, das derzeit nur die Programmiersprache Scala unterstützt. Dieses Front-End stellt eine geeignete Zwischendarstellung der SKilL-Spezifikation bereit und soll nun um die Programmiersprache Ada erweitert werden. Anschließend soll als zweites Ziel dieser Arbeit das Ada-Binding auf seine Performance evaluiert werden. Optional kann das Ada-Binding mit der in der Programmiersprache Ada vorhandenen sprachabhängigen Serialisierung verglichen werden. Optional können mögliche Verbesserungen des Serialisierungsformats, die sich bei der Entwicklung des Ada-Bindings ergeben, vorgeschlagen werden.

1.2 Gliederung

In Kapitel 2 werden die notwendigen Grundlagen für diese Arbeit vermittelt.

In Kapitel 3 wird das Scala-Binding als verwandte Arbeit vorgestellt. Derzeit ist es das einzige verfügbare Binding, das fast die komplette Serialisierungssprache SKilL abdeckt.

In Kapitel 4 wird die Testumgebung für alle Tests in den nachfolgenden Kapiteln vorgestellt.

In Kapitel 5 werden das Softwaredesign und die entdeckten Bottlenecks, die nach der Auflösung zur Performanceverbesserung des Ada-Bindings führten, beschrieben. Anschließend wird die Nutzung des generierten Bindings anhand eines allgemeinen Codebeispiels erläutert.

In Kapitel 6 und 7 wird das in dieser Arbeit entwickelte Ada-Binding mit dem Scala-Binding und der in der Programmiersprache Ada vorhandenen sprachabhängigen Serialisierung verglichen und evaluiert.

In Kapitel 8 wird ein Verbesserungsvorschlag für das Serialisierungsformat erläutert.

Abschließend werden in Kapitel 9 die Ergebnisse der Arbeit zusammengefasst und durch ein Ausblick abgerundet.

2 Grundlagen

Dieses Kapitel fasst die benötigten Grundlagen aus dem Technischen Bericht 2013/06 [Fel13] zusammen. Der Technische Bericht ist in der englischen Sprache verfasst und bietet zusätzliche, aber für diese Arbeit uninteressante Informationen wie z. B. die formale Semantik der Serialisierungssprache und die Restriktionen.

2.1 SKilL Kernsprache und Floats

Ein Binding für die Serialisierungssprache SKilL muss mindestens die Kernsprache [Fel13, § C] abdecken. Daher werden in dieser Sektion die Eigenschaften der SKilL Kernsprache und zusätzlich die Floats erläutert.

2.1.1 Grundtypen

Unter dem Begriff Grundtypen werden die Integer-Typen, `string`, `bool`, `annotation` und `Floats` zusammengefasst und nachfolgend basierend auf dem Technischen Bericht 2013/06 [Fel13, § 4.1] vorgestellt.

Integer-Typen: Es gibt zwei Arten von Integer-Typen, nämlich mit fixer und variabler Länge. Die Typen i8, i16, i32 und i64 benötigen jeweils ein, zwei, vier oder acht Byte(s). Der Typ v64 hat die Besonderheit, dass die benötigten Bytes vom Wert abhängen. Dadurch wird sichergestellt, dass kleine Werte ($[0, 128[\in \mathbb{N}_0$) in einem einzelnen Byte abgespeichert werden können. Der Nachteil ist jedoch, dass große ($\geq 2^{55}$) und negative Werte ein zusätzliches, neuntes Byte benötigen.

`string`: Ein String ist eine Zeichenkette mit variabler Länge bestehend aus UTF-8 kodierten Unicode-Zeichen. Die Strings sollten keine Nullzeichen beinhalten, weil das Probleme mit verschiedenen Programmiersprachen verursachen kann.

`bool`: Die Wahrheitswerte können die Werte wahr oder falsch speichern. Es ist zu beachten, dass die Serialisierungssprache hierfür nicht den Integer-Typ verwendet, sondern einen eigenen Typ anbietet.

`annotation`: Die Annotationen sind als Haupterweiterungspunkt gedacht. Sie können auf beliebige benutzerdefinierte Typen zeigen. Um dies zu erreichen, wird zusätzlich zum Zeiger noch sein Typ gespeichert.

`Floats`: Ein Float ist eine binäre Gleitkommazahlen mit 32 Bit oder 64 Bit nach der Norm IEEE 754 [IEE08].

2.1.2 Zusammengesetzte Typen

Zusätzlich zu den Grundtypen bietet die Serialisierungssprache auch zusammengesetzte Typen an. Es existieren hierfür Arrays mit konstanter und variabler Länge, sowie Lists, Maps und Sets. Zusammengesetzte Typen können nicht weitere zusammengesetzte Typen beinhalten, d. h. als Basistyp sind nur die Grundtypen und die benutzerdefinierten Typen erlaubt. Ein Set darf die selbe Instanz nicht zweimal beinhalten. Eine Map hat mindestens zwei Basistypen als Argument. [Fel13, § 4.2]

2.1.3 Benutzerdefinierte Typen mit Untertypen

Ein benutzerdefinierter Typ kann als eine Ansammlung von Kombinationen aus einem Typ und seinem dazugehörigen Feldnamen angesehen werden. Die Abbildung 2.1 zeigt einen benutzerdefinierten Typ namens *Node* mit den Feldern *name* und *edges*. Das Feld *name* ist vom Typ string und das Feld *edges* ist ein Set, das sich selbst als Basistyp hält. [Fel13, § 4.3]

Listing 2.1: Benutzerdefinierter Typ

```
Node {
  string name;
  Set<Node> edges;
}
```

Des Weiteren bietet die Serialisierungssprache eine Einfachvererbung für die benutzerdefinierten Typen an. Somit verhält es sich wie bei der objektorientierten Programmierung mit einer Einfachvererbung. Die Beziehungen zwischen den benutzerdefinierten Typen müssen azyklisch sein. Ein benutzerdefinierter Typ kann somit seine Struktur vererben (Supertyp) oder eine Struktur ererben (Untertyp) und diese erweitern. [Fel13, § 3.3]

2.1.4 `const` und `auto` Felder

Die `const` Felder, auch Konstanten genannt, können nur die Integer-Typen erweitern. Sie können z. B. zur Versionierung genutzt werden. Bei der Deserialisierung werden die Werte der Konstanten überprüft. Bei einer Nichtübereinstimmung wird das als Fehler gewertet und abgefangen. Dieser Mechanismus verhindert, dass beliebige Dateien als gültig gelesen werden können. [Fel13, § 3.4]

Die `auto` Felder verhalten sich wie die Variablen in der Programmiersprache Java, die mit dem Modifier *transient* gekennzeichnet sind [GJS+13, § 8.3.1.3]. Somit werden diese Felder vom Codegenerator generiert, jedoch nicht serialisiert. Diese zusätzlichen Felder in der Datenstruktur sollen Berechnungen vereinfachen, indem davon ausgegangen wird, dass die Werte nach der Berechnung nicht mehr benötigt werden. Dieser Mechanismus kann auch für Felder genutzt werden, die sehr schnell berechnet werden können und somit nicht serialisiert werden müssen. Das Schlüsselwort `auto` wurde dafür

gewählt, weil davon ausgegangen wird, dass der Wert des Feldes automatisch berechnet wird. [Fel13, § 3.4]

2.1.5 Reflexion

Es ist sicherzustellen, dass auch beliebige unbekannte Dateien gelesen werden können, ohne dabei unberechtigte Fehler zu produzieren. Daher muss bei der Deserialisierung das serialisierte Typsystem verarbeitet werden, sodass auch mit unbekannten Typen umgegangen werden kann. [Fel13]

2.2 Dateiformat

Das Dateiformat besteht aus einer sich wiederholenden Kombination aus einem String- und Typblock. Das stellt sicher, dass neue Instanzen schnell angehängt werden können. Um die Typsicherheit zu gewähren, werden die Typen mit den dazugehörigen Felddaten gemeinsam in einem Typblock gespeichert. Zudem erlaubt diese Vorgehensweise, mit unbekannten Typen umgehen zu können. [Fel13, § 6.2]

2.2.1 Stringblock

Der Stringblock wurde mit der Intention entworfen, möglichst viele Daten überspringen zu können. Das hat den Vorteil, dass bei vielen Strings, die hauptsächlich für die Felddaten bestimmt sind, nicht gelesen werden müssen. Ein Stringblock besteht deshalb aus zwei Einheiten. Die erste Einheit hält die Endpositionen aller Strings im Integer-Typ i32 vor und die zweite Einheit enthält die Daten. Falls der Index eines Strings bekannt ist, kann nun aus der ersten Einheit die Länge und Position des Strings in der zweiten Einheit berechnet werden. Mit diesem Mechanismus kann fast der komplette Stringblock übersprungen werden. [Fel13, § 6.2.1]

2.2.2 Typblock

Um beschädigte Dateien frühzeitig erkennen zu können, müssen die Typinformationen gelesen werden, bevor die Felddaten gelesen werden. Aus diesem Grund besteht auch der Typblock aus zwei Einheiten. Die erste Einheit beinhaltet die Typdeklarationen und in der zweiten Einheit folgen die dazugehörigen Felddaten der Typdeklarationen. Eine Typdeklaration enthält Informationen über seinen Supertyp, die Anzahl der Instanzen, die Restriktionen[1] und die Felddeklarationen. Eine Felddeklaration wiederum beinhaltet die Restriktionen[1], den Typ, den Namen und die Endposition der dazugehörigen Felddaten. Sollte eine Typdeklaration bereits verarbeitet worden sein, wird davon ausgegangen, dass die Informationen über seinen Supertyp und die Restriktionen[1] bei der nächsten Verarbeitung der Typdeklaration nicht mehr vorhanden sind. Dies gilt auch für die Felddeklarationen aus den jeweiligen Typdeklarationen. Falls eine Felddeklaration bereits verarbeitet worden ist, wird

[1] Im Rahmen dieser Arbeit wird nur auf die SKilL Kernsprache und die Floats eingegangen.

davon ausgegangen, dass die Restriktionen[1], der Typ und der Name bei der nächsten Verarbeitung der Felddeklaration nicht mehr vorhanden sind. [Fel13, § 6.2.2]

2.3 Vergleich des Speicherplatzverbrauchs: i64 und v64

Die Abbildung 2.1 visualisiert den Speicherplatzverbrauch der Integer-Typen i64 und v64 auf der Festplatte. Es wurden jeweils n i64- und v64-Werte geschrieben. Dabei wurde der Wert jeweils mit der Zahl beginnend bei der Eins aufsteigend in Einer-Schritten gesetzt (siehe Listing 2.2).

Listing 2.2: i64 / v64

```
for I in 1 .. N loop
  Write_i64 (I); -- oder Write_v64 (I);
end loop;
```

Man erkennt, dass beim Typ i64 jeder Wert immer acht Bytes benötigt. Beim Typ v64 hängt die Anzahl der benötigten Bytes vom abzuspeichernden Wert ab. Kleine Werte ($[0, 128[\in \mathbb{N}_0$) benötigen immer ein Byte. Aufsteigende Werte bis $0.25 * 10^9$ benötigen im Durchschnitt immer noch nur die Hälfte an Bytes im Vergleich zum Typ i64. Jedoch benötigen große ($\geq 2^{55}$) und negative Werte immer neun Bytes.

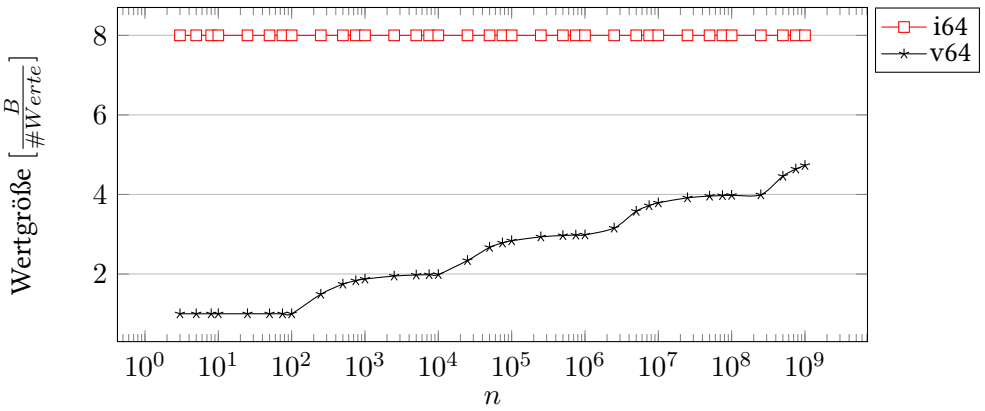

Abbildung 2.1: Durchschnittlicher Speicherplatzverbrauch auf der Festplatte pro Wert in Bytes.

Aus diesem Grund verwendet die Serialisierungssprache SKilL den Typ v64 für die Referenzen. Im Arbeitsspeicher werden beim Ada-Binding die beiden Integer-Typen als Long repräsentiert und benötigen für jeden Wert immer acht Bytes.

3 Verwandte Arbeiten

Die verwandten Arbeiten zur Serialisierungssprache SKilL werden im Technischen Bericht 2013/06 [Fel13, § 1.2] vorgestellt, das insbesondere die Unterschiede zu Apache Thrift, Protocol Buffers und XML beleuchtet. Zurzeit ist das Scala-Binding die einzige verwandte Arbeit zu dieser Arbeit.

3.1 Scala-Binding

Das Scala-Binding ist derzeit das einzige verfügbare Binding für die Serialisierungssprache SKilL [SKI]. Es dient in dieser Arbeit als Referenzimplementierung und wird für die Performance-Evaluation des Ada-Bindings herangezogen.

Das Listing 3.1 zeigt ein allgemeines Codebeispiel zur Nutzung des Scala-Bindings. Das allgemeine Codebeispiel zur Nutzung des Ada-Bindings in Listing 5.4 ist an diesem Codebeispiel angelehnt.

Listing 3.1: Codebeispiel

```
val State = SkillState.create // oder SkillState.read(fileName)
State.T(parameter1, parameter2, ...)
State.write(fileName) // oder State.append
State.T(parameter1, parameter2, ...)
State.append

val instances = State.T.all
instances.foreach { o => println(o.field) }
```

Erläuterungen zum Listing 3.1

T: Ein benutzerdefinierter Typ.

*parameter**: Die Felder des benutzerdefinierten Typs *T*.

Der Hauptunterschied (abgesehen von der Programmiersprache) zwischen dem Scala- und Ada-Binding ist, dass das Scala-Binding die Serialisierungssprache SKilL zum jetzigen Zeitpunkt fast vollständig abdeckt. Das Ada-Binding deckt derzeit wie gefordert nur die Kernsprache und zusätzlich die Floats ab.

Zudem ist zu beachten, dass während dieser Arbeit stetig am Scala-Binding weiterentwickelt wurde. Deshalb wurde in Kapital 4 eine Version des Scala-Bindings festgesetzt, die für die nachfolgenden Performancetests verwendet wurde.

4 Testumgebung

In diesem Kapitel wird die Testumgebung für alle Tests in den nachfolgenden Kapiteln vorgestellt. Die Tests wurden auf einem Apple MacBook Air (Mid 2013) mit folgender Ausstattung ausgeführt:

CPU: 1,7 GHz Intel Core i7-4650U

RAM: 8 GB 1600 MHz DDR3

SSD: APPLE SSD SD0256F

Betriebssystem: Mac OS X 10.9

4.1 Ada

Die Tests für die Programmiersprache Ada wurden mit dem GNAT 2013 (GPL Edition von AdaCore) Compiler [Ada13b] und den nachfolgenden Binder- und Compiler-Parametern kompiliert. Zudem unterstützt dieser Compiler defaultmäßig den neusten Ada Standard 2012.

Binder-Parameter

`-gnatn`: Aktiviert das Back-End Inlining von pragma *Inline* spezifizierten Unterprogrammen [Ada13a, § 3.2].

`-gnatN`: Aktiviert das Front-End Inlining von pragma *Inline* spezifizierten Unterprogrammen [Ada13a, § 3.2].

