Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Kompression von Felddaten in SKilL

Dominik Müller

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Dr. rer. nat. Timm Felden
Beginn am:	4. Juni 2018
Beendet am:	4. Dezember 2018

Kurzfassung

Die Sprache SKilL erlaubt es, Datenformate zu beschreiben, die sich zur plattform- und sprachunabhängigen Serialisierung eignen und auf große Datenmengen ausgelegt sind. Durch eine Anbindung an eine Programmiersprache wie Java oder C++, können Daten dieses Formats gelesen und geschrieben werden. Diese Arbeit erweitert SKilL und implementiert das Konzept eines Encodings. Dabei werden die serialisierten Daten mit einem Encoding komprimiert. Dieses Dokument geht nicht nur auf die Implementierung in Java und C++ ein, sondern evaluiert auch die Auswirkungen der Kompression auf die Performance im Kontext eines in der Praxis verwendeten Datensatzes.

Inhaltsverzeichnis

1. Einleitung

Die im Rahmen einer Softwareanalyse anfallenden Daten, welche von unterschiedlichen Programmiersprachen verarbeitet werden müssen, führten zur Entwicklung der *Serialization Killer Language* (SKilL). Diese erlaubt eine sprach- und plattformunabhängige Serialisierung von Datenstrukturen. Zunächst wird mit einer Spezifikation ein Datenformat spezifiziert, welches nach der Generierung einer Anbindung für eine beliebige Sprache, gelesen und geschrieben werden kann. Im weiteren Fokus steht dabei die Änderungstoleranz, so dass eine Änderung einer Spezifikation den Datensatz nicht invalidiert. [Fel17, p. 3f]

1.1. Zielsetzung

Zur Verkleinerung der serialisierten Datenmenge, bieten sich Kompressionsalgorithmen an. Die Kompression eines Datensatzes verursacht allerdings zusätzlichen Rechenaufwand und könnte daher die Effizienz des Lese- und Schreibvorganges negativ beeinflussen. Diese Arbeit erweitert SKilL so, dass Datensätze mit einer Bitvector-Kompression oder einer LZ4-Kompression serialisiert werden können. Die Implementierung erfolgt dabei für die Java- und C++-Anbindungen an SKilL [ISO13][GJS+18]. Diese Implementierung wird auf ihren Einfluss auf die Performanz des Lese- und Schreibvorganges untersucht um festzustellen, ob und wie sie sinnvoll anzuwenden sind. Des weiteren wird ein Mechanismus implementiert, der es erlaubt Daten, die nur aus Standardwerten bestehen, von der Serialisierung auszuschließen (Defaultwert-Optimierung). Die Implementierung soll so gestaltet werden, dass die Änderungstoleranz beibehalten wird und sie sich dynamisch an in einer Datei vorgefundene Umstände anpasst.

1.2. Gliederung

In Kapitel 2 werden zunächst die Grundkonzepte von SKilL anhand eines Beispiels erklärt. Kapitel 3 geht auf die Implementierungen zunächst allgemein und dann auf die einzelnen Programmiersprachen ein. Dazu wird zunächst immer die bisherige Implementierung betrachtet und danach besprochen, was geändert wurde. Anschließend beschäftigt sich Kapitel 4 mit der Evaluation der zuvor beschriebenen Arbeit. Zum Abschluss wird in Kapitel 5 ein Fazit gezogen und ein Ausblick auf Erweiterungen dieser Arbeit gegeben.

1.3. Begriffsklärung

Hier werden einige häufig auftretende Begriffe genauer definiert:

SKilL Kontextabhängig der Sprachgenerator, die Spezifikationssprache, das binäre Serialisierungsformat, oder die Implementierung von SKilL.

.sf-Datei Datei mit binär serialisierten Daten, die der Struktur einer bestimmten Spezifikation entsprechen.

State Repräsentation der Daten und Typen einer *.sf-Datei* nach ihrer Deserialisierung als Objektstruktur im lesenden Programm. Beispiel: Ein Feld wird zu einer Instanz von `FieldDeclaration`.

Chunk Bereich im Felddatenbereich eines Typblocks, in dem die Daten eines bestimmten Feldes serialisiert sind.

Encoding Daten eines Feldes werden durch ein Encoding reversibel verändert serialisiert. Im allgemeinen oft und hier nur zu Zwecken der Kompression, dann auch *Kompression*, respektive werden *Kodieren* und *komprimieren* bzw. *Dekodieren* und *Dekomprimieren* synonym verwendet.

Serialisierung Der Begriff Serialisierung sowie davon abgeleitete Wörter wie z.B. „serialisieren“ beschreiben sowohl den Lese- als auch den Schreibvorgang und nicht nur den Schreibvorgang wie es bei „Serialisierung“ im engeren Sinne der Fall ist.

1.4. Die Encodings

Diese Arbeit implementiert zwei Encodings: Das Bitvector-Encoding und das LZ4-Encoding.

Ein Bitvector-Encoding kann auf ein bool'sches Feld angewandt werden. Dabei werden die Instanzdaten in einem Bitvector komprimiert. Ein gewöhnlicher bool'scher Wert nimmt einen Byte an Speicher ein. Das ist normal der Fall, da ein gewöhnlicher Prozessor nicht weniger als einen Byte adressieren kann. Für die logische Darstellung eines bool reicht aber ein einzelnes Bit aus. Will man nun viele bool'sche Werte zusammen speichern, bietet es sich an jeweils eine Gruppe aus 8 Werten in einem Byte zu speichern. Die Adressierung eines Bits lässt sich über Maskierungstechniken problemlos realisieren.

LZ4 ist ein verlustfreier Kompressions-Algorithmus, der sich besonders durch seine hohe Kompressions- und Dekompressionsgeschwindigkeit auszeichnet. Der Algorithmus verwendet ein wörterbuchbasiertes Verfahren. Dabei wird der Eingabetext nach sich wiederholenden Zeichensequenzen durchsucht. Die sich wiederholende Sequenz kann mit einer Referenz auf ein vorheriges Vorkommen ersetzt werden. Dies reduziert den Speicherverbrauch. [LZ4][Ami16]

Die Entwickler geben Werte von 400 MB/s pro Kern an [Col18]. LZ4 bietet auch einen weiteren Modus an (*LZ4HC*), bei dem das Kompressionsverhältnis auf Kosten der Prozessorzeit verbessert werden kann. Eine Unterstützung für LZ4HC wird zusätzlich zur normalen LZ4-Unterstützung in SKilL implementiert.