`-gnatp`: Unterdrückt alle Laufzeitüberprüfungen [Ada13a, § 3.2.6].

Compiler-Parameter

`-march=corei7`: Generiert Anweisungen für die Prozessor-Architektur Core i7 [SGDC13, § 3.17.16].

`-O3`: Aktiviert die höchste Optimierungsstufe [SGDC13, § 3.10].

4.2 Scala

Die Tests für die Programmiersprache Scala wurden mit den nachfolgenden VM-Parametern auf der Oracle JVM 1.7.0_51 ausgeführt. Da sich während dieser Arbeit das Scala-Binding stetig in Entwicklung befand, wurde die Version des Scala-Bindings mit dem Commit `5c44071` vom 4. April 2014 festgesetzt.

VM-Parameter

- `-Xmx7680m`: Legt die maximale Heapgröße auf 7,5 GB fest [Ora14].
- `-XX:MaxHeapFreeRatio=99`: Verkleinert den Heap erst, wenn mehr als 99 % freier Heapspeicher zur Verfügung steht. Daher gibt die JVM mit einer hohen `MaxHeapFreeRatio` nicht benötigten Heapspeicher eher nicht mehr frei. [Ora14]

5 Ada-Binding

In diesem Kapitel werden das Softwaredesign und die entdeckten Bottlenecks beschrieben sowie ein allgemeines Codebeispiel zur Nutzung des Ada-Bindings gegeben.

5.1 Softwaredesign

In dieser Sektion werden das Paketdiagramm eines generierten Ada-Bindings, die Flussdiagramme für das Lesen und Schreiben von SKilL-Dateien sowie das Mapping der SKilL-Typen auf die entsprechenden Ada-Typen vorgestellt.

5.1.1 Paketdiagramm

Ein generiertes Ada-Binding lässt sich in sieben Pakete unterteilen (siehe Abbildung 5.1). Der Nutzer kommuniziert ausschließlich mit den Paketen `User_Defined` und `Api`.

`User_Defined`: Für jede SKilL-Spezifikation wird ein Paketname erwartet. Dieser angegebene Paketname ersetzt den Begriff `User_Defined` im Paketdiagramm. Es enthält alle benutzerdefinierten Typen, die Zugriffsfunktionen auf die Felder, den SKilL-Zustand sowie die Hilfsfunktionen (z. B. Hash- und Vergleichsfunktionen) für die zusammengesetzten Typen.

`Api`: Dieses Paket stellt dem Nutzer die Kernfunktionalität zum Manipulieren eines SKilL-Zustandes zur Verfügung. Die Kernfunktionalität beinhaltet das Erstellen eines neuen Zustandes, das Lesen und Schreiben von Dateien sowie das Anhängen von neuen Feldern und Instanzen an ein bereits gelesenen oder geschriebenen Zustand. Weil der Nutzer keinen direkten Zugriff auf ein Storage Pool erhält, enthält es außerdem Funktionen zum Verwalten von Instanzen aus dem Storage Pool.

`Api.Internal.File_Reader`: Dieses Paket liest eine Datei mit Hilfe des Pakets `Byte_Reader` ein. In einer Schleife werden jeweils der String- und Typblock bis an das Ende einer Datei eingelesen. Da die Felddaten erst nach den Typ- und Felddeklarationen kommen, wird eine Warteschlange für die Felddaten gebildet und nach dem Einlesen aller Typdeklarationen abgearbeitet.

`Api.Internal.File_Writer`: Dieses Paket schreibt eine Datei mit Hilfe der Pakete `Byte_Writer` und `Byte_Reader`. Es kann sowohl in eine Datei schreiben sowie an eine Datei anhängen. Es schreibt jeweils den String- und Typblock. Da bei den Felddeklarationen auch die Länge der Felddaten benötigt wird, schreibt es zur Ermittlung der Länge der Felddaten die Felddaten in einen temporären Stream und kopiert nach den Typdeklarationen die Felddaten aus dem

temporären Stream in die Datei. Aus diesem Grund hat dieses Paket eine Abhängigkeit zum Paket `Byte_Reader` im Paketdiagramm.

`Api.Internal.State_Maker`: Dieses Paket sorgt dafür, dass ein Zustand immer alle bekannten benutzerdefinierten Typen nach dem Erstellen eines neuen Zustandes sowie nach dem Einlesen einer Datei kennt.

`Api.Internal.Byte_Reader`: Dieses Paket liest je nach Bedarf die benötigten Bytes aus einem Stream und baut den angeforderten Typen in der korrekten Byte-Reihenfolge (Big-Endian) zusammen.

`Api.Internal.Byte_Writer`: Analog zum `Byte_Reader`.

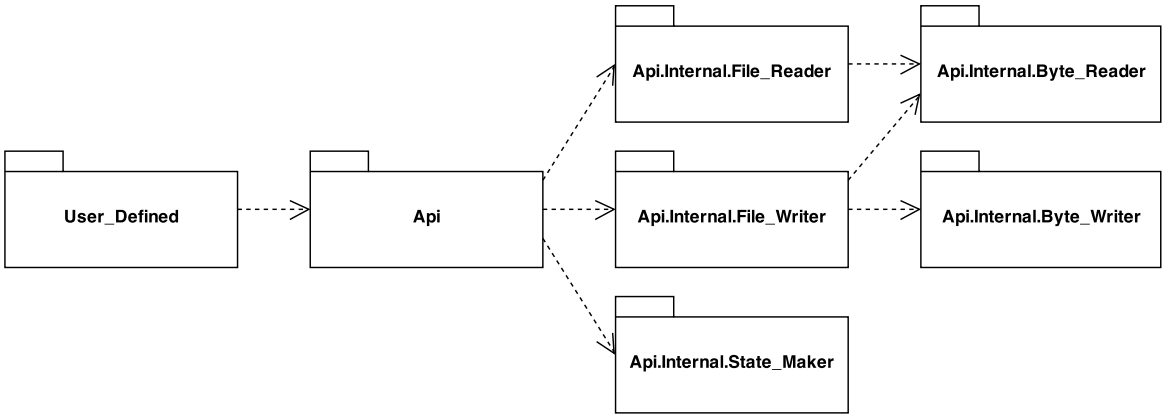

Abbildung 5.1: Das Paketdiagramm eines generierten Ada-Bindings.

5.1.2 Flussdiagramme

In dieser Sektion werden die Flussdiagramme für das Lesen und Schreiben einer SKilL-Datei vorgestellt. Es existieren sechs Flussdiagramme. Die ersten drei Flussdiagramme beschreiben das Vorgehen, wie eine SKilL-Datei eingelesen und ein String- und Typblock verarbeitet werden. Die letzten drei Flussdiagramme beschreiben das Vorgehen, wie ein String- und Typblock in eine SKilL-Datei geschrieben oder angehängt werden.

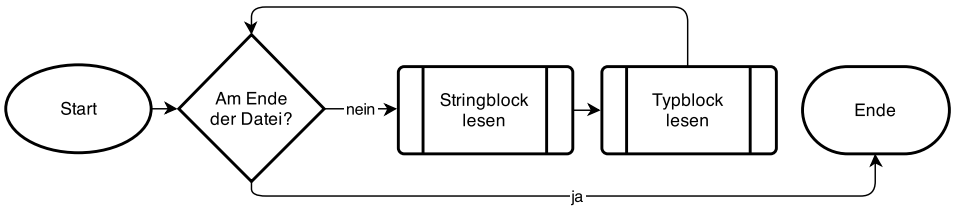

Abbildung 5.2: Das Flussdiagramm zum Lesen einer SKilL-Datei. In einer Schleife werden jeweils der String- und dann der Typblock bis an das Ende der Datei eingelesen.

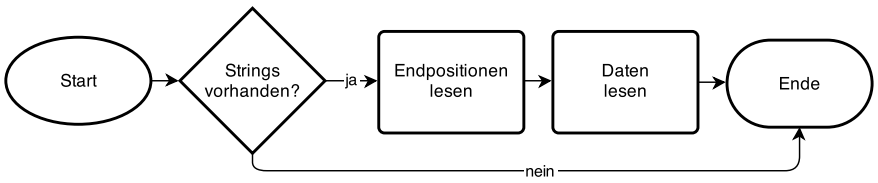

Abbildung 5.3: Das Flussdiagramm zum Lesen eines Stringblocks. Falls der Stringblock nicht leer ist, werden zuerst die Endpositionen und dann die dazugehörigen Daten gelesen.

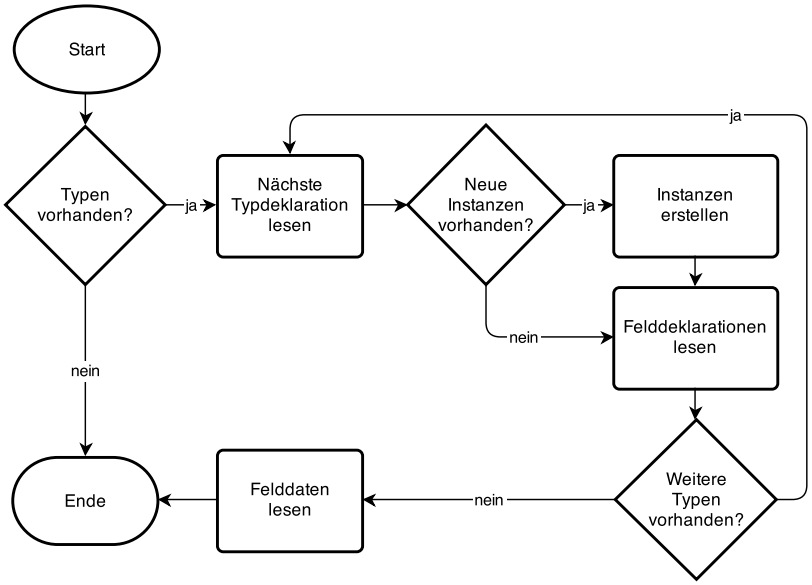

Abbildung 5.4: Das Flussdiagramm zum Lesen eines Typblocks. Falls der Typblock nicht leer ist, werden die Typdeklarationen, die dazugehörigen Felddeklarationen und danach die Felddaten gelesen.

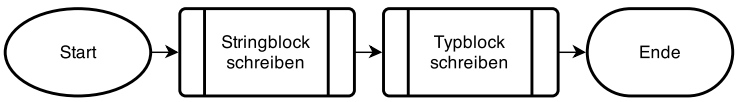

Abbildung 5.5: Das Flussdiagramm zum Schreiben einer SKilL-Datei. Zuerst wird der String- und dann der Typblock geschrieben.

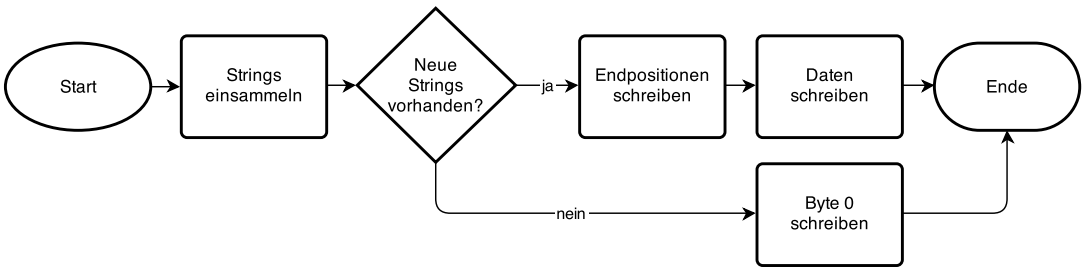

Abbildung 5.6: Das Flussdiagramm zum Schreiben eines Stringblocks. Als Erstes werden die Strings eingesammelt. Falls neue Strings vorhanden sind, werden zuerst die Endpositionen und dann die Daten geschrieben. Ansonsten wird ein Byte mit dem Wert null geschrieben.

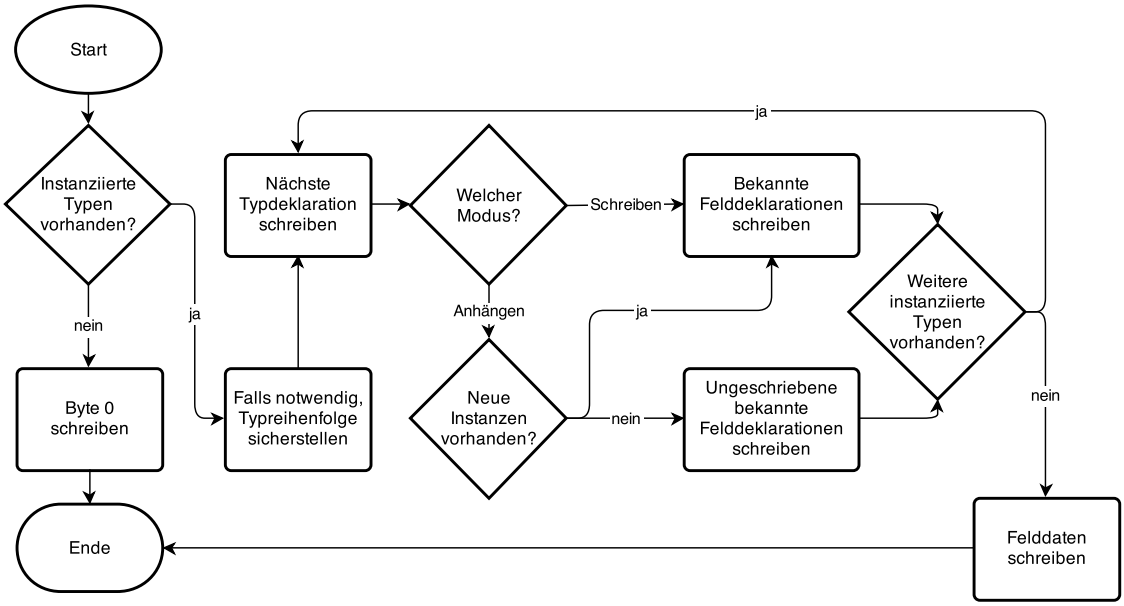

Abbildung 5.7: Das Flussdiagramm zum Schreiben eines Typblocks. Falls instanziierte Typen vorhanden sind, wird, falls notwendig, die Typreihenfolge der Instanzen für alle instanziierte Typen sichergestellt. Dann werden die Typdeklarationen, die dazugehörigen Felddeklarationen in Abhängigkeit des Modus und danach die Felddaten geschrieben. Ansonsten wird ein Byte mit dem Wert null geschrieben.

5.1.3 Typen

Die Serialisierungssprache SKilL bietet verschiedene Typen an, die nun auf die entsprechenden Ada-Typen abgebildet werden müssen. Die Tabelle 5.1 zeigt, wie die SKilL-Typen auf die entsprechenden Ada-Typen bei einem generierten Ada-Binding abgebildet werden.

SKilL	Ada
annotation	Skill_Type_Access
bool	Boolean
i8	i8
i16	i16 oder Short
i32	i32
i64	i64 oder Long
v64	v64 oder Long
f32	f32 oder Float
f64	f64 oder Double
string	String_Access
T[i]	array (1 .. i) of T
T[]	Ada.Containers.Vectors
list<T>	Ada.Containers.Doubly_Linked_Lists
set<T>	Ada.Containers.Hashed_Sets
map<T_1, ..., T_n>	Ada.Containers.Hashed_Maps
T	T_Type_Access

Tabelle 5.1: Das Mapping der SKilL-Typen auf die entsprechenden Ada-Typen.

In Ada können die Wertebereiche des Integer- und Long_Integer-Typs je nach Plattform variieren. Die einzige Implementierungsvoraussetzung eines Ada-Compilers ist, dass der Integer-Typ mindestens 16-Bit und der Long_Integer-Typ mindestens 32-Bit lang sein müssen [TDB+12, § 3.5.4, Absatz 21-22]. Aus diesem Grund nutzt das Ada-Binding die Integer- und Float-Typen aus dem Paket Interfaces (siehe Listing 5.1).

Listing 5.1: Integer- und Float-Typen

```
type i8 is new Interfaces.Integer_8;
type i16 is new Interfaces.Integer_16;
subtype Short is i16;
type i32 is new Interfaces.Integer_32;
type i64 is new Interfaces.Integer_64;
subtype v64 is i64;
subtype Long is i64;
type f32 is new Interfaces.IEEE_Float_32;
subtype Float is f32;
type f64 is new Interfaces.IEEE_Float_64;
subtype Double is f64;
```

In Ada muss bei einer Stringdefinition die Länge des Strings mitangegeben werden. Bei einem Array mit konstanter Länge ist dies aber nicht möglich, weil die Strings unterschiedliche Längen haben könnten. Um dieses Problem zu umgehen, werden für Strings Referenzen verwendet (siehe Listing 5.2).

Listing 5.2: String-Typ

```
type String_Access is access String;
```

In Ada kann zwischen geordneten und gehashten Maps und Sets gewählt werden. Es wird jedoch davon ausgegangen, dass die meisten Nutzer eher ein gehashtes als ein geordnetes Map oder Set erwarten.

Um die Vererbung bei den benutzerdefinierten Typen zu ermöglichen, wurde dieses Problem objektorientiert gelöst. Ein benutzerdefinierter Typ wird immer vom `tagged Skill_Type`-Typ oder einem anderen benutzerdefinierten Typ abgeleitet. Der `Skill_Type`-Typ ist `abstract`, sodass keine Instanzen davon erstellt werden können. Außerdem sind sowohl der `Skill_Type`-Typ als auch die benutzerdefinierten Typen `private`, sodass der Nutzer nur über die sichtbaren Zugriffsfunktionen auf die Felder zugreifen kann. Der Nutzer arbeitet immer mit Referenzen zu den benutzerdefinierten Typen (siehe Abbildung 5.3). Jeder benutzerdefinierte Typ kann zum `Skill_Type`-Typ konvertiert werden, das z. B. bei der Annotation notwendig ist.

Listing 5.3: Benutzerdefinierter Typ

```
type Skill_Type is abstract tagged private;
type Skill_Type_Access is access all Skill_Type'Class;

type T_Type is new Skill_Type with private;
type T_Type_Access is access all T_Type;
```

Weil das Ada-Binding auf dem Heap allokiert (dynamische Speicherreservierung), ist es selbst für die Freigabe des Speichers verantwortlich.

5.2 Bottlenecks

In dieser Sektion werden die drei wesentlichen Bottlenecks der naiven Implementierung beschrieben, die nach der Auflösung zur Performanceverbesserung des Ada-Bindings beitrugen.

5.2.1 Buffer

In der naiven Implementierung nutzten die Pakete `Byte_Reader` und `Byte_Writer` keinen Buffer für die zu lesenden oder schreibenden Bytes. Die Abbildung 5.8 zeigt die Performanceverbesserung nach dem Einbau des Buffers für die unterschiedlichen Buffergrößen. Zu beachten ist, dass währenddessen keine Integer-Typen zusammengebaut oder aufgesplittet werden, d. h. es werden nur einzelne Bytes gelesen oder geschrieben (siehe A.1 und A.4). Auffallend ist der deutliche Performanceunterschied zwischen dem Lesen und Schreiben. Der Buffer muss zudem noch nicht einmal im Megabyte-Bereich liegen, es reichen schon wenige Kilobytes aus.

Aus dieser Konsequenz heraus nutzt das Binding einen Buffer mit der Größe von 2^{12} (= 4096) Bytes. Diese Größe ist seit Januar 2011 die standardisierte Sektorgröße moderner Festplatten, auch Advanced Format Drives (AFD) genannt [Sea].

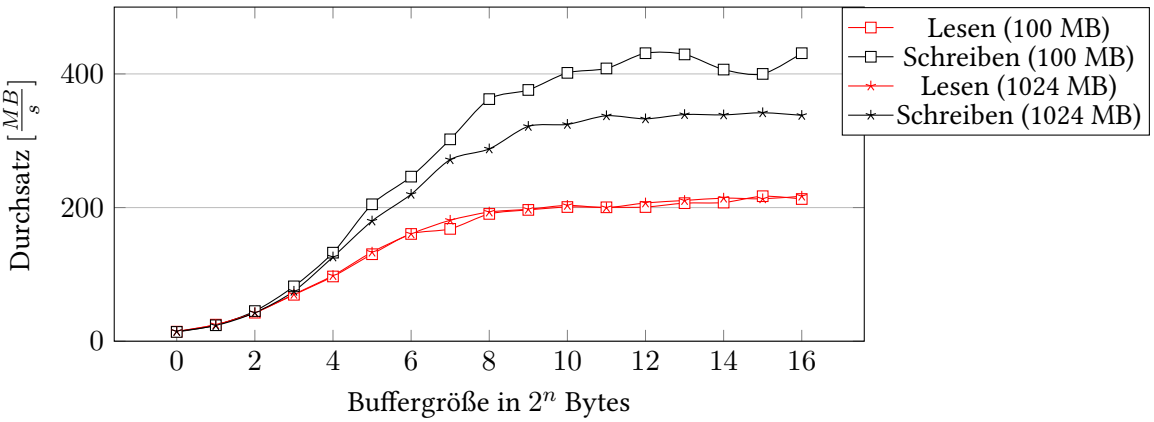

Abbildung 5.8: Durchschnittlicher Durchsatz in MB/s je Buffergröße.

5.2.2 Indefinite Containers

Das Binding nutzt intern nur Strings statt Stringreferenzen außer für die Stringfelder der benutzerdefinierten Typen. In der naiven Implementierung wurden für alle Container die `indefinite` Variante gewählt, ohne die Performanceeinbußen zu kennen. Die Abbildung 5.9 und Tabelle 5.2 zeigen, dass das Anhängen von n Referenzen in einen `Vector` um mindestens 200 % schneller ist als in einen `Indefinite_Vector`.

Aus dieser Konsequenz heraus setzt das Ada-Binding nur für den String-Pool-Vector und die Typen-Hashmap die `indefinite` Container ein. Bei allen anderen Container, insbesondere beim Storage-Pool-Vector, wurde die normale Variante gewählt.

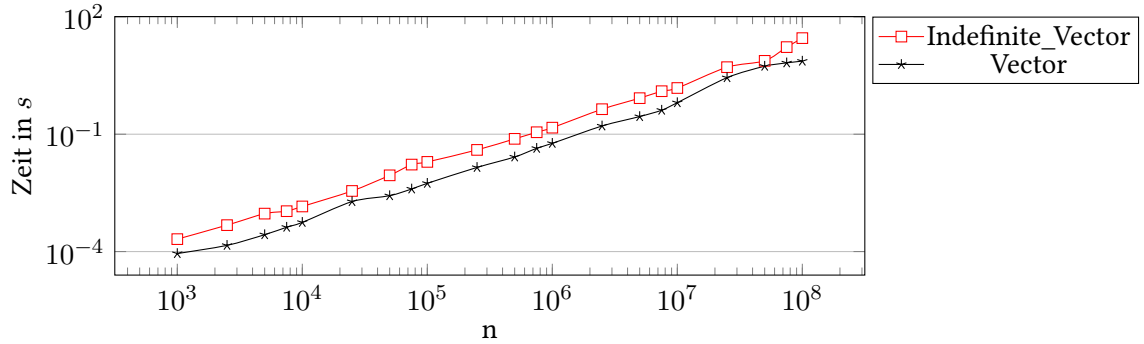

Abbildung 5.9: Gemessene Laufzeit für das Anhängen von n Referenzen in einen `Indefinite_Vector` und `Vector`.

n	Indefinite_Vector in s	Vector in s	speed-up
10^3	0,0002	0,0001	200 %

10^4	0,0014	0,0006	233 %
10^5	0,0196	0,0056	350 %
10^6	0,1471	0,0584	252 %
10^7	1,5179	0,6401	237 %
10^8	28,6132	7,3689	388 %

Tabelle 5.2: Gemessene Laufzeit für das Anhängen von n Referenzen in einen `Indefinite_Vector` und `Vector`.

5.2.3 v64 Schreibalgorithmus

In der naiven Implementierung wurde der v64 Schreibalgorithmus aus dem Anhang des Technischen Berichts 2013/06 [Fel13] übernommen (siehe A.6). Da dieser mit 50 MB/s nicht effizient genug ist, wurden zwei weitere Algorithmen ausprobiert. Als Erstes wurde er durch den Algorithmus aus dem Scala-Binding ausgetauscht, der die manuell ausgeführte Optimierungsmethode *Loop unrolling* verwendet (siehe A.7). Als Zweites wurden die if-Abfragen durch eine BSR-Instruktion [Int11, § 3-95] und case-Anweisung ersetzt (siehe A.8). Die beiden neuen Algorithmen scheinen jedoch laut Abbildung 5.10 gleichwertig zu sein.

Aus dieser Konsequenz heraus nutzt das Binding den v64 Schreibalgorithmus aus dem Scala-Binding ohne die BSR-Modifikation, um die Portabilität und somit die Plattformunabhängigkeit des Bindings zu gewähren.

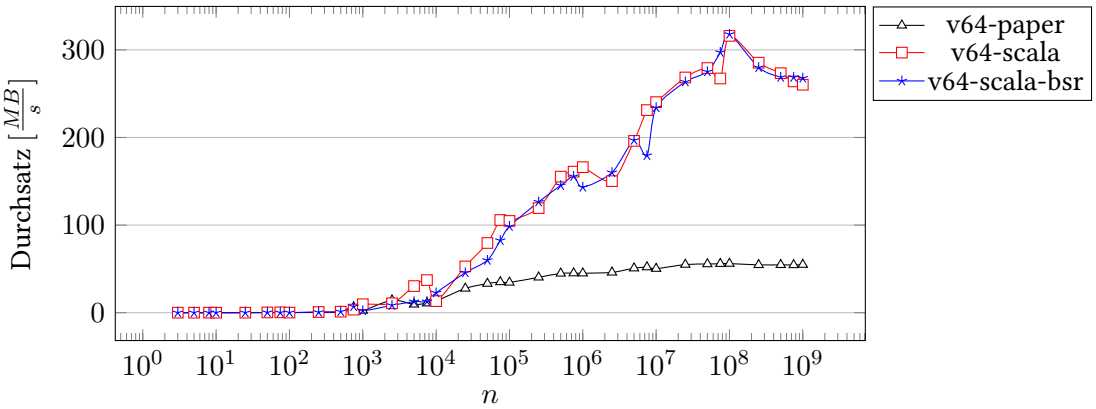

Abbildung 5.10: Gemessener Durchsatz in MB/s mit einer Buffergröße von 2^{12} Bytes für die v64 Schreibalgorithmen aus dem Anhang des Technischen Berichts 2013/06 [Fel13] und dem Scala-Binding [SKI], jeweils mit und ohne die BSR-Modifikation [Int11].

5.3 Codebeispiel

Die Abbildung 5.4 zeigt ein allgemeines Codebeispiel zur Nutzung des Ada-Bindings. Das Codebeispiel ist analog zum allgemeinen Codebeispiel des Scala-Bindings in Abbildung 3.1. Es wurde versucht, dem Stil des Scala-Bindings nahezukommen, ohne sich von der Programmiersprache Ada zu entfremden. Man beginnt immer damit, einen neuen Zustand zu erstellen. Dieser Zustand wird dazu genutzt, ihn mit den bekannten Typen zu befüllen (`Create`) oder eine Datei einzulesen (`Read`). Danach kann je nach Situation entweder der Zustand in eine Datei geschrieben (`Write`) oder angehängt (`Append`) werden. Anschließend können die Instanzen eines Storage Pools aus dem Zustand geholt (`Get_`*`T`*`s`) werden. Weil die Programmiersprache Ada keinen GC nutzt, gibt es im Gegensatz zum Scala-Binding außerdem eine `Close`-Funktion, um den nicht mehr benötigten Speicher freizugeben und dadurch Speicherlecks zu verhindern.

Listing 5.4: Codebeispiel

```
declare
  State : access Skill_State := new Skill_State;
begin
  Create (State); -- oder Read (State, File_Name);
  New_T (State, Parameter1, Parameter2, ...);
  Write (State, File_Name); -- oder Append (State);
  New_T (State, Parameter1, Parameter2, ...);
  Append (State);

  declare
    Instances : T_Type_Accesses := Get_Ts (State);
  begin
    for I in Instances'Range loop
      Ada.Text_IO.Put_Line (Instances (I).Get_Field);
    end loop;
  end;

  Close (State);
end;
```

Die Prozedur `New_`*`T`* gibt es auch als Funktion, die die erstellte Instanz zurückliefert. Das ermöglicht es, mit der erstellten Instanz direkt weiterarbeiten zu können.

Listing 5.5: New_*T*-Funktion

```
declare
  Instance : T_Type_Access := New_T (State, Parameter1, Parameter2, ...);
begin
  Instance.Set_Field (Value); -- oder Get_Field (Value);
end;
```

Die Funktion `Get_`*`T`*`s` kopiert alle Instanzen aus einem Storage Pool in ein Array. Die Datenkapselung soll verhindern, dass der Nutzer direkten Zugriff auf den Storage Pool erhält. In manchen Situationen ist dieses Array aber nicht notwendig, wenn z. B. nur wenige Instanzen benötigt werden. Aus diesem Grund gibt es die Funktion `Get_`*`T`*, die eine einzelne Instanz an einem bestimmten Index zurückliefern kann.

Listing 5.6: Get_*T*-Funktion

```
declare
  Instance : T_Type_Access := Get_T (State, Index);
begin
  Instance.Set_Field (Value); -- oder Get_Field (Value);
end;
```

5.4 Performance der Byte-Pakete

Die Abbildung 5.11 visualisiert die Performance der Pakete `Byte_Reader` und `Byte_Writer` für die Integer-Typen i64 und v64 mit einer Buffergröße von 2^{12} (= 4096) Bytes.

Im Unterschied zu Abbildung 5.8, das nur einzelne Bytes gelesen oder geschrieben hat (siehe A.1 und A.4) und somit keine Integer-Typen zusammengebaut oder aufgesplittet hat, werden hier die Integer-Typen beim Lesen zusammengebaut und beim Schreiben aufgesplittet (siehe A.2, A.3, A.5 und A.7). Obwohl das Zusammenbauen aus einzelnen Bytes oder das Aufsplitten in einzelne Bytes aufwendiger ist als das direkte Lesen oder Schreiben von Bytes, so ist die Performance für die Integer-Typen i64 und v64 erheblich besser (siehe Abbildung 5.8 und Abbildung 5.11).

Im Rahmen dieser Arbeit konnte keine plausible Erklärung für dieses Verhalten gefunden werden. Ein Hinweis für einen möglichen Lesecache findet sich aber bei $n = 10^9$ durch den plötzlichen Geschwindigkeitsabfall beim Lesen von ca. 800 MB/s auf ca. 450 MB/s. Die Geschwindigkeit von ca. 450 MB/s war die maximale Lesegeschwindigkeit einzelner Bytes. Deshalb kann davon ausgegangen werden, dass aus zurzeit unbekannten Gründen der Lesecache beim Lesen von einzelnen Bytes nicht greift.