2. SKilL

Bei der *Serialization Killer Language* (SKilL) handelt es sich um ein plattform- und sprachunabhängiges Serialisierungssystem, welches auf eine hohe Performanz und starke Vorwärts-/Rückwärtskompatibilität ausgelegt ist. SKilL besteht aus folgenden Komponenten: Mithilfe der *Spezifikationssprache* kann ein eigenes Typsystem definiert werden, welches den Aufbau der zu serialisierenden Daten bestimmt. Der *Codegenerator* ist fähig, eine solche Spezifikation in eine Codeanbindung für eine bestimmte Programmiersprache umzusetzen. Mithilfe dieser Anbindung können dann beliebige Daten basierend auf der zuvor deklarierten Struktur in einem binären Dateiformat serialisiert werden. Mit *SKilL* wird kontextabhängig das gesamte System, die Spezifikationssprache, der Generator oder auch die Spezifikation für das binäre Dateiformat bezeichnet.[Fel17, p. 3f]

2.1. Spezifikationssprache

Die Spezifikationssprache dient dazu eine Spezifikation zu erstellen. In einer Spezifikation werden Typen definiert. Ein Typ ist dabei eine Sammlung aus Datenfeldern eines bestimmten Typs. Es wird Einfachvererbung unterstützt. Typen und Felder können mit bestimmten Annotationen versehen werden. Eine davon sind die Restrictions welche im Bereich dieser Arbeit eine zentrale Rolle einnehmen und in Abschnitt 2.3 behandelt werden. Eine Spezifikation kann modular auf mehrere Dateien verteilt werden.

SKilL stellt einige grundlegende Typen zur Vefügung. Dazu gehören Integer mit verschiedenen Größen (i8...i64), Integer variabler Größe (v64), Bool'sche Werte, Strings und IEEE-754 Floating Point-Werte (f32, f64). Zusätzliche zusammengesetzte Container-Typen sind Arrays mit konstanter und variabler Größe, Listen, Sets und Maps. Typen die in einer Spezifikation definiert werden, werden *benutzerdefinierte Typen* genannt. [Fel17, p. 18ff]

Felder können ebenfalls einen benutzerdefinierten Typ haben (User Type), welcher statisch spezifiziert wird und Referenzsemantik hat. Auch dynamisch typisierte Referenzen sind möglich (sog. Annotation), dabei ist der tatsächliche Typ zusammen mit der Objektreferenz ein Teil des Wertes der Annotation. Diese kann auch einen unbekannten Typ beinhalten und erlaubt somit dynamische Erweiterungspunkte. [Fel17, p. 19]

Zusätzlich erlaubt SKilL auch weitere Hilfstypen wie Interfaces, Enums, Typedefs und Views, welche auf klassische Typdefinitionen reduziert werden können. In diesem Fall sind sie rein dekorativ. Die weitere Behandlung dieser Typen liegt nicht im Rahmen dieser Arbeit. [Fel17, p. 21f.]

Listing 2.1 zeigt ein Beispiel für eine sehr simple Spezifikation

Listing 2.1 Beispiel für eine SKilL-Spezifikation

```
1  /** Benutzerdefinierter Typ */
2  Student {
3          string name;
4          /** Integer mit variabler Größe */
5          v64 matriculationNumber;
6
7          /** Liste variabler Größe mit benutzerdefiniertem Typ */
8          list<Exam> exams;
9  }
10
11 Exam {
12         string moduleName;
13         v64 creditPoints;
14 }
15
16 /** Nutzung von Vererbung */
17 GradedExam extends Exam {
18         /** Neues Feld */
19         f32 grade;
20 }
```

2.2. Codegenerator

Aus der Spezifikation kann eine Sprachanbindung generiert werden, welche eine Schnittstelle zur Repräsentation der Typen im Typsystem der entsprechenden Programmiersprache darstellt. Intern wird für die verschiedenen Typen und Felder Code zum Lesen, Schreiben, Erzeugen und Manipulieren generiert. Diese ist jedoch nicht zwingend nötig. Auch unbekannte Typen und Felder (welche nicht spezifizert aber in einer serialisierten Datei vorhanden ist) können gelesen und verändert werden. Dabei wird aber kein spezifischer Code generiert, so dass der Zugriff auf die Daten reflexiv und generisch und damit nicht optimiert oder statisch prüfbar erfolgt.

Listing 2.2 zeigt die Verwendung der generierten Sprachanbindung für die Spezifikation in Listing 2.1. Wie zu sehen ist, wird jeweils eine Klasse für `Student` und `Exam` erzeugt. Diese Klasse ist repräsentativ für ein Objekt vom entsprechenden Typ, und damit eine direkte Abbildung der spezifizierten Datenstruktur auf das Java-Typsystem. Diese Klassen bezeichnen wir als *generierte Typen*.

2.3. Restrictions

Wie zuvor erwähnt ist es möglich Typen und Felder zu annotieren. Eine dieser Möglichkeiten sind Restrictions. Hier von Bedeutung sind hauptsächlich die Restrictions auf Feldern, daher werden auch nur diese hier weiter behandelt. Restrictions werden zusammen mit der Felddefinition serialisiert, sie sind also nicht nur in der Spezifikation existent, sondern kann auch aus einer serialisierten Datei gelesen werden. Einige Restrictions beschränken dem Namen folgend den Wertebereich eines

Listing 2.2 Beispiel für die Verwendung einer Java-Anbindung

```
1  // Datei lesen
2  SkillFile file = SkillFile.open("students.sf");
3  // Ein neuer Student
4  Student student = file.Students().make();
5  student.setName("Max Mustermann");
6
7  // Neues Exam - Polymorphie funktioniert
8  Exam exam = file.GradedExams().make(1.0f, 120L, "Mustermodul");
9  student.setExams(new LinkedList<>());
10 student.getExams().add(exam);
11
12 file.close();
```

Feldes und stellen eine Invariante dar. Bei der Serialisierung und Deserialisierung wird überprüft ob der Wert den entsprechenden Restrictions genügt. Ist das nicht der Fall ist der State (Repräsentation der gelesenen Daten) ungültig. Folgende Restrictions fallen in diese Kategorie: [Fel17, p. 25ff.]