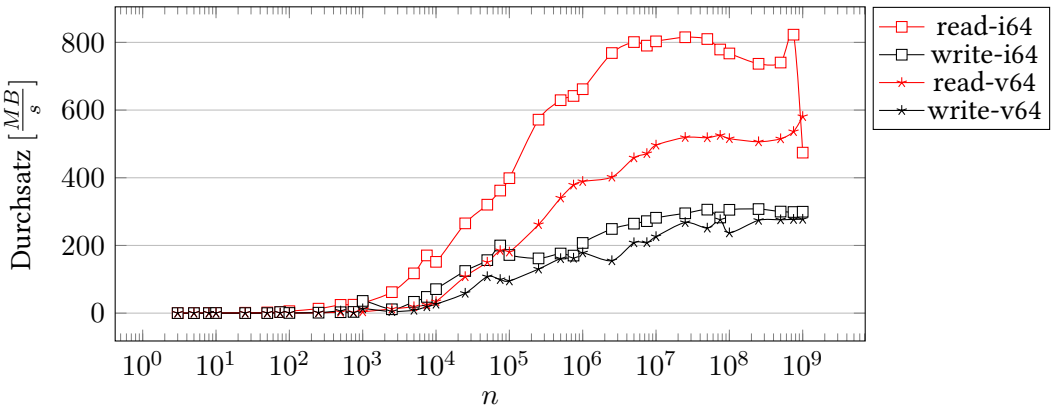

Abbildung 5.11: Gemessener Durchsatz in MB/s mit einer Buffergröße von 2^{12} (= 4096) Bytes für das Lesen und Schreiben der Integer-Typen i64 und v64.

5.5 Weitere Hinweise

Es wurde vorgegeben, dass das Ada- und Scala-Binding ein gemeinsames Front-End besitzen. Deshalb wurde für die Entwicklung des Ada-Bindings der Codegenerator für die Programmiersprache Scala dupliziert und entsprechend modifiziert [SKI]. Durch diese Vorgehensweise ist sichergestellt, dass große Codeteile vom bereits vorhandenen Codegenerator wiederverwendet werden und beide Codegeneratoren in gleicher Weise funktionieren. Das erleichtert die Erstellung eines einheitlichen Codegenerators für die Programmiersprachen Ada und Scala sowie deren Wartung.

Für die Tests des Ada-Bindings wurde das Test-Framework Ahven in der Version 2.4 verwendet [Kos]. Es existieren Tests, die die SKilL Kernsprache und die Floats abdecken.

Im Gegensatz zum Scala-Binding werden die SKilL-IDs für die Instanzen bereits bei der Erstellung der Instanz und nicht erst beim Schreiben oder Anhängen vergeben.

Die weitere Dokumentation zum Ada-Binding ist in den generierten Spezifikationsdateien des Ada-Bindings zu finden.

6 Performance-Evaluation

In diesem Kapitel wird das Ada- und Scala-Binding bezüglich ihrer Performance verglichen.

Die Performance-Evaluation besteht aus vier verschiedenen Tests. Die ersten zwei namens Number und Date sind zwei simple Performancetests. Die letzten zwei namens Graph 1 und Graph 2 (in Anlehnung an [Fel14]) sind zwei komplexere Performancetests, die als praxisnah angesehen werden.

Jeder Test durchläuft jeweils für das Ada- und Scala-Binding folgende fünf Phasen:

`create`: In der ersten Phase werden n Instanzen erstellt. Es wird die gemessene Laufzeit dieser Operation protokolliert.

`write`: In der zweiten Phase werden die erstellten Instanzen in eine Datei geschrieben. Es wird die gemessene Laufzeit dieser Operation und die Dateigröße protokolliert.

`read`: In der dritten Phase wird die zuvor geschriebene Datei eingelesen. Es wird die gemessene Laufzeit dieser Operation protokolliert.

`create-more`: In der vierten Phase werden zu den bereits vorhandenen Instanzen n weitere Instanzen erstellt. Es wird die gemessene Laufzeit dieser Operation protokolliert.

`append`: In der fünften Phase werden die neuen Instanzen an die zuvor geschriebene Datei angehängt. Es wird die gemessene Laufzeit dieser Operation und die Dateigröße protokolliert.

Jeder Test wurde zehn mal wiederholt. Die Anzahl der Wiederholungen waren aufgrund der Verfügbarkeit der Testhardware limitiert. Es wurde der Durchschnitt aus den elf Runden gebildet.

Listing 6.1: Ablauf des Benchmarks

```
for R in 0 .. Repetitions loop -- Ein obligatorischer Durchlauf und die Wiederholungen
  for E in 1 .. Exponent loop
    for P of Phases loop -- Die fünf Phasen: create, write, read, create-more, append
      Measure ((10 ** E) * 0.25, P);
      Measure ((10 ** E) * 0.50, P);
      Measure ((10 ** E) * 0.75, P);
      Measure ((10 ** E) * 1.00, P);
    end loop;
  end loop;
end loop;

-- Korrigiere die Dateigröße in der append-Phase
-- Bilde die Durchschnitte aus den gemessenen Werten
-- Generiere die Graphen
```

Aus den protokollierten Daten wurden vier Graphen generiert. Der erste und zweite Graph visualisieren die gemessene Laufzeit in s aller fünf Phasen für jedes n. Der dritte Graph visualisiert die durchschnittlichen geschriebenen Bytes pro Instanz für jedes n. Für die `append`-Phase wurde von der protokollierten Dateigröße die Dateigröße aus der `write`-Phase abgezogen. Es ist wichtig, dass in diesem Graphen die Linien beider Bindings in beiden Phasen überlappen. Ist das nicht der Fall, so ist das ein guter Indikator dafür, dass mindestens eines der Bindings falsche Dateien produziert. Der vierte Graph visualisiert den Durchsatz der I/O-Phasen `write`, `read` und `append` für jedes n. Dabei wird die Dateigröße in MB durch die gemessene Laufzeit in s geteilt.

Es wurde immer direkt vor und nach der zu messenden Operation gemessen. Dadurch wurde insbesondere die Startup-Zeit der JVM außer Acht gelassen. Des Weiteren wurde beim Scala-Binding der GC vor jeder zu messenden Operation gestartet, um die Seiteneffekte der JVM weitgehend zu minimieren.

6.1 Number

Der Test namens Number ist der erste simple Performancetest. Er beinhaltet nur einen benutzerdefinierten Typ namens *Number* mit einem Feld namens *number* vom Typ i64.

Listing 6.2: SKilL-Spezifikation

```
Number {
  i64 number;
}
```

Bei diesem Performancetest wird der Zustand mit n Number-Instanzen befüllt. Das i64-Feld wird mit der Zahl Eins aufsteigend in Einer-Schritten befüllt. Der Performancetest konnte mit dem Scala-Binding nur bis $0.25{*}10^8$ Instanzen ausgeführt werden, bevor es zum `java.lang.OutOfMemoryError: GC overhead limit exceeded` kam.

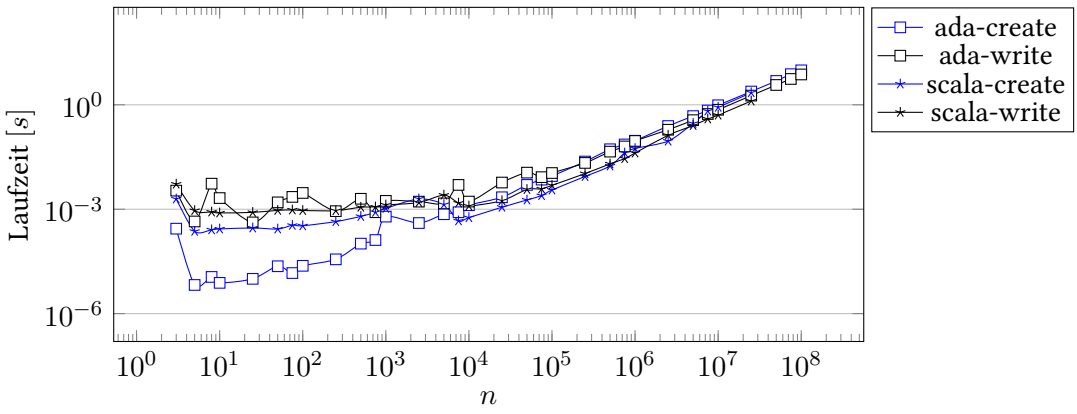

Abbildung 6.1: Gemessene Laufzeit der ersten zwei Phasen.

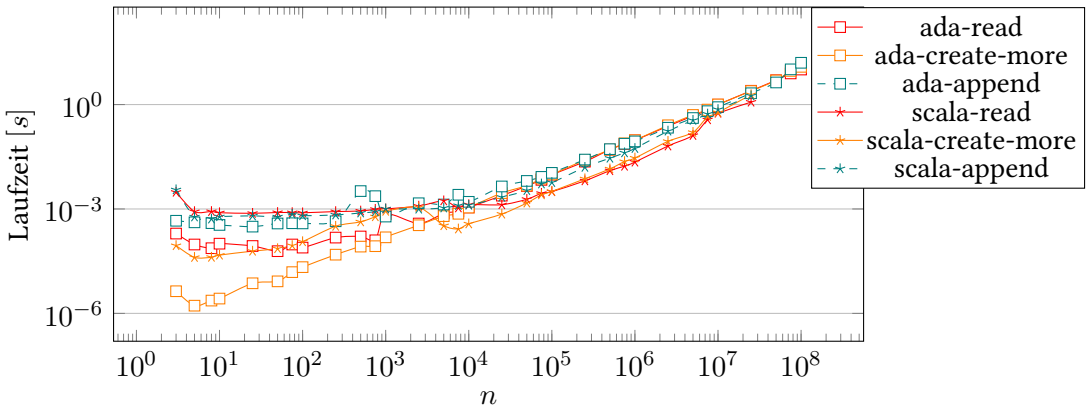

Abbildung 6.2: Gemessene Laufzeit der letzten drei Phasen.

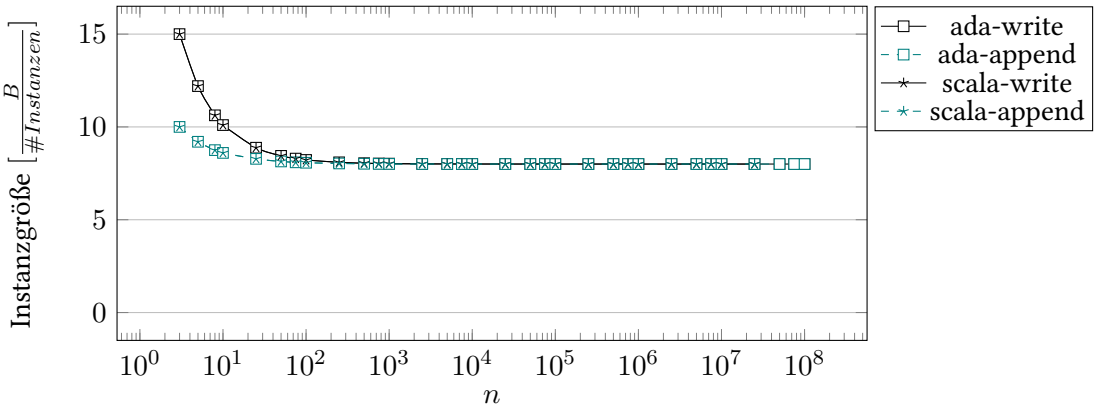

Abbildung 6.3: Gemessener Speicherplatzverbrauch auf der Festplatte pro Number-Instanz in Bytes.

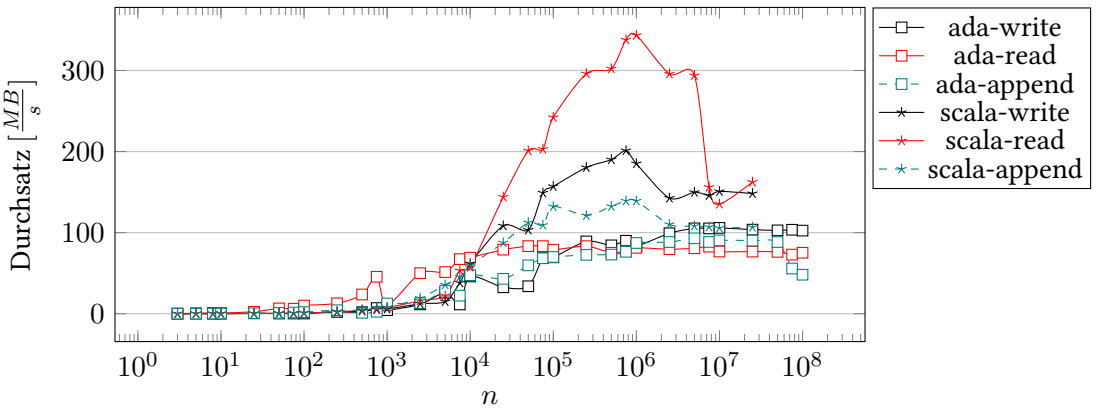

Abbildung 6.4: Gemessener Durchsatz der drei I/O-Phasen in MB/s.

Im dritten Graphen stimmen die Instanzgrößen von beiden Bindings überein. Am Anfang dominieren die Headerdaten und werden schnell vernachlässigbar. In der `append`-Phase müssen bereits bekannte Headerdaten nicht mehr geschrieben werden, das sich am Anfang stärker auf die Instanzgröße auswirkt.

Der vierte Graph zeigt, dass das Scala-Binding dem Ada-Binding überlegen ist. Das Scala-Binding erreicht beim Schreiben, Lesen und Anhängen Spitzengeschwindigkeiten von ca. 200, 345 und 140 MB/s. Im Gegensatz dazu das Ada-Binding mit Spitzengeschwindigkeiten von ca. 105, 80 und 90 MB/s.

6.2 Date

Der Test namens Date ist der zweite simple Performancetest. Er beinhaltet nur einen benutzerdefinierten Typ namens *Date* mit einem Feld namens *date* vom Typ v64. Der Unterschied zum Performancetest Number ist, dass hier ein Integer-Typ variabler Länge genutzt wird.

Listing 6.3: SKilL-Spezifikation

```
Date {
  v64 date;
}
```

Bei diesem Performancetest wird der Zustand mit n Date-Instanzen befüllt. Das v64-Feld wird mit der Zahl Eins aufsteigend in Einer-Schritten befüllt. Der Performancetest konnte mit dem Scala-Binding nur bis $0.25 * 10^8$ Instanzen ausgeführt werden, bevor es zum `java.lang.OutOfMemoryError: GC overhead limit exceeded` kam.

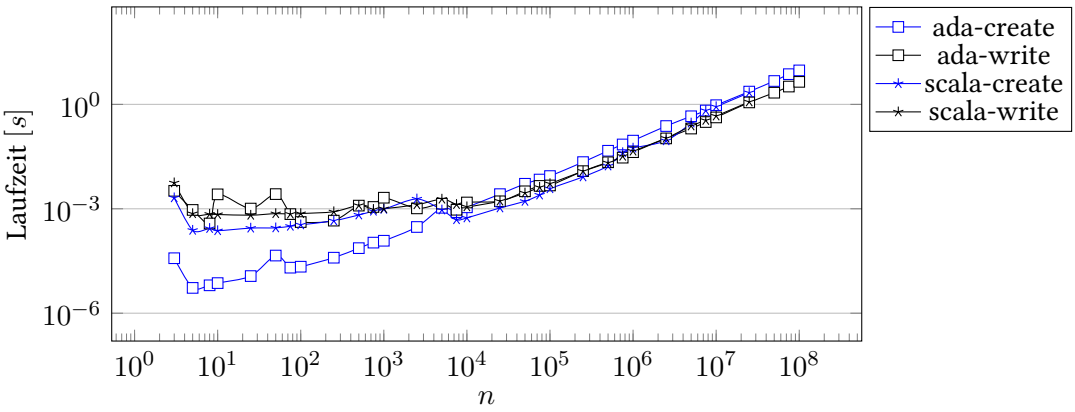

Abbildung 6.5: Gemessene Laufzeit der ersten zwei Phasen.

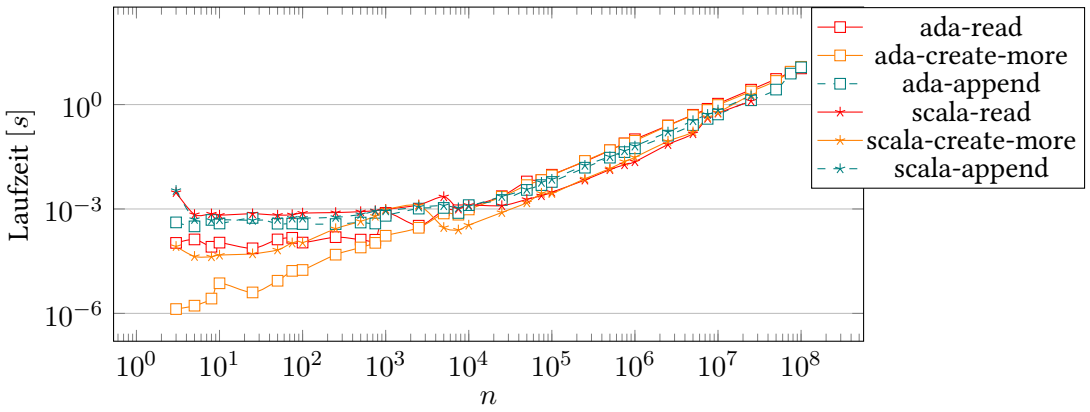

Abbildung 6.6: Gemessene Laufzeit der letzten drei Phasen.

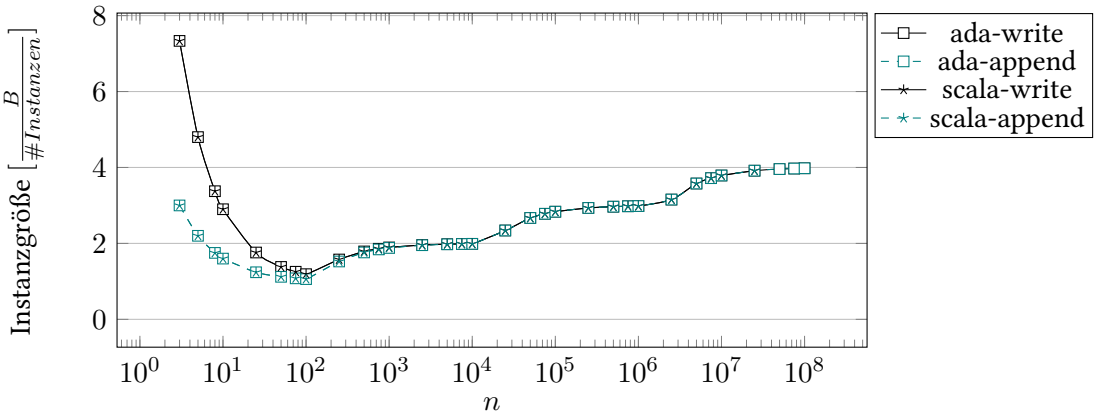

Abbildung 6.7: Gemessener Speicherplatzverbrauch auf der Festplatte pro Date-Instanz in Bytes.

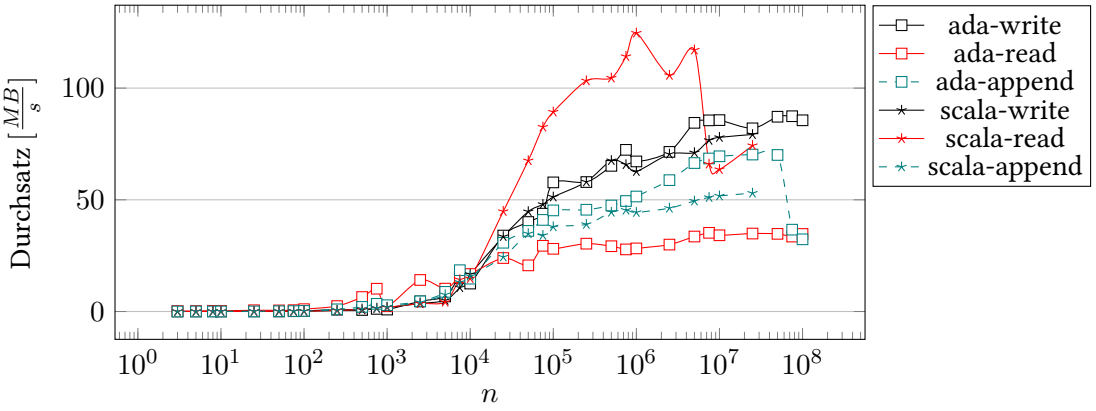

Abbildung 6.8: Gemessener Durchsatz der drei I/O-Phasen in MB/s.

Im dritten Graphen stimmen die Instanzgrößen von beiden Bindings überein. Am Anfang dominieren die Headerdaten und werden schnell vernachlässigbar. In der `append`-Phase müssen bereits bekannte Headerdaten nicht mehr geschrieben werden. Das wirkt sich am Anfang stärker auf die Instanzgröße aus. Im Gegensatz zu den Number-Instanzen im Number-Test ist jede Date-Instanz bis mindestens 10^8 Instanzen immer unter acht Bytes groß. Dieser Graph zeigt die typische Treppe für den Typ v64. Die Funktion $g = 128^n$ bildet für $n = [1, 9[\in \mathbb{N}$ die Wertgrenze zum nächsten Byteverbrauch.

Der vierte Graph zeigt, dass das Ada-Binding beim Lesen vom Scala-Binding dominiert wird. Das Scala-Binding erreicht beim Schreiben, Lesen und Anhängen Spitzengeschwindigkeiten von ca. 80, 120 und 50 MB/s. Das Ada-Binding erreicht Spitzengeschwindigkeiten von ca. 90, 35 und 70 MB/s. Dabei wurde festgestellt, dass beim Ada-Binding noch Verbesserungsbedarf für das Lesen vom Typ v64 besteht. Laut Abbildung 5.11 kann ausgeschlossen werden, dass der v64 Lesealgorithmus (A.3) das Bottleneck ist. Zudem kann laut Abbildung 6.5 durch die `ada-create`-Phase ausgeschlossen werden, dass die Speicherverwaltung das Bottleneck ist.

Im Vergleich zum Number-Test ist beim Scala-Binding eine Auffälligkeit vorhanden. Die Linien im vierten Graphen beim Number-Test verlaufen in ähnlicher Form wie im vierten Graphen beim Date-Test, nur dass es beim Number-Test um den Faktor drei schneller ist.

6.3 Graph 1

Der Test namens Graph 1 ist der erste komplexere Performancetest. Er beinhaltet einen benutzerdefinierten Typ namens *Node* mit vier weiteren Feldern. Die vier weiteren Felder sind die vier Himmelsrichtungen *north*, *east*, *south* und *west* und zeigen jeweils auf einen Node.

Listing 6.4: SKilL-Spezifikation

```
Node {
  Node north;
  Node east;
  Node south;
  Node west;
}
```

Bei diesem Performancetest wird der Zustand mit n Node-Instanzen befüllt. Für die vier Himmelsrichtungen werden pseudozufällige Nodes ausgewählt. Damit dieser Test reproduzierbar ist, wird die Pseudozufälligkeit durch die Hashfunktion MurmurHash3 [AMH] [SMH] realisiert. Jedes Feld hat in den I/O-Phasen `write` und `append` einen unterschiedlichen Seed, sodass jedes Feld auf einen anderen Node zeigen sollte. Der Performancetest konnte mit dem Scala-Binding nur bis $0.25 * 10^8$ Instanzen ausgeführt werden, bevor es zum `java.lang.OutOfMemoryError: Java heap space` kam.

Das Listing 6.4 verwendet normale Typdefinitionen, d. h. dass der benutzerdefinierte Typ sowie die Feldnamen ohne eine Performanceveränderung umbenannt werden können.

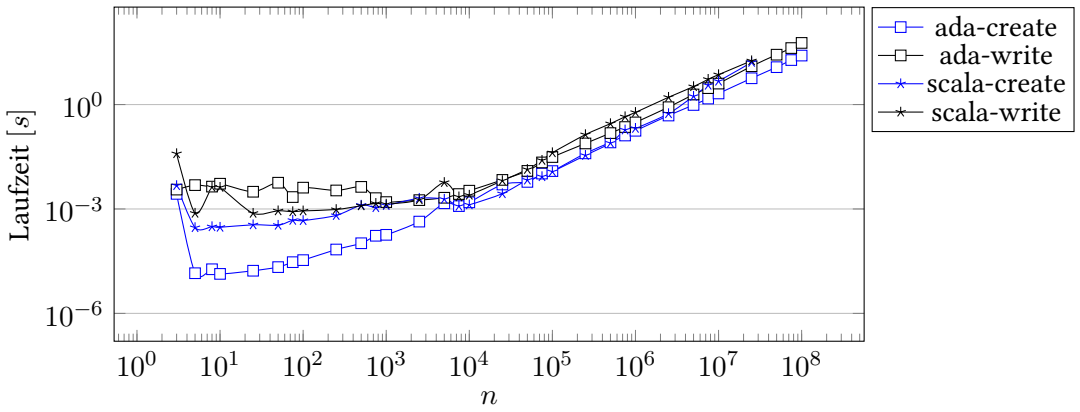

Abbildung 6.9: Gemessene Laufzeit der ersten zwei Phasen.

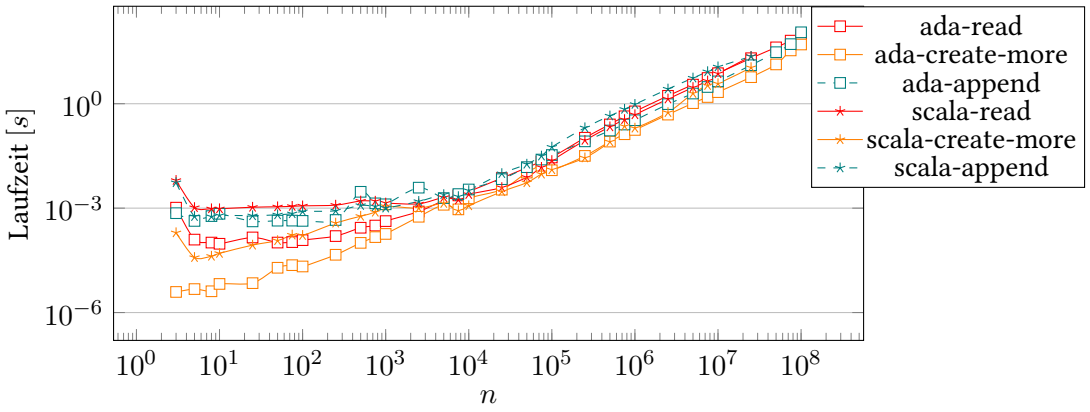

Abbildung 6.10: Gemessene Laufzeit der letzten drei Phasen.

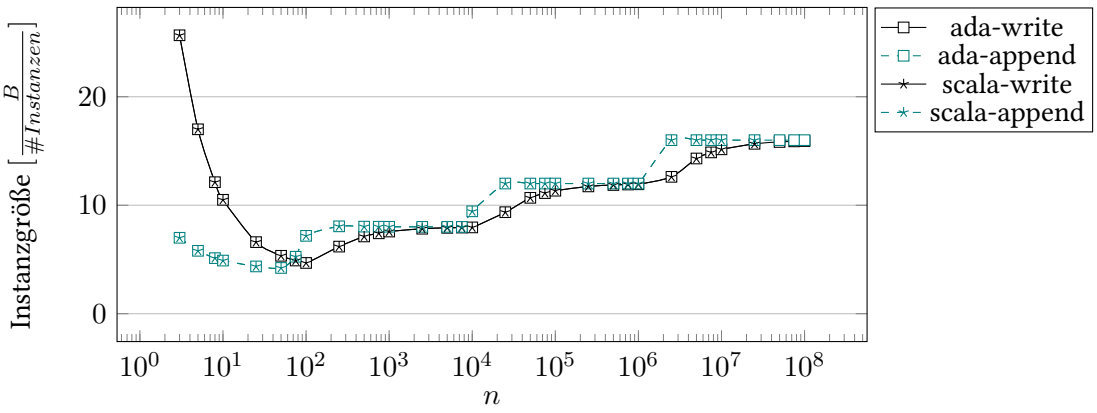

Abbildung 6.11: Gemessener Speicherplatzverbrauch auf der Festplatte pro Node-Instanz in Bytes.

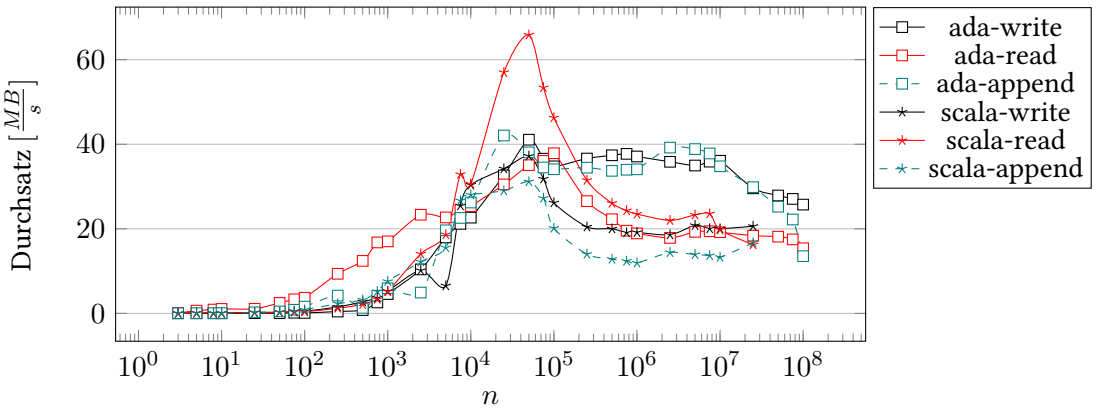

Abbildung 6.12: Gemessener Durchsatz der drei I/O-Phasen in MB/s.

Im dritten Graphen stimmen die Instanzgrößen von beiden Bindings überein. Auch hier dominieren am Anfang die Headerdaten und werden schnell wieder vernachlässigbar.

Der vierte Graph zeigt, dass das Scala-Binding beim Lesen einen kurzen Peak von ca. 65 MB/s bei $n = 0.5 * 10^5$ besitzt. Das Scala-Binding erreicht beim Schreiben, Lesen und Anhängen Spitzengeschwindigkeiten von ca. 40, 65 und 30 MB/s. Das Ada-Binding erreicht Spitzengeschwindigkeiten von ca. 40 MB/s in allen drei I/O-Phasen. Auffällig ist, dass beide Bindings bis $n = 0.5 * 10^5$ gleichwertige Schreib- und Anhängegeschwindigkeiten haben, das Scala-Binding aber danach in allen drei I/O-Phasen auf ca. 20 MB/s abfällt.

6.4 Graph 2

Der Test namens Graph 2 ist der zweite komplexere Performancetest. Er beinhaltet einen benutzerdefinierten Typ namens *Node* mit zwei weiteren Feldern. Das erste Feld heißt *color* und ist vom Typ string. Das zweite Feld heißt *edges* und ist ein Set aus Nodes.

Listing 6.5: SKilL-Spezifikation

```
Node {
  string color;
  set<Node> edges;
}
```

Bei diesem Performancetest wird der Zustand mit n Node-Instanzen befüllt. Das erste Feld wird abwechselnd auf die Farbe *black* oder *red* gesetzt. Das zweite Feld wird pseudozufällig mit $max(50, n)$ Node-Instanzen befüllt. Die Pseudozufälligkeit wird wie in Graph 1 durch die Hashfunktion MurmurHash3 [AMH] [SMH] realisiert. Es hat außerdem in den I/O-Phasen `write` und `append` einen unterschiedlichen Seed, sodass die zwei Sets nicht identisch sein sollten. Der Performancetest konnte nur bis $0.25 * 10^7$ Instanzen ausgeführt werden, bevor der Arbeitsspeicher für das Ada-Binding nicht mehr ausreichte und es beim Scala-Binding zum `java.lang.OutOfMemoryError: Java heap space` kam.