- **NonNull** Null nicht als Wert für Referenztypen außer Annotation erlaubt (String, User Type)
- **Range** Beschränkt den Wertebereich von integralen und Gleitkommatypen auf ein Intervall
- **OneOf** Beschränkt die möglichen Typen einer Annotation

Alle anderen Restrictions lassen sich nicht „überprüfen“, sondern haben anderweitige Auswirkungen auf das Feld:

- **Default** Spezifiziert den Standardwert für dieses Feld
- **Coding** Serialisiert/Deserialisiert Felddaten und wendet dabei eine Kodierung an (z.B. LZ4)
- **Constant-Length Pointer** Erzwingt die Verwendung von i64 für Referenzen und verwendet keinen v64-Integer

[Fel17, p. 29ff.]

Jede Restriction besitzt eine ID, die den Restriction-Typ eindeutig definiert. Die binäre Repräsentation einer Restriction setzt sich zusammen aus ihrer ID und den Parametern. Welche Parameter eine Restriction besitzt, ist abhängig von ihrem Typ.

Diese Arbeit implementiert die Coding-Restriction und beschreibt im Zuge dessen die Implementierung der Serialisierungs-Infrastruktur für Restrictions im Allgemeinen.

2.4. Binäres Dateiformat

Zur Serialisierung der Daten wird ein binäres Dateiformat verwendet. Dieses speichert neben den Daten auch die zugehörigen Typinformationen, sodass unbekannte Typen und Felder auch vorhanden und über Reflection zugreifbar sind. Um im folgenden Bezug auf eine Datei in diesem Format zu nehmen, wird der Begriff *.sf-Datei* verwendet.

Eine .sf-Datei besteht aus einer sich wiederholenden Sequenz aus einem String- und einem Typblock. Die String-Blöcke enthalten alle verwendeten Strings, welche in den serialisierten Daten später referenziert werden. Jeder String hat eine ID, die sich implizit aus seiner Position ableitet und bei 1 beginnt. Auf String-Blöcke wird hier nicht weiter eingegangen.

2.4.1. Typblock

Der erste Wert gibt an, wie viele Typen in diesem Block definiert sind. Danach kommt die Definition der einzelnen Typen, welche unter anderem spezifizieren, wie viele Instanzen in diesem Block serialisiert sind und wie viele Felder der Typ besitzt. Nachdem alle Typen definiert wurden kommen die einzelnen Felddefinitionen der Typen. [Fel17, p. 40]

Eine Felddefinition beginnt mit der ID des Feldes. Diese identifiziert das Feld Typ-lokal eindeutig. Die IDs werden von 1 aufsteigend vergeben. Auf die ID folgt der Name des Feldes. Dieser ist direkt aus der Spezifikation abgeleitet und ist als eine String-Referenz gespeichert. Der eigentlich String liegt in dem vorgehenden Stringblock. Als nächstes wird der Typ des Feldes angegeben. Jeder Typ hat eine eigene ID. Zusammengesetzte Typen wie Listen serialisieren zusätzlich noch die ID der Typparameter. Auch benutzerdefinierte Typen haben eine generische Typ-ID. Eine Auflistung der IDs ist in [Fel17, p. 61] zu finden. Darauf folgend sind die Restrictions dieses Feldes definiert. Als erstes wird die Anzahl der Restrictions angegeben, danach wird für jede Restriction ihre ID und ihre spezifischen Parameter serialisiert. Diese hängen von dem Restriction-Typ ab. So wird bei einer Coding-Restriction der Name des Encodings spezifiziert, z.B. „`lz4`“. Name, Typ und Restrictions werden nur bei der ersten vorkommenden Felddefinition angegeben und sind bei Append-Definitionen nicht vorhanden (mehr dazu in Abschnitt 2.5) Zuletzt findet man den Offset in der Definition. Dieser gibt den Offset der Felddaten dieses Feldes von dem Beginn aller Felddaten in Bytes an. So kann die Position eines Chunks in konstanter Zeit berechnet werden. Dies dient dazu Chunks die On-Demand geladen werden zu überspringen. Anfang und Ende können so dadurch auch konstant bestimmt werden. Außerdem ist es so möglich die Chunks parallel zu laden. [Fel17, p. 40]

Nach der letzten Felddefinition folgt der Felddaten-Bereich. SKilL speichert Daten gruppiert nach Feldern und nicht nach Instanzen. Das bedeutet dass der Felddaten-Bereich eine Sequenz aus Chunks ist. Jedes Feld hat einen Chunk der die Werte dieses Feldes speichert. Die Chunks sind nach Feld-ID aufsteigend und damit nach Definitionsreihenfolge sortiert. Die Bereiche der einzelnen Chunks sind durch das Offset-Feld der Definition wohldefiniert. Ein Feldwert kann von der Typ-Klasse gelesen werden. Wird eine Anbindung für das Feld generiert, kann der generierte Code dies vermeiden, da der Typ statisch bekannt ist. Der serialisierende Code enthält dann direkt die Serialisierungslogik anstatt den Wert an die Typ-Klasse weiterzuleiten. [Fel17, p. 40]

2.5. Appends

SKilL bietet zwei verschiedene Möglichkeiten an, in eine .sf-Datei zu serialisieren. Die einfachste Möglichkeit ist es, den gesamten State von Beginn an in die Datei zu schreiben (*Write*). Wurde der State aus einer bereits existierenden .sf-Datei gelesen, wird der komplette Inhalt erneut serialisiert. Die Datei enthält danach genau einen String- und einen Typblock, welcher offensichtlich alle existierenden benutzerdefinierten Typen und Felder enthält.