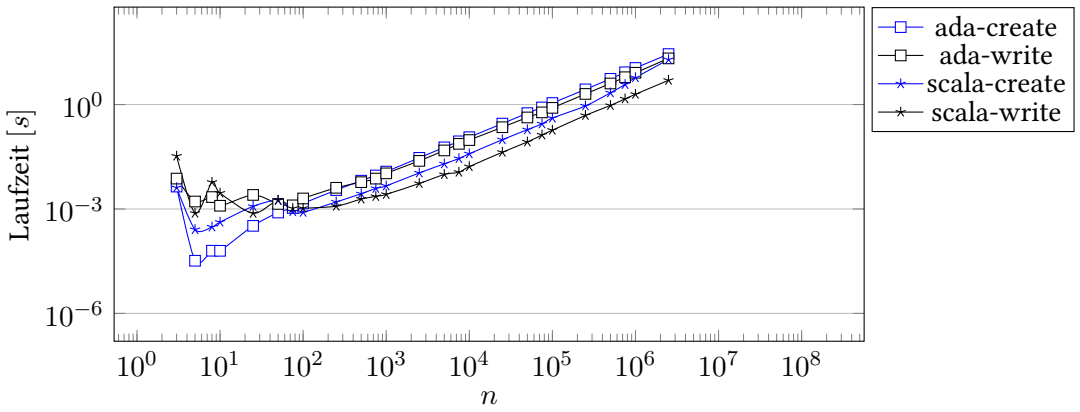

Abbildung 6.13: Gemessene Laufzeit der ersten zwei Phasen.

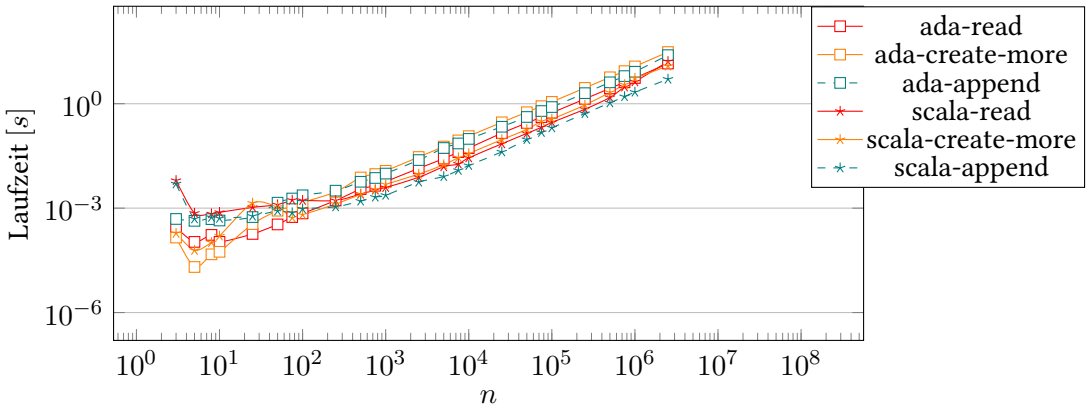

Abbildung 6.14: Gemessene Laufzeit der letzten drei Phasen.

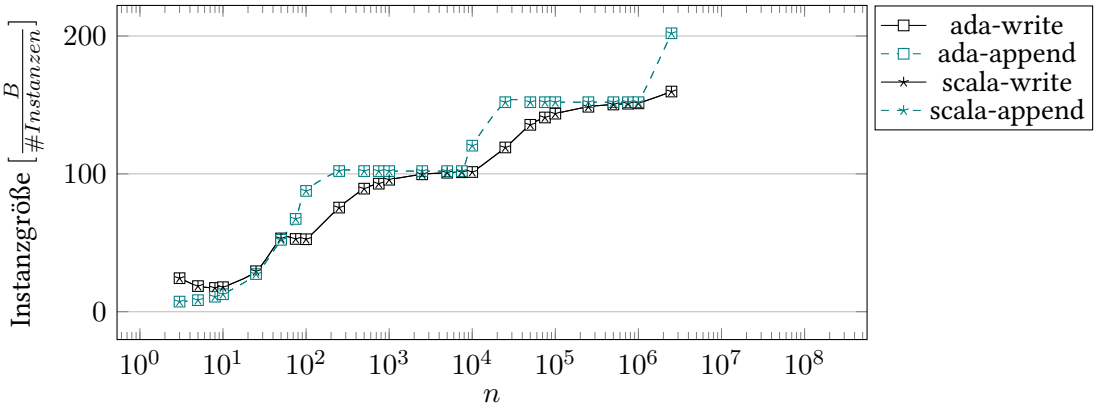

Abbildung 6.15: Gemessener Speicherplatzverbrauch auf der Festplatte pro Node-Instanz in Bytes.

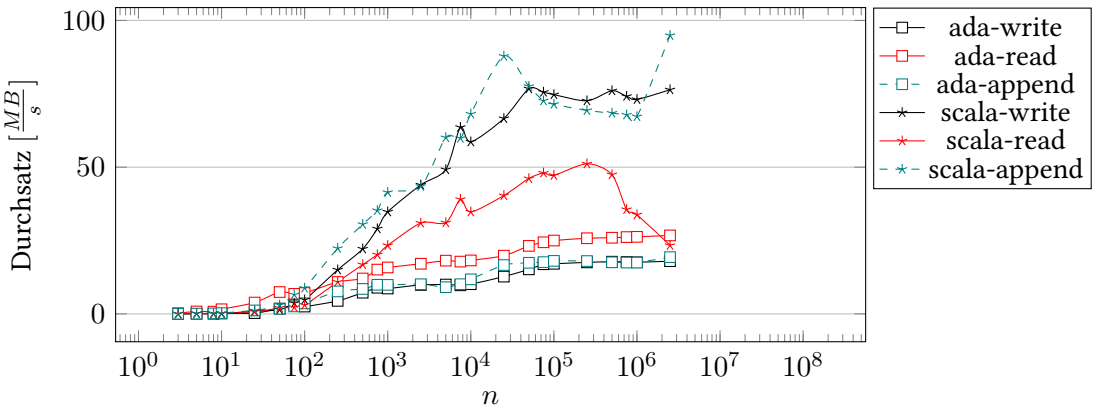

Abbildung 6.16: Gemessener Durchsatz der drei I/O-Phasen in MB/s.

Im dritten Graphen stimmen die Instanzgrößen von beiden Bindings überein. Auffällig ist, dass am Anfang die Headerdaten nicht auffallen und somit sofort vernachlässigbar sind.

Der vierte Graph zeigt, dass das Ada-Binding ein großes Verbesserungsbedürfnis besitzt. Es erreicht beim Schreiben, Lesen und Anhängen Spitzengeschwindigkeiten von ca. 20, 25 und 20 MB/s. Im Gegensatz dazu das Scala-Binding mit Spitzengeschwindigkeiten von ca. 75, 50 und 95 MB/s.

Im Vergleich zum Graph 1-Test sind zwei Auffälligkeiten vorhanden. Erstens hat das Scala-Binding keinen Geschwindigkeitsabfall beim Schreiben und Anhängen, sondern wird sogar immer besser, obwohl der Graph 1-Test simpler ist als der Graph 2-Test mit der zusätzlichen Datenstruktur eines gehashten Sets. Jedoch gibt es am Ende beim Lesen einen Geschwindigkeitsabfall. Zweitens ist das Ada-Binding in allen drei I/O-Phasen gleich langsam. Daher ist davon auszugehen, dass hier das Bottleneck die zusätzliche Datenstruktur eines gehashten Sets ist.

Dieser Test ist auch der erste Performancetest, bei dem das Ada-Binding nicht mehr Instanzen allokieren konnte als das Scala-Binding. Deshalb ist davon auszugehen, dass die Datenstruktur gehashtes Set ineffizient arbeitet und das zu einem höheren Speicherverbrauch führt.

6.5 Zusammenfassung

Zusammenfassend lässt sich aus den ersten zwei Graphen aller Tests hervorheben, dass aufgrund des JVM-Overheads beim Scala-Binding das Ada-Binding bei kleinen Daten und ohne Wiederholung schneller ist.

Beim dritten Graphen aller Tests stimmen die Instanzgrößen beider Bindings überein. Somit ist der Indikator, dass mindestens ein Binding falsch arbeiten könnte, nicht vorhanden. Die Headerdaten spielen bei den Instanzgrößen eine untergeordnete Rolle.

Beim Ada-Binding ist auffallend, dass es bei den Tests Date und Graph 1 im vierten Graphen deutliche Performanceunterschiede zwischen dem Lesen und dem Schreiben oder Anhängen gibt. Die Gemeinsamkeit beider Performancetests ist, dass nur der Integer-Typ v64 verwendet wird.

7 Vergleich mit der Ada-Serialisierung

In diesem Kapitel wird das Ada-Binding mit der in der Programmiersprache Ada vorhandenen sprachabhängigen Serialisierung verglichen.

Die Performance-Evaluation besteht aus den Tests aus Kapitel 6. Jedoch konnten die letzten zwei komplexeren, praxisnahen Performancetests nicht evaluiert werden. In der Ada-Serialisierung werden Referenzen als flache Kopie serialisiert. Ein möglicher Weg für die Serialisierung einer Referenz wäre daher, einen Hashcode als Ersatz für die Referenz zu serialisieren. Bei der Deserialisierung besteht jedoch das Problem, dass auf Instanzen referenziert werden könnten, die noch nicht vorhanden sind. Um dieses Problem zu lösen, müssten entsprechende leere Instanzen vorgehalten werden. Im Rahmen dieser Arbeit konnte dieses Problem in einem angemessenen Zeitraum nicht gelöst werden.

Bei diesen Performancetests wird bereits davon ausgegangen, dass die n Instanzen erstellt wurden und sich in einem Storage Pool eines SKilL-Zustandes befinden. Für die Ada-Serialisierung wurden die zu serialisierenden Instanzen aus dem Storage Pool geholt und geschrieben (siehe Listing 7.1).

Listing 7.1: Das Schreiben der Instanzen bei der Ada-Serialisierung

```
declare
  Instances : T_Type_Accesses := Get_Ts (State);
begin
  for I in Instances'Range loop
    T_Type'Write (Stream, Instances (I).all); -- oder 'Output
  end loop;
end;
```

Jeder Test durchläuft jeweils für das Ada-Binding und die Ada-Serialisierung folgende zwei Phasen:

`write`: In der ersten Phase werden die Instanzen in eine Datei geschrieben. Es wird die gemessene Laufzeit dieser Operation und die Dateigröße protokolliert.

`read`: In der zweiten Phase wird die zuvor geschriebene Datei eingelesen. Es wird die gemessene Laufzeit dieser Operation protokolliert.

Jeder Test wurde zehn mal wiederholt. Die Anzahl der Wiederholungen waren aufgrund der Verfügbarkeit der Testhardware limitiert. Es wurde der Durchschnitt aus den elf Runden gebildet.

Listing 7.2: Ablauf des Benchmarks

```
for R in 0 .. Repetitions loop -- Ein obligatorischer Durchlauf und die Wiederholungen
  for E in 1 .. Exponent loop
    for P of Phases loop -- Die zwei Phasen: write, read
      Measure ((10 ** E) * 0.25, P);
      Measure ((10 ** E) * 0.50, P);
```

```
      Measure ((10 ** E) * 0.75, P);
      Measure ((10 ** E) * 1.00, P);
    end loop;
  end loop;
end loop;

-- Bilde die Durchschnitte aus den gemessenen Werten
-- Generiere die Graphen
```

Aus den protokollierten Daten wurden drei Graphen generiert. Der erste Graph visualisiert die gemessene Laufzeit in s aller zwei Phasen für jedes n. Der zweite Graph visualisiert die durchschnittlichen geschriebenen Bytes pro Instanz für jedes n. Zu beachten ist, dass die Ada-Serialisierung bei den beiden simplen Performancetests die gleiche Instanzgröße haben müssen, weil die Integer-Typen i64 und v64 als Long repräsentiert werden. Der dritte Graph visualisiert den Durchsatz der I/O-Phasen `write` und `read` für jedes n. Dabei wird die Dateigröße in MB durch die gemessene Laufzeit in s geteilt.

Es wurde immer direkt vor und nach der zu messenden Operation gemessen.

7.1 Number

Der Test namens Number ist der erste simple Performancetest. Er beinhaltet nur einen benutzerdefinierten Typ namens *Number* mit einem Feld namens *number* vom Typ i64.

Listing 7.3: SKilL-Spezifikation

```
Number {
  i64 number;
}
```

Bei diesem Performancetest wird der Zustand mit n Number-Instanzen befüllt. Das i64-Feld wird mit der Zahl Eins aufsteigend in Einer-Schritten befüllt.

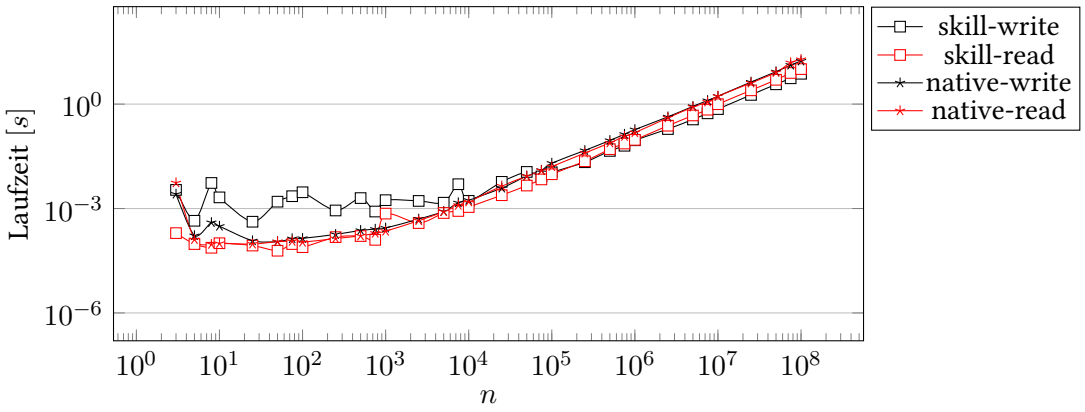

Abbildung 7.1: Gemessene Laufzeit der zwei I/O-Phasen.

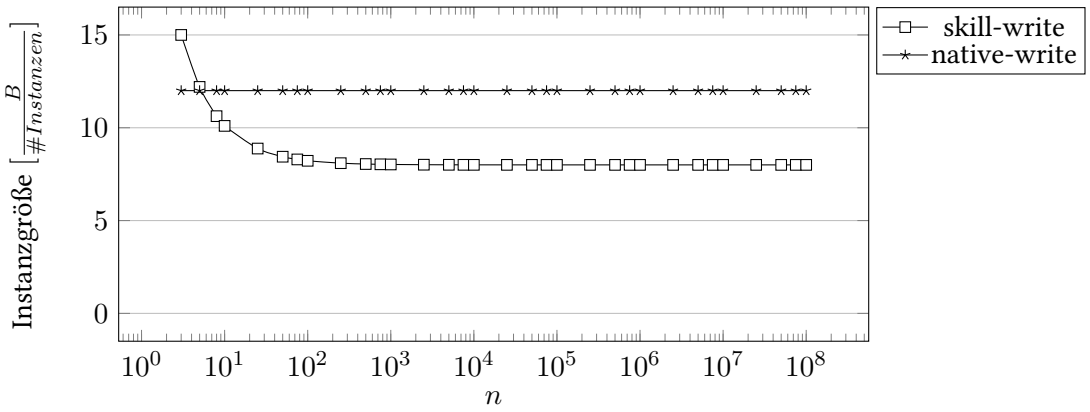

Abbildung 7.2: Gemessener Speicherplatzverbrauch auf der Festplatte pro Number-Instanz in Bytes.

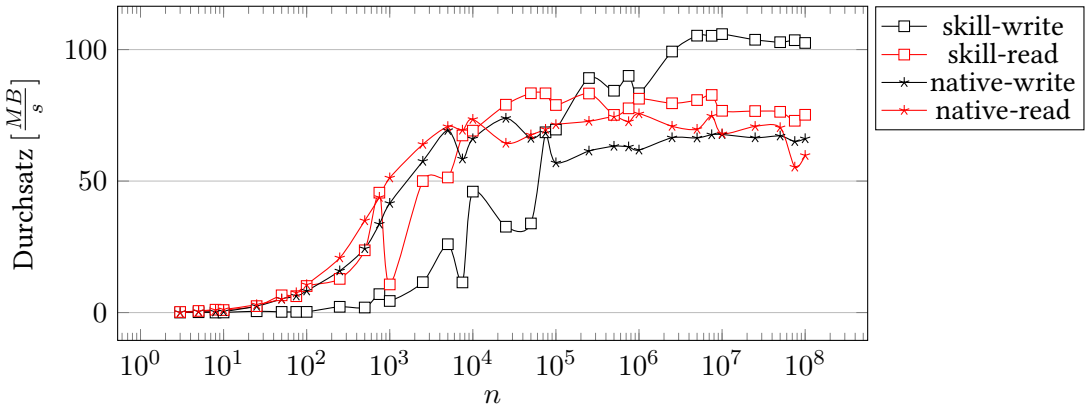

Abbildung 7.3: Gemessener Durchsatz der zwei I/O-Phasen in MB/s.

Der zweite Graph zeigt, dass für jedes n in der Ada-Serialisierung die Instanzgröße immer bei zwölf Bytes liegt. Nach der Analyse einer geschriebenen Datei aus der Ada-Serialisierung werden vier Bytes für die Stelle und acht Bytes für den Wert benötigt. Beim Ada-Binding dominieren kurz die Headerdaten und werden schnell wieder vernachlässigbar.

Der dritte Graph zeigt, dass die Ada-Serialisierung beim Schreiben und Lesen früh die Spitzengeschwindigkeiten von ca. 70 und 75 MB/s erreicht. Beim Ada-Binding gibt es immer wieder kleine Geschwindigkeitseinbußen, erreicht jedoch Spitzengeschwindigkeiten von ca. 105 und 75 MB/s.

7.2 Date

Der Test namens Date ist der zweite simple Performancetest. Er beinhaltet nur einen benutzerdefinierten Typ namens *Date* mit einem Feld namens *date* vom Typ v64. Der Unterschied zum Performancetest Number ist, dass hier ein Integer-Typ variabler Länge genutzt wird.

Listing 7.4: SKilL-Spezifikation

```
Date {
  v64 date;
}
```

Bei diesem Performancetest wird der Zustand mit n Date-Instanzen befüllt. Das v64-Feld wird mit der Zahl Eins aufsteigend in Einer-Schritten befüllt.

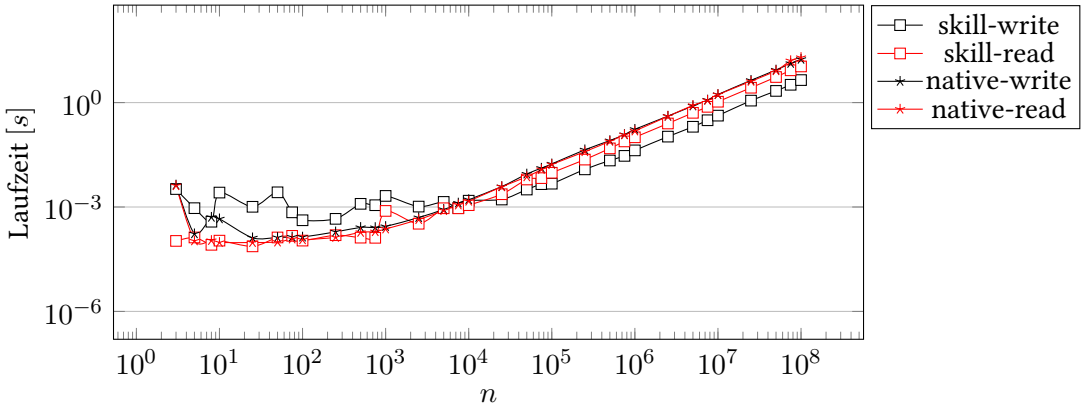

Abbildung 7.4: Gemessene Laufzeit der zwei I/O-Phasen.

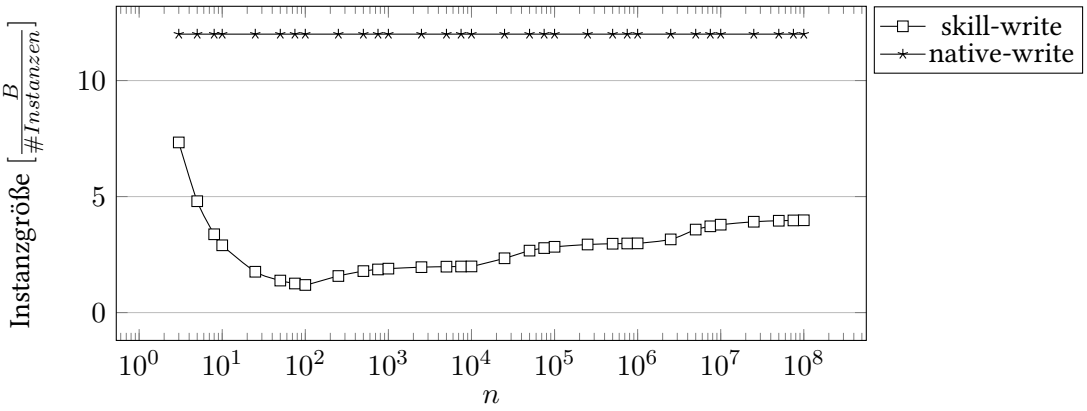

Abbildung 7.5: Gemessener Speicherplatzverbrauch auf der Festplatte pro Date-Instanz in Bytes.

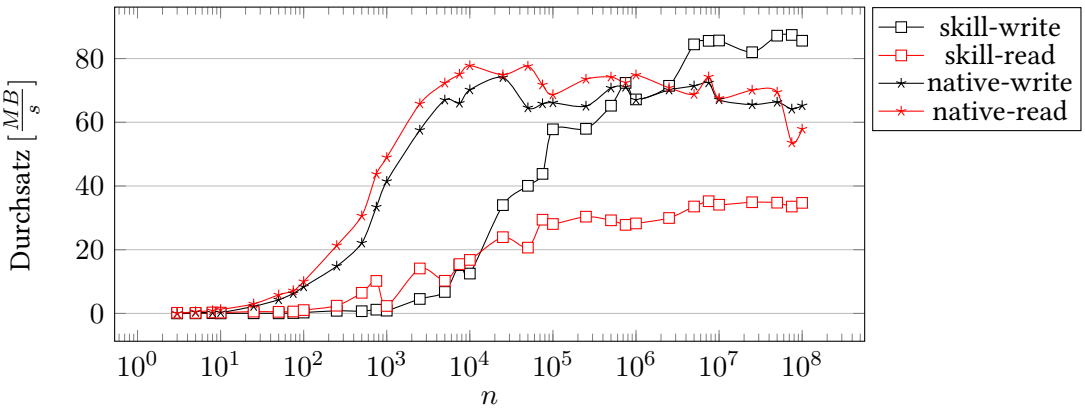

Abbildung 7.6: Gemessener Durchsatz der zwei I/O-Phasen in MB/s.

Die Ada-Serialisierung unterscheidet nicht zwischen den Typen i64 und v64. Daher zeigt der zweite Graph für die Ada-Serialisierung wieder zwölf Bytes pro Instanz. Beim Ada-Binding gibt es die gewohnte Treppe.

Der dritte Graph zeigt, dass die Ada-Serialisierung beim Schreiben und Lesen früh die Spitzengeschwindigkeiten von ca. 70 und 75 MB/s erreicht. Diese Werte waren ähnlich zum Number-Test und sogar der kleine Geschwindigkeitsabfall am Ende ist beim Lesen vorhanden. Jedoch ist für das Ada-Binding auch hier das Lesen vom Typ v64 nicht zufriedenstellend wie schon zuvor beim Date-Test aus Kapitel 6 festgestellt wurde. Das Ada-Binding erreicht Spitzengeschwindigkeiten von ca. 85 und 35 MB/s.

7.3 Zusammenfassung

Zusammenfassend lässt sich feststellen, dass das Ada-Binding meist mit der Ada-Serialisierung mithalten kann. Jedoch lässt sich auch hier wie schon zuvor in Kapitel 6 feststellen, dass das Ada-Binding für das Lesen vom Typ v64 Verbesserungspotenzial besitzt. Im Übrigen hat das Ada-Binding ähnliche Spitzengeschwindigkeiten wie in Kapitel 6 erreicht. Die Ada-Serialisierung hat in beiden Performancetests, wie zu erwarten war, ähnliche Spitzengeschwindigkeiten erreicht.

Die Ada-Serialisierung hat das Problem, dass sie mehr Speicherplatz auf der Festplatte verbraucht als das Ada-Binding und dabei trotzdem nicht typsicher ist. Liest man z. B. beim Number-Test die erste vom Benchmark generierte Datei mit den drei Number-Instanzen das Feld *number* als Typ i16 anstatt i64 ein, so erhält man die Number-Instanzen 1, 0, 2, 0, 3 und 0 zurück. Außerdem muss die Ada-Serialisierung (unintuitiverweise) manuell erweitert werden, um Referenzen korrekt serialisieren zu können.

8 Verbesserungsvorschlag für das Serialisierungsformat

Im Serialisierungsformat fallen beim Anhängen von bereits geschriebenen Typdeklarationen die Felder, die Informationen zum Supertyp und den Restriktionen halten, weg. Jedoch liegt das Feld *Anzahl der Instanzen* dazwischen und es muss beim Lesen und Schreiben unnötigerweise eine if-Abfrage aufgesplittet werden (siehe Listing 8.1). Durch das Tauschen der Felder *Anzahl der Instanzen* und *Restriktionen* in der Typdeklaration kann das Aufsplitten der if-Abfrage vermieden werden (siehe Listing 8.2).

Listing 8.1: Das Lesen einer Typdeklaration im Ist-Zustand

```
Lese Name
Falls noch unbekannt:
  Lese Supertyp
Lese Anzahl der Instanzen
Falls noch unbekannt:
  Lese Restriktionen
```

Listing 8.2: Das Lesen einer Typdeklaration im Kann-Zustand

```
Lese Name
Falls noch unbekannt:
  Lese Supertyp
  Lese Restriktionen
Lese Anzahl der Instanzen
```

9 Zusammenfassung und Ausblick

Um ein gutes Binding für die Serialisierungssprache SKilL zu erhalten, benötigt man für ähnliche Programmiersprachen wie Ada Arrays mit schnellen `append`- und `get`-Operationen sowie Hashmaps mit schnellen `get`-Operationen.

Soll das Binding auf Performance ausgelegt und die notwendige Zeit zum Testen vorhanden sein, sollte man vorerst nur den Integer-Typ v64 aus der Kernsprache implementieren und dann versuchen, die gewünschten Performancekriterien zu erreichen. Nachdem dieses Ziel erreicht wurde, können die restlichen Punkte aus der Kernsprache mit einem guten Gewissen implementiert werden. Es ist empfehlenswert, neben den Unittests auch regelmäßig die Performance durch eine kontinuierliche Integration sicherzustellen.

Das Ada-Binding hat das Problem, dass durch die Nutzung des SKilL-Zustandes einiges an Performance verloren geht im Vergleich zu der isolierten Betrachtung der Byte-Pakete, die eine zufriedenstellende Performance vorweisen können (siehe Abbildung 5.11, Abbildung 6.8 und Abbildung 7.6).

Ein weiteres Problem ist, dass möglicherweise die Datenstrukturen in der Programmiersprache Ada zu starken Performanceeinbußen führen. Ein Hinweis dafür könnte z. B. die Abbildung 6.16 sein. Im Graph 2-Test wird die Datenstruktur gehashtes Set verwendet und in allen drei I/O-Phasen wird ein ähnlich langsamer Durchsatz erreicht. Es sollte evaluiert werden, ob die Performanceeinbußen bei allen benötigten Datenstrukturen vorhanden sind. Bei einem positiven Ergebnis sollten die benötigten Datenstrukturen mit definierten Performancekriterien nochmals implementiert werden.

Beim Vergleich mit der Ada-Serialisierung gab es das Problem, dass die zwei komplexeren, praxisnahen Performancetests Graph 1 und Graph 2 aufgrund der Referenzen nicht evaluiert werden konnten. Dieses Problem sollte noch gelöst werden. Bei Erfolg sollten die noch fehlenden Performancetests durchgeführt werden. Zudem sollte evaluiert werden, ob die Ada-Serialisierung und die Serialisierungssprache SKilL sinnvoll fusioniert werden können.

Des Weiteren könnte, falls eine externe GPL-Bibliothek verwendet werden darf, evaluiert werden, ob die Ersetzung von *Ada.Streams* durch *GNATCOLL.Mmap* [Ada13c, § 6] im Paket `Byte_Reader` zu einem Performancegewinn führen würde.

Zudem wäre es interessant, wie sich die Bindings gegen verschiedene XML-Bibliotheken mit semantisch äquivalenten Daten in Bezug auf die Dateigröße und den Durchsatz behaupten.

A Anhang

Listing A.1: byte Lesealgorithmus

```
function Read_Byte (Stream : ASS_IO.Stream_Access) return Byte is
begin
  if Buffer_Size = Buffer_Index then
    Buffer'Read (Stream, Buffer_Array);
    Buffer_Index := 0;
  end if;

  Buffer_Index := Buffer_Index + 1;

  declare
    Next : Byte := Buffer_Array (Buffer_Index);
  begin
    return Next;
  end;
end Read_Byte;
```

Listing A.2: i64 Lesealgorithmus

```
function Read_i64 (Stream : ASS_IO.Stream_Access) return i64 is
  A : i64 := i64 (Read_Byte (Stream));
  B : i64 := i64 (Read_Byte (Stream));
  C : i64 := i64 (Read_Byte (Stream));
  D : i64 := i64 (Read_Byte (Stream));
  E : i64 := i64 (Read_Byte (Stream));
  F : i64 := i64 (Read_Byte (Stream));
  G : i64 := i64 (Read_Byte (Stream));
  H : i64 := i64 (Read_Byte (Stream));
begin
  return A * (2 ** 56) +
         B * (2 ** 48) +
         C * (2 ** 40) +
         D * (2 ** 32) +
         E * (2 ** 24) +
         F * (2 ** 16) +
         G * (2 ** 8) +
         H;
end Read_i64;
```

Listing A.3: v64 Lesealgorithmus (Version aus dem Anhang des Technischen Berichts 2013/06)

```
function Read_v64 (Stream : ASS_IO.Stream_Access) return v64 is
  use Interfaces;
```

```
   function Convert is new Ada.Unchecked_Conversion (Unsigned_64, v64);

   Count : Natural := 0;
   Return_Value : Unsigned_64 := 0;
   Bucket : Unsigned_64 := Unsigned_64 (Read_Byte (Stream));
begin
   while (Count < 8 and then 0 /= (Bucket and 16#80#)) loop
      Return_Value := Return_Value or ((Bucket and 16#7f#) * (2 ** (7 * Count)));
      Count := Count + 1;
      Bucket := Unsigned_64 (Read_Byte (Stream));
   end loop;

   case Count is
      when 8 => Return_Value := Return_Value or (Bucket * (2 ** (7 * Count)));
      when others => Return_Value := Return_Value or ((Bucket and 16#7f#) * (2 ** (7 * Count)));
   end case;

   return Convert (Return_Value);
end Read_v64;
```

Listing A.4: byte Schreibalgorithmus

```
procedure Write_Byte (Stream : ASS_IO.Stream_Access; Next : Byte) is
begin
   Buffer_Index := Buffer_Index + 1;
   Buffer_Array (Buffer_Index) := Next;

   if Buffer_Size = Buffer_Index then
      Finalize_Buffer (Stream);
   end if;
end Write_Byte;
```