Die andere Möglichkeit (*Append*) setzt die existierende .sf-Datei fort. Dabei werden die bereits vorhanden Blöcke in der Datei nicht verändert oder überschrieben. Das führt dazu, dass pro Append-Vorgang ein weiterer String- und Typblock in die Datei geschrieben wird. Der neue Stringblock enthält alle Strings die neu hinzugekommen sind, z.B. durch neue Instanzen eines Felds vom Typ String, oder Namen neuer Felder/Typen. Im einfachsten Szenario kommen nur neue Instanzen diverser Typen vor. Der neue Typblock enthält eine Typdefinition von jedem Typ der neue Instanzen hat und seinen Feldern. Dabei werden aber nicht alle Informationen serialisiert. Bei den Feldern wird nur die ID und der Offset gespeichert. Name, Typ und Restrictions sind an dieser Stelle schon bekannt durch die erste Definition im ersten Block. Sie gelten auch in diesem Block, da es sich um dasselbe Feld handelt. Die ID wird benötigt um Referenz auf das Feld zu nehmen welches hier einen neuen Chunk besitzt. Der Offset ist block-lokal und gibt im neuen Block an wo die neuen Daten für das Feld stehen. Der Felddatenbereich enthält dann einen Chunk für jedes Feld mit den Daten der neuen Instanzen des besitzenden Typs.

In einem etwas komplizierten Szenario können beim Append-Vorgang neue Typen und insbesondere neue Felder für bereits existierende Typen hinzukommen. Dies ist möglich z.B. wenn eine neuere Version einer Spezifikation zusätzliche Felder besitzt, die in der .sf-Datei nicht existieren. Beim Lesen aus der Datei erhalten diese Felder Standardwerte. Beim serialisieren werden diese neuen Felder auch gespeichert. Bei einem Append ist das nicht-trivial. Der neue Typblock enthält in diesem Fall weitere Felddefinitionen für die neuen Felder. Da diese neu definiert werden, werden im Gegensatz zu den anderen Definitionen in dem neuen Block auch Name, Typ und Restrictions geschrieben. Die Problemstellung in diesem Fall ist, dass ältere Typblöcke Instanzen eines Typen enthalten können, der bei diesem Append-Vorgang ein weiteres Feld bekommt. Die Instanzen können logisch insofern geändert worden sein, dass die neu definierten Felder Werte zugewiesen bekommen haben. Bereits serialisierte Werte können zwar geändert werden, das hat bei einem Append aber keine Wirkung. Die Werte für diese älteren Instanzen müssen in dem neuen Typblock gespeichert werden. Wird ein Feld zum ersten Mal definiert, muss es also die Werte aller bereits existierenden Typinstanzen speichern.

2.6. Storage-Pools

Ein Storage-Pool existiert für jeden benutzerdefinierten Typ und speichert die Instanzen dieses Typs und damit die Daten aller Felder. Der Storage-Pool eines Basis-Typs, der von keinem Typ erbt, wird Base-Pool genannt. Jede Instanz besitzt eine *SkillID*, welche bei 1 startet. Die ID ist für jeden Base-Pool und damit seine Subtypen eindeutig. Zwei Objekte mit derselben ID können somit keinen gemeinsamen Basistypen haben. Die SkillID wird nicht serialisiert, denn sie kann durch die Position in der .sf-Datei bestimmt werden. Sie dient außerdem als Mittel zur Referenzierung einer bestimmten Instanz bei Feldern vom Typ Annotation oder mit einem benutzerdefinierten Typ [Fel17, p. 44].

Der Base-Pool speichert die Instanzen in einem Array indiziert nach ihrer SkillID. Der Typ des Arrays ist dabei bei bekannten Typen der generierte Typ (s. Abschnitt 2.2), welcher somit die Werte aller bekannten Felder speichert. Daraus folgt, dass bei unbekannten Feldern die Werte nicht im Storage-Pool gespeichert werden können. Da die Vererbung genauso auf den generierten Typen abgebildet wird, kann dieses Array unter Ausnutzung von Polymorphie auch Instanzen von

Subtypen enthalten. Aus diesem Grund existiert für jeden Basis-Typ und seine Hierarchie ein Array, welches der Base-Pool besitzt. Die untergeordneten Storage-Pools speichern nur Referenzen auf dieses Array.

SKilL sichert eine gewisse Typordnung zu, welche garantiert, dass ein Supertyp im Sinne der Ordnung kleiner ist als ein Subtyp [Fel17, p. 41]. Innerhalb eines Typblocks gibt es eine bestimmte Anzahl an Instanzen eines Typs. Die Position einer Instanz in der .sf-Datei bestimmt implizit seine SkillID. Daher liegen die *dynamischen Instanzen* eines Basistyps innerhalb eines Blocks im Daten-Array adjazent zueinander. Als dynamische Instanzen eines Typs bezeichnen wir alle Instanzen dieses Typs und seiner Subtypen. Durch die Typordnung liegen Instanzen eines bestimmten Typs innerhalb eines Blocks ebenfalls nebeneinander. Der *Base Pool Offset* (BPO) welcher für jeden Typ und Block existiert, spezifiziert bei welcher SkillID die Instanzen dieses Typs beginnen.

Jeder Storage-Pool reflektiert die Blockstruktur indem er Informationen darüber hält. Dazu gehört für jeden Block der BPO und die Anzahl der Instanzen. So können die Instanzen dieses Typs im Daten-Array referenziert werden.

3. Implementierung

Die Implementierung teilt sich in drei logische Bereiche auf: Die unspezifische Implementierung, die Java-Implementierung und die C++-Implementierung. Die unspezifische Implementierung erklärt die Konzepte und Realisierungen die in beiden Sprachen verwendet werden und schafft eine Basis auf der die sprachspezifischen Implementierungen aufbauen. Für jeden Bereich wird zunächst auf die bisher existierende Implementierung eingegangen, bevor die Änderungen durch diese Arbeit besprochen werden.

3.1. Unspezifische bestehende Implementierung

Die Implementierungen von C++ und Java besitzen viele Gemeinsamkeiten, die hier beschrieben und in den spezifischen Teilen in Abschnitt 3.3 und Abschnitt 3.5 genauer abgegrenzt werden.

3.1.1. Allgemeines

Restrictions werden durch die Klasse `FieldRestriction` beschrieben. Diese enthält die ID der Restriction. Der detaillierte Aufbau der Hierarchie ist sprachspezifisch. Jeder unterstützte Restriction-Typ besitzt eine eigene Unterklasse. Storage-Pools werden durch die Klasse `StoragePool` beschrieben. Es existiert eine Subklasse `BasePool` welche spezielles Verhalten für Base-Pools enthält.

Ein Feld wird durch die Klasse `FieldDeclaration` repräsentiert. Diese Klasse enthält z.B. alle Restrictions die das Feld besitzt, den Typ des Feldes, den besitzenden benutzerdefinierten Typen in dem sie enthalten sind und Chunk-Informationen u.v.m.. Im besonderen Fokus dieser Arbeit liegen allerdings die Methoden, die relevant bei der Serialisierung sind. Diese werden im Folgenden erläutert. Zuvor ist noch eine Begriffsklärung bezüglich der Chunks nötig. Es gibt zwei Arten von Chunks: Der *BulkChunk* ist der erste Chunk eines Feldes der im gleichen Block wie der ersten Felddefinition auftritt. Bei diesem besteht die Möglichkeit, dass er, wie in Abschnitt 2.5 beschrieben, Werte für Instanzen aus vorherigen Blöcken enthält. Alle weiteren Chunks sind *SimpleChunks*. Bei einem Write-Vorgang wird nur ein Block und damit nur ein Chunk pro Feld geschrieben. Hier wird ein SimpleChunk verwendet, obwohl es der erste Chunk des Feldes sein wird. Dies ist möglich, da die zusätzlichen Informationen eines BulkChunks für vorherige Blöcke nicht benötigt werden.

Ein Feld muss 3 wichtige Operationen durchführen: Lesen, Schreiben und Größe berechnen. Beim Lesen werden die Feldwerte aus einem Stream gelesen und in das Daten-Array des Storage-Pools eingeordnet. Beim Schreiben werden die entsprechenden Daten aus dem Daten-Array in einen Stream geschrieben. Die Größen-Berechnung wird auch Offset-Berechnung genannt und bestimmt die Größe der beim Schreiben geschriebenen Daten im voraus. Es gibt jeweils Versionen für SimpleChunks und BulkChunks. Die Namen der Methoden lauten: `rsc` (Read Simple Chunk), `rbc` (Read Bulk Chunk), `osc` (Offset Simple Chunk), `obc` (Offset Bulk Chunk), `wsc` (Write Simple Chunk)

und `wbc` (Write Bulk Chunk). Die SimpleChunk-Methoden erhalten ein SkillID-Intervall welches bestimmt welche Instanzen verarbeitet werden sollen. Die BulkChunk-Methoden benötigen mehr Informationen (siehe auch Abschnitt 3.1.2).

Die Methoden welche BulkChunks verarbeiten sind bei bekannten Feldern so implementiert, dass für jeden Block, für dessen Instanzen der Chunk Daten enthält, ein Aufruf der SimpleChunk-Methode auf diesen Instanzen erfolgt. Für bekannte Felder enthält der generierte Code die Implementierung für die SimpleChunk-Methoden.

3.1.2. Blocks und Chunks

Der State enthält auch Informationen über die Blöcke und Chunks der .sf-Datei, bzw. neue Blöcke und Chunks die noch geschrieben werden. Jeder `StoragePool` hält Blockinformationen für jeden Block in dem er eine Typdefinition hat. Zu diesen Informationen gehört die Anzahl der Instanzen in diesem Block, sowie der BPO. Die Reihenfolge der Blöcke aus der .sf-Datei bleibt im State erhalten. Die Felder der einzelnen Typen enthalten Informationen über die Chunks. Das bedeutet Anfang und Ende des Chunks in der .sf-Datei, sowie die Anzahl der Instanzen zu denen Feldwerte im Chunk gespeichert werden. Ein SimpleChunk kennt noch den BPO, so dass er aus der Anzahl der Instanzen mit dem BPO berechnen kann für welche SkillIDs er Feldwerte enthält. Es handelt sich dabei um das Intervall $[BPO, BPO + Count)$. Der BulkChunk muss zusätzlich speichern, für wie viele Typblöcke er Daten enthält. Über die Blockinformationen aus dem besitzenden Storage-Pool kann die Anzahl der Instanzen in jedem Block und der jeweilige BPO abgefragt werden.

3.1.3. Streams

Beide Implementierungen stellen eine Stream-API zur Verfügung. Diese Streams können mit Dateien oder Speicherbereichen interagieren. Die API wird in jeder Implementierung genauer erklärt. Der Allgemeinheit halber wird in diesem unspezifischen Teil generalisiert mit `InStream` und `OutStream` operiert. Die Streams bieten Methoden an um SKilL-eigene Datentypen zu schreiben z.B. *v64*.

3.1.4. Lesen

Das Lesen einer .sf-Datei wird hier nicht genauer erläutert. Die Blöcke und ihre Definitionen werden sukzessive gelesen und in den State überführt. Bei Veränderungen durch die Implementierung im Zuge dieser Arbeit werden genauere Details angeführt.

3.1.5. Schreiben

Es folgt eine Übersicht über den Serialisierungsprozess jeweils für eine Write-Operation und eine Append-Operation. Die nicht relevanten Bereiche werden dabei nur oberflächlich erklärt. Das dient dazu die vorgestellten Änderungen im Serialisierungsprozess besser einordnen zu können.

Als erstes erfolgt ein Vorbereitungsschritt, der von beiden Serialisierungsmodi geteilt wird. Zunächst werden die Strings gesammelt. Das bedeutet alle Strings, die in den Stringblock serialisiert werden müssen, werden beim `StringPool` registriert. Dazu gehören die Namen aller Typen und Felder, sowie alle im Graphen referenzierten Strings. Des weiteren ist es wichtig, dass auch die Daten der LazyFields im State geladen sind, was in letzter Instanz hier erfolgt.

Write

Der erste Schritt der Serialisierung ist die Kompression der Storage-Pools. Da seit dem Erstellen des States neue Instanzen hinzugekommen oder gelöscht worden sein können, müssen die Instanzen in das Instanz-Array einsortiert/entfernt werden. Wie in Abschnitt 2.6 beschrieben, müssen Instanzen eines Typs innerhalb eines Blocks adjazent zueinander liegen. Daher werden die SkillIDs und der BPO des letzten Blocks (nur dieser wird geschrieben) neu berechnet. Wurde der State aus einer .sf-Datei mit mehreren Typblöcken geladen, müssen diese auf einen komprimiert werden. Die Neuvergabe der IDs erfolgt parallel auf allen Base-Pools. [Fel18, p. 150f.]

Danach wird der Stringblock geschrieben. Die Strings wurden in der Vorbereitung alle gesammelt und können nun geschrieben werden. Die ID jedes Strings steht zur Referenzierung für den weiteren Prozess zur Verfügung. Der nächste Schritt ist die parallelisierte Offset-Berechnung aller Felder. Jedes Feld berechnet dabei die Größe seiner Daten um in der Felddefinition berechnen zu können wo der zugehörige Chunk endet. Da der gesamte State in einem Block gespeichert wird, besitzt jedes Feld nur einen SimpleChunk. Daher wird dafür die in Abschnitt 3.1.1 beschriebene Methode `osc` verwendet.

Zuerst werden die Typdefinitionen geschrieben. Dieser Prozess ist hier nicht weiter von Bedeutung. Nach diesem Schritt erfolgt das Schreiben der Felddefinitionen in Reihenfolge der Typdefinitionen. Dabei werden die in Abschnitt 2.4 beschriebenen Daten geschrieben. Insbesondere findet hier die Serialisierung der Restrictions des Feldes statt. Als letztes erfolgt das parallelisierte Serialisieren der Felddaten. Hier werden für jeden Chunk jedes Feldes die Funktionen `wsc`/`wbc` aufgerufen. Bei einer Write-Operation gibt es wie vorhin erläutert nur einen Chunk, weswegen pro Feld ein `wsc`-Aufruf erfolgt.

Append

Auch hier folgt nach den oben beschriebenen Vorbereitungsschritten eine Vergabe von SkillIDs an neue Instanzen. Die Kompression bleibt allerdings aus, da die Blockinformationen über bisherige Typblöcken nicht geändert werden kann. Außerdem werden hier die Chunk-Informationen für die neu hinzugekommenen Daten generiert.

Als nächstes wird festgestellt welche Storage-Pools relevant für die Serialisierung sind. Ein Storage-Pool ist relevant, wenn er bisher noch nicht in der .sf-Datei existiert, oder eines seiner Felder neue Daten zu serialisieren hat. Dies ist wiederum der Fall wenn es neue Instanzen des StoragePools gibt, oder das Feld neu ist und bisherigen Instanzen neue Werte zuweist.

Die erste tatsächliche Serialisierungsoperation ist das Schreiben des Stringblocks. Dieser enthält alle neuen Strings. Danach wird die Größe der einzelnen Felddaten berechnet. Auch hier kommt genau ein weiterer Chunk pro Feld hinzu. Abhängig davon ob dieser ein SimpleChunk oder ein BulkChunk ist, wird parallel auf jedem Feld `osc` bzw. `obc` aufgerufen.

Im nächsten Schritt werden die Typdefinitionen aller relevanten Storage-Pools geschrieben und die Relevanz der einzelnen Felder festgestellt. Nachdem alle Typdefinitionen geschrieben wurden, werden nun alle Felddefinitionen geschrieben, die davor als relevant angesehen wurden. Je nachdem ob das Feld neu ist, wird zusätzlich zu der ID und dem Offset noch der Name, der Typ und die Restrictions geschrieben. Der letzte Schritt ist das parallelisierte Serialisieren der Felddaten, welches wie bei der Write-Operation durch `wsc`/`wbc` erfolgt.

3.2. Unspezifische Implementierung der Kompression von Felddaten

Hier werden neue Konzepte und Realisierungen vorgestellt, die im Zuge der Einführung der Kompression von Felddaten entstanden sind.

3.2.1. Restrictions

Die Coding-Restriction beinhaltet ein *Encoding*, welches die weitere Kompressionslogik enthält. Über eine Factory-Methode ([GJHV11, p. 131ff]) kann eine Coding-Restriction aus einem String erzeugt werden. Die Absicht dahinter ist, dass eine Restriction mit passendem Encoding direkt aus dem gelesenen Stringparameter erstellt werden kann. Es gilt folgende Abbildung:

- „bitvector“ - `BitvectorEncoding`
- „lz4“ - `LZ4FastEncoding`
- „lz4cached“ - `LZ4FastEncoding (cached)`
- „lz4hc“ - `LZ4HCEncoding`
- „lz4hccached“ - `LZ4HCEncoding (cached)`

Die Encodings werden in Abschnitt 3.2.2 genauer definiert.

Restrictions werden in der Spezifikation angegeben. Wenn der State nicht aus einer .sf-Datei gelesen wird und damit keine Restrictions gelesen werden, muss das Feld diese trotzdem enthalten um sie später zu schreiben. Dazu fügt der generierte Code im Konstruktor die entsprechenden Restrictions in die FieldDeclaration ein. Dadurch entsteht jedoch ein Konflikt. Wird eine .sf-Datei gelesen, wird sowohl die Restriction aus der .sf-Datei, als auch die generierte Restriction der FieldDeclaration hinzugefügt. Aufgrund der hohen Flexibilität von SKilL ist es auch möglich, dass beide Restrictions unterschiedlich sind. Die Behandlung dieses Konfliktes liegt außerhalb des Fokus dieser Arbeit. Es wurde eine behelfsmäßige Änderung vorgenommen. Diese verbietet mehrere Restrictions des gleichen Typs, um einem linearen Wachstum redundanter Restrictions vorzubeugen. Diese Implementierung ist so, dass die spezifizierte Restriction dominant ist. Die Spezifikation empfiehlt eine Zusammenführung aller Restrictions, die korrekte Umsetzung dieser Empfehlung ist

nicht Teil dieser Arbeit [Fel17, p. 25f.]. Auf diese Implementierung wird nicht weiter eingegangen. Dieser Konflikt tritt aus später beschriebenen Gründen nicht bei der Coding-Restriction auf und ist deshalb hier nicht von weiterer Bedeutung.

3.2.2. Encodings

Ein Encoding kapselt die Logik für das kodieren, dekodieren, etc. Die Abtrennung der Encodings von der Coding-Restriction geschieht nicht nur aus Designgründen. Sie ist auch aufgrund der Tatsache nötig, dass Encodings ohne eine entsprechende Restriction existieren können und müssen. Eine `FieldDeclaration` besitzt ein `Encoding`, welches beim Lesen und Schreiben der Felddaten dieses Feldes verwendet wird. Die Existenz einer Coding-Restriction in den Restrictions der `FieldDeclaration` dient nur zum Schreiben dieser. Das `Encoding` in jener Restriction wird nur verwendet, wenn das `Encoding` der `FieldDeclaration` gesetzt wird. Eine genauere Erläuterung dieser Infrastruktur findet sich in Abschnitt 3.2.4.

Encodings sind ebenfalls durch eine Klassenhierarchie realisiert, die von der Basisklasse `Encoding` ausgeht. Diese definiert eine Schnittstelle, die zum Kodieren und Dekodieren der Felddaten verwendet werden kann. Mittels `encode` kann ein Felddaten-Chunk kodiert werden. Dabei liest die Methode die Daten aus dem InStream und schreibt die kodierten Daten in den OutStream. `decode` wird verwendet um einen kodierten Daten-Chunk in rohe Felddaten zu transformieren. Der InStream liefert dabei die kodierten Felddaten. `numElements` gibt an, wieviele Instanzen des Feldes in dem Chunk serialisiert wurden. Diese Information benötigt das `BitvectorEncoding` wie später erläutert wird. Mit `reset` bietet das Encoding die Möglichkeit, einen eventuell vorhandenen State zurückzusetzen, um die Korrektheit über mehrere Serialisierungsvorgänge hinweg zu behalten. Ein Encoding kennt außerdem seinen eigenen Namen, da beim Schreiben einer Coding-Restriction dieser geschrieben wird.

Die spezifische Implementierung unterteilt sich weiter in zwei Arten. Beim Schreiben muss die Größe der serialisierten Daten beim Schreiben der Felddefinition feststehen. Das ist notwendig, da die Felddefinition das Ende des Daten-Chunks relativ zum Beginn der Felddaten angibt (siehe Abschnitt 2.4.1). Dadurch wird die Berechnung dieser Offsets früh durchgeführt (s. Abschnitt 3.1.5). Da die Größe der Daten sich durch das Encoding auf dem Feld ändert, muss die Offset-Berechnung die Größe der kodierten Daten berechnen.

Für das `BitvectorEncoding` ist diese Berechnung trivial und durch die Formel $\lceil \frac{size}{8} \rceil$ gegeben, wobei $size$ für die Originalgröße der Daten steht. Solche Encodings, die die kodierte Größe aus der Originalgröße ableiten können, werden als *SimpleEncoding* bezeichnet.

Bei LZ4 ist diese Vorgehensweise nicht möglich. Die kodierte Größe hängt hier von den Daten an sich ab. Um herauszufinden wie groß diese ist, muss der Datensatz komprimiert werden. Das Komprimieren der Felddaten bei der Offset-Berechnung wird als *Prepass*, das Encoding entsprechend als *PrepassEncoding* bezeichnet. Das Ergebnis des Prepass kann verworfen werden, um die Speicherkomplexität des Programms zu reduzieren. Dies hat zur Folge, dass bei dem Schreiben der Felddaten diese erneut komprimiert werden müssen. Alternativ können die komprimierten Daten zwischengespeichert werden. Beide Varianten sind durch diese Arbeit implementiert und evaluiert worden, sowohl mit LZ4, als auch mit LZ4HC. Die Encodings mit Zwischenspeicherung des Prepass heißen *LZ4Cached* bzw, *LZ4HCCached*.

Die beiden Typen von Encodings spiegeln sich auch in der Klassenhierarchie wider. Beide implementieren die Berechnung der Größe der komprimierten Daten. Dies geschieht entweder auf der Basis von der Größe der alten Daten, oder auf Basis der konkreten Daten, die komprimiert werden sollen. Das `PrepassEncoding` implementiert außerdem das Caching des Prepasses, welches ein- und ausschaltbar ist. Ein weiteres Feature des `PrepassEncoding` ist die sogenannte *Omission*. Bei der Offset-Berechnung wird festgestellt ob die komprimierte Größe tatsächlich kleiner als die Originalgröße der Daten ist. Ist das nicht der Fall, wird die Kompression später auch nicht angewandt, da sie kontraproduktiv wäre und viel Rechenzeit kostet. Dies ist bei hochentropen Datensätzen der Fall, die schwer komprimierbar sind und der Overhead des LZ4-Blockformats größer als der Kompressionseffekt ist.

Abbildung 3.1 zeigt das Klassendiagramm der Encoding-Klassenhierarchie. `encodedSize` ist jeweils die Methode, die die Größe der komprimierten Daten berechnet. Die Klasse `LZ4Encoding` implementiert alles für die LZ4-Kompression mit der Ausnahme, dass der Kompressor, der in `encode` verwendet wird, von der genauen Subklasse abhängt. Das Diagramm ist von der Java-Implementierung abgeleitet. Die Datentypen besitzen in C++ ein entsprechendes Pendant (z.B. `ByteBuffer`). In C++ existiert die `getCompressor`-Methode nicht. Dort wird eine Methode `compress` virtualisiert, die die Kompression vornimmt.

Die Methode `encodedSize` des `PrepassEncoding` komprimiert die übergebenen Daten zunächst mittels `encode` in einen Buffer. Dieser ist höchstwahrscheinlich größer als die Daten an sich, da wir davor nicht wissen können wie groß diese werden. Die Methode `maxEncodedSize` liefert eine obere Schranke für die Größe komprimierter Daten auf Basis der Ausgangsgröße. Über die Stream-Position kann bestimmt werden, wie viele Daten `encode` tatsächlich geschrieben hat und damit wie groß die kodierten Daten sind. Ist Caching aktiviert, wird der Buffer gespeichert.

Die Reset-Operation ist bei einem `SimpleEncoding` leer implementiert, da dieses zustandsfrei ist. Bei einem `PrepassEncoding` löscht sie den Cache und setzt das omit-Flag zurück welches speichert, ob die Kompression ausbleiben soll. Zuletzt sei erwähnt, dass das `Encoding` seinen Namen speichert, mit dem die umgebende Coding-Restriction später serialisiert wird. Dies geschieht in beiden Implementierungen auf unterschiedliche Weise und wird dort jeweils genauer erläutert.

Die Implementierung des `BitvectorEncoding` ist sehr simpel. Beim Kodieren wird jedes gelesene Byte als Bit in den OutStream geschrieben. Das Schreiben eines Bits ist ein neu hinzugefügtes Feature und wird später genauer erläutert. Um zu Dekodieren, werden `numElements` viele Bits aus dem Stream gelesen und als Bytes in einem Buffer zurückgegeben. Dieser Parameter ist erforderlich, da beim Kodieren ein Padding mit Nullen erfolgt um volle Bytes zu schreiben. Daher ist die Anzahl der tatsächlichen Werte nicht eindeutig bestimmbar. `numElements` gibt an, wieviele bool'sche Werte gelesen werden.

Beim `LZ4Encoding` schreibt die `encode`-Methode zunächst die Größe der übergebenen Daten als v64 in den OutStream. Dies erlaubt eine schnellere Dekomprimierung, wie später noch erläutert wird. Danach werden die Daten in den hinter dem OutStream liegenden Buffer komprimiert. Welcher Kompressor genau verwendet wird hängt von der Implementierung von `getCompressor` ab, bzw. wird in C++ an `compress` delegiert. LZ4HC wird mit der höchsten Kompressionsstufe verwendet um den Unterschied beider Methoden deutlicher darstellen zu können.

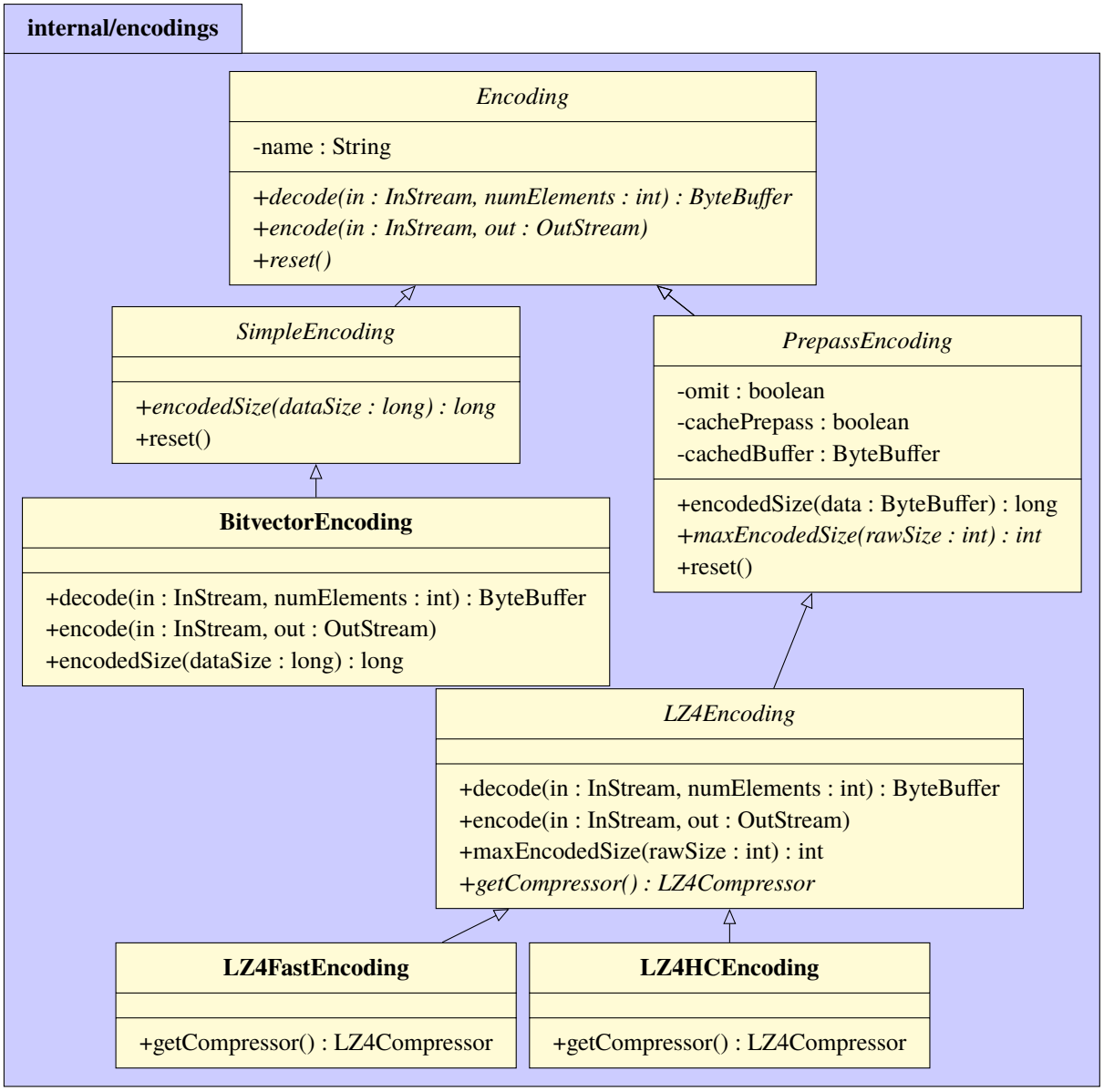


Abbildung 3.1.: Klassendiagramm der Encoding-Hierarchie

Die Dekompression liest zuerst einen v64. Dieser enthält die Größe der Originaldaten. Damit ist es möglich einen passenden Buffer zu erzeugen. Die Dekompression der Daten erfolgt dann in den Buffer hinter dem InStream. `maxEncodedSize` ist trivial, da die LZ4-Bibliotheken eine entsprechende Methode zur Verfügung stellt, die die obere Schranke berechnet. Auf diese muss nur noch die Größe des v64-Wertes, der noch zusätzlich die Originalgröße speichert, hinzugerechnet werden.

3.2.3. Felder

Bisher wurden die Methoden `osc` bzw. `obc` aufgerufen, um den Offset eines Feldes zu berechnen. Mit der Hinzunahme der Encodings und der Defaultwert-Optimierung ändert sich der Berechnungsprozess. Die eben erwähnten Methoden berechnen nur die Größe