Listing A.5: i64 Schreibalgorithmus

```
procedure Write_i64 (Stream : ASS_IO.Stream_Access; Value : i64) is
   use Interfaces;

   function Convert is new Ada.Unchecked_Conversion (i64, Unsigned_64);

   A : Unsigned_64 := (Convert (Value) / (2 ** 56)) and 16#ff#;
   B : Unsigned_64 := (Convert (Value) / (2 ** 48)) and 16#ff#;
   C : Unsigned_64 := (Convert (Value) / (2 ** 40)) and 16#ff#;
   D : Unsigned_64 := (Convert (Value) / (2 ** 32)) and 16#ff#;
   E : Unsigned_64 := (Convert (Value) / (2 ** 24)) and 16#ff#;
   F : Unsigned_64 := (Convert (Value) / (2 ** 16)) and 16#ff#;
   G : Unsigned_64 := (Convert (Value) / (2 ** 8)) and 16#ff#;
   H : Unsigned_64 := Convert (Value) and 16#ff#;
begin
   Write_Byte (Stream, Byte (A));
   Write_Byte (Stream, Byte (B));
   Write_Byte (Stream, Byte (C));
   Write_Byte (Stream, Byte (D));
   Write_Byte (Stream, Byte (E));
   Write_Byte (Stream, Byte (F));
   Write_Byte (Stream, Byte (G));
```

```
   Write_Byte (Stream, Byte (H));
end Write_i64;
```

Listing A.6: v64 Schreibalgorithmus (Version aus dem Anhang des Technischen Berichts 2013/06)

```
procedure Write_v64 (Stream : ASS_IO.Stream_Access; Value : v64) is
   type Byte_v64_Type is array (Natural range <>) of Byte;

   function Get_v64_Bytes (Value : v64) return Byte_v64_Type is
      use Interfaces;

      function Convert is new Ada.Unchecked_Conversion (v64, Unsigned_64);

      Size : Natural := 0;
   begin
      declare
         Buckets : Unsigned_64 := Convert (Value);
      begin
         while (Buckets > 0) loop
            Buckets := Buckets / (2 ** 7);
            Size := Size + 1;
         end loop;
      end;

      case Size is
         when 0 => return (0 => 0);
         when 10 => Size := 9;
         when others => null;
      end case;

      declare
         Return_Value : Byte_v64_Type (0 .. Size - 1);
         Count : Natural := 0;
      begin
         while (Count < 8 and then Count < Size - 1) loop
            Return_Value (Count) := Byte (((Convert (Value) / (2 ** (7 * Count))) or 16#80#) and
               16#ff#);
            Count := Count + 1;
         end loop;
         Return_Value (Count) := Byte ((Convert (Value) / (2 ** (7 * Count))) and 16#ff#);
         return Return_Value;
      end;
   end Get_v64_Bytes;
   pragma Inline (Get_v64_Bytes);

   Return_Value : Byte_v64_Type := Get_v64_Bytes (Value);
begin
   for I in Return_Value'Range loop
      Write_Byte (Stream, Return_Value (I));
   end loop;
end Write_v64;
```

Listing A.7: v64 Schreibalgorithmus (Version aus dem Scala-Binding)

```
procedure Write_v64 (Stream : ASS_IO.Stream_Access; Value : v64) is
```

```
   use Interfaces;

   function Convert is new Ada.Unchecked_Conversion (v64, Unsigned_64);
begin
   if Convert (Value) < 128 then
      declare
         A : Unsigned_64 := Convert (Value) and 16#ff#;
      begin
         Write_Byte (Stream, Byte (A));
      end;
   elsif Convert (Value) < (128 ** 2) then
      declare
         A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
         B : Unsigned_64 := (Convert (Value) / (2 ** 7)) and 16#ff#;
      begin
         Write_Byte (Stream, Byte (A));
         Write_Byte (Stream, Byte (B));
      end;
   elsif Convert (Value) < (128 ** 3) then
      declare
         A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
         B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
         C : Unsigned_64 := (Convert (Value) / (2 ** 14)) and 16#ff#;
      begin
         Write_Byte (Stream, Byte (A));
         Write_Byte (Stream, Byte (B));
         Write_Byte (Stream, Byte (C));
      end;
   elsif Convert (Value) < (128 ** 4) then
      declare
         A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
         B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
         C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
         D : Unsigned_64 := (Convert (Value) / (2 ** 21)) and 16#ff#;
      begin
         Write_Byte (Stream, Byte (A));
         Write_Byte (Stream, Byte (B));
         Write_Byte (Stream, Byte (C));
         Write_Byte (Stream, Byte (D));
      end;
   elsif Convert (Value) < (128 ** 5) then
      declare
         A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
         B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
         C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
         D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
         E : Unsigned_64 := (Convert (Value) / (2 ** 28)) and 16#ff#;
      begin
         Write_Byte (Stream, Byte (A));
         Write_Byte (Stream, Byte (B));
         Write_Byte (Stream, Byte (C));
         Write_Byte (Stream, Byte (D));
         Write_Byte (Stream, Byte (E));
      end;
   elsif Convert (Value) < (128 ** 6) then
```

```
   declare
     A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
     B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
     C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
     D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
     E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
     F : Unsigned_64 := (Convert (Value) / (2 ** 35)) and 16#ff#;
   begin
     Write_Byte (Stream, Byte (A));
     Write_Byte (Stream, Byte (B));
     Write_Byte (Stream, Byte (C));
     Write_Byte (Stream, Byte (D));
     Write_Byte (Stream, Byte (E));
     Write_Byte (Stream, Byte (F));
   end;
elsif Convert (Value) < (128 ** 7) then
   declare
     A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
     B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
     C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
     D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
     E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
     F : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 35))) and 16#ff#;
     G : Unsigned_64 := (Convert (Value) / (2 ** 42)) and 16#ff#;
   begin
     Write_Byte (Stream, Byte (A));
     Write_Byte (Stream, Byte (B));
     Write_Byte (Stream, Byte (C));
     Write_Byte (Stream, Byte (D));
     Write_Byte (Stream, Byte (E));
     Write_Byte (Stream, Byte (F));
     Write_Byte (Stream, Byte (G));
   end;
elsif Convert (Value) < (128 ** 8) then
   declare
     A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
     B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
     C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
     D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
     E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
     F : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 35))) and 16#ff#;
     G : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 42))) and 16#ff#;
     H : Unsigned_64 := (Convert (Value) / (2 ** 49)) and 16#ff#;
   begin
     Write_Byte (Stream, Byte (A));
     Write_Byte (Stream, Byte (B));
     Write_Byte (Stream, Byte (C));
     Write_Byte (Stream, Byte (D));
     Write_Byte (Stream, Byte (E));
     Write_Byte (Stream, Byte (F));
     Write_Byte (Stream, Byte (G));
     Write_Byte (Stream, Byte (H));
   end;
else
   declare
```

```
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
      D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
      E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
      F : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 35))) and 16#ff#;
      G : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 42))) and 16#ff#;
      H : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 49))) and 16#ff#;
      I : Unsigned_64 := (Convert (Value) / (2 ** 56)) and 16#ff#;
    begin
      Write_Byte (Stream, Byte (A));
      Write_Byte (Stream, Byte (B));
      Write_Byte (Stream, Byte (C));
      Write_Byte (Stream, Byte (D));
      Write_Byte (Stream, Byte (E));
      Write_Byte (Stream, Byte (F));
      Write_Byte (Stream, Byte (G));
      Write_Byte (Stream, Byte (H));
      Write_Byte (Stream, Byte (I));
    end;
  end if;
end Write_v64;
```

Listing A.8: v64 Schreibalgorithmus (Version aus dem Scala-Binding mit der BSR-Modifikation)

```
procedure Write_v64 (Stream : ASS_IO.Stream_Access; Value : v64) is
  use Interfaces;

  function Convert is new Ada.Unchecked_Conversion (v64, Unsigned_64);

  Result : Long range 0 .. 63;
  Index : Long range 0 .. 9;
begin
  System.Machine_Code.Asm ("bsr %1, %0",
    Outputs => Long'Asm_Output ("=a", Result),
    Inputs => Long'Asm_Input ("a", Value)
  );

  Index := Result / 7;

  case Index is
    when 0 =>
      declare
        A : Unsigned_64 := Convert (Value) and 16#ff#;
      begin
        Write_Byte (Stream, Byte (A));
      end;

    when 1 =>
      declare
        A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
        B : Unsigned_64 := (Convert (Value) / (2 ** 7)) and 16#ff#;
      begin
        Write_Byte (Stream, Byte (A));
        Write_Byte (Stream, Byte (B));
```

```
   end;

when 2 =>
   declare
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (Convert (Value) / (2 ** 14)) and 16#ff#;
   begin
      Write_Byte (Stream, Byte (A));
      Write_Byte (Stream, Byte (B));
      Write_Byte (Stream, Byte (C));
   end;

when 3 =>
   declare
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
      D : Unsigned_64 := (Convert (Value) / (2 ** 21)) and 16#ff#;
   begin
      Write_Byte (Stream, Byte (A));
      Write_Byte (Stream, Byte (B));
      Write_Byte (Stream, Byte (C));
      Write_Byte (Stream, Byte (D));
   end;

when 4 =>
   declare
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
      D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
      E : Unsigned_64 := (Convert (Value) / (2 ** 28)) and 16#ff#;
   begin
      Write_Byte (Stream, Byte (A));
      Write_Byte (Stream, Byte (B));
      Write_Byte (Stream, Byte (C));
      Write_Byte (Stream, Byte (D));
      Write_Byte (Stream, Byte (E));
   end;

when 5 =>
   declare
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
      D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
      E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
      F : Unsigned_64 := (Convert (Value) / (2 ** 35)) and 16#ff#;
   begin
      Write_Byte (Stream, Byte (A));
      Write_Byte (Stream, Byte (B));
      Write_Byte (Stream, Byte (C));
      Write_Byte (Stream, Byte (D));
      Write_Byte (Stream, Byte (E));
```

```
      Write_Byte (Stream, Byte (F));
    end;

  when 6 =>
    declare
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
      D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
      E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
      F : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 35))) and 16#ff#;
      G : Unsigned_64 := (Convert (Value) / (2 ** 42)) and 16#ff#;
    begin
      Write_Byte (Stream, Byte (A));
      Write_Byte (Stream, Byte (B));
      Write_Byte (Stream, Byte (C));
      Write_Byte (Stream, Byte (D));
      Write_Byte (Stream, Byte (E));
      Write_Byte (Stream, Byte (F));
      Write_Byte (Stream, Byte (G));
    end;


  when 7 =>
    declare
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
      D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
      E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
      F : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 35))) and 16#ff#;
      G : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 42))) and 16#ff#;
      H : Unsigned_64 := (Convert (Value) / (2 ** 49)) and 16#ff#;
    begin
      Write_Byte (Stream, Byte (A));
      Write_Byte (Stream, Byte (B));
      Write_Byte (Stream, Byte (C));
      Write_Byte (Stream, Byte (D));
      Write_Byte (Stream, Byte (E));
      Write_Byte (Stream, Byte (F));
      Write_Byte (Stream, Byte (G));
      Write_Byte (Stream, Byte (H));
    end;

  when 8 | 9 =>
    declare
      A : Unsigned_64 := (16#80# or Convert (Value)) and 16#ff#;
      B : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 7))) and 16#ff#;
      C : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 14))) and 16#ff#;
      D : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 21))) and 16#ff#;
      E : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 28))) and 16#ff#;
      F : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 35))) and 16#ff#;
      G : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 42))) and 16#ff#;
      H : Unsigned_64 := (16#80# or (Convert (Value) / (2 ** 49))) and 16#ff#;
      I : Unsigned_64 := (Convert (Value) / (2 ** 56)) and 16#ff#;
```

```
         begin
           Write_Byte (Stream, Byte (A));
           Write_Byte (Stream, Byte (B));
           Write_Byte (Stream, Byte (C));
           Write_Byte (Stream, Byte (D));
           Write_Byte (Stream, Byte (E));
           Write_Byte (Stream, Byte (F));
           Write_Byte (Stream, Byte (G));
           Write_Byte (Stream, Byte (H));
           Write_Byte (Stream, Byte (I));
         end;
      end case;
   end Write_v64;
```

Abkürzungsverzeichnis

AFD Advanced Format Drives. 22

BSR Bit Scan Reverse. 24

CPU Central Processing Unit. 15

GC Garbage Collector. 7, 25, 30

GPL General Public License. 47

I/O Input/Output. 30, 31, 33, 34, 36, 38, 40–43, 47

IEEE Institute of Electrical and Electronics Engineers. 10

JIT Just-In-Time. 7

JVM Java Virtual Machine. 7, 16, 30, 38

RAM Random-Access Memory. 15

SKiIL Serialization Killer Language. 3, 7, 9, 11–13, 17–21, 27, 39, 47

SSD Solid-State-Drive. 15

VM Virtual Machine. 16

XML Extensible Markup Language. 13

Literaturverzeichnis

[Ada13a] AdaCore. *GNAT GPL User's Guide*, 2013. The GNAT Ada Compiler, GNAT GPL Edition, Document revision level 291686. (Zitiert auf Seite 15)

[Ada13b] AdaCore. *GNAT Reference Manual*, 2013. GNAT, The GNU Ada Compiler, GNAT GPL Edition, Document revision level 291686. (Zitiert auf Seite 15)

[Ada13c] AdaCore. *GNATColl Documentation*, 2013. Release 1.6w. (Zitiert auf Seite 47)

[AMH] Implementation of MurmurHash3 in Ada. URL http://commons.ada.cx/Deterministic_Hashing. (Zitiert auf den Seiten 34 und 36)

[Fel13] T. Felden. The SKilL Language. Technischer Bericht Informatik 2013/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Deutschland, 2013. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2013-06&engl=0. (Zitiert auf den Seiten 7, 9, 10, 11, 12, 13 und 24)

[Fel14] T. Felden. Efficient and Change-Tolerant Serialization for Program Analysis Tool-Chains. In *16. Workshop Software-Reengineering und -Evolution (WSR'14)*. Bad Honnef, Germany, 2014. (Zitiert auf Seite 29)

[GJS+13] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley. The Java Language Specification, Java SE 7 Edition, 2013. URL http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf. (Zitiert auf Seite 10)

[IEE08] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, S. 1–70, 2008. doi:10.1109/IEEESTD.2008.4610935. URL http://ieeexplore.ieee.org/servlet/opac?punumber=4610933. (Zitiert auf Seite 10)

[Int11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2011. URL http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf. Volume 2A, Instruction Set Reference, A-M. (Zitiert auf Seite 24)

[Kos] T. Koskinen. Ahven - Unit Testing Library for Ada Programming Language. URL http://ahven.stronglytyped.org. (Zitiert auf Seite 27)

[Ode11] M. Odersky. *The Scala Language Specification*, 2011. URL http://www.scala-lang.org/old/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf. (Zitiert auf Seite 7)

[Ora14] Oracle. *Java Platform Standard Edition 7 Documentation*, 2014. Java Virtual Machine Technology. (Zitiert auf den Seiten 7 und 16)

[RVP06] A. Raza, G. Vogel, E. Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Reliable Software Technologies – Ada-Europe 2006*, Band 4006 von *Lecture Notes in Computer Science*, S. 71–82. Springer Berlin Heidelberg, 2006. (Zitiert auf Seite 7)

[Sea] Seagate. Transition to Advanced Format 4K Sector Hard Drives. URL http://www.seagate.com/tech-insights/advanced-format-4k-sector-hard-drives-master-ti. (Zitiert auf Seite 22)

[SGDC13] R. M. Stallman, the GCC Developer Community. *Using the GNU Compiler Collection*, 2013. For GCC version 4.7.4. (Zitiert auf Seite 15)

[SKI] SKilL on Github. URL https://github.com/skill-lang/skill. (Zitiert auf den Seiten 13, 24 und 27)

[SMH] Implementation of MurmurHash3 in Scala. URL http://www.scala-lang.org/files/archive/nightly/docs/library/index.html#scala.util.hashing.MurmurHash3$. (Zitiert auf den Seiten 34 und 36)

[TDB+12] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, P. Leroy. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652:2012(E)*, 2012. URL http://www.ada-auth.org/standards/12rm/RM-Final.pdf. (Zitiert auf den Seiten 7 und 21)

Alle URLs wurden zuletzt am 16. 05. 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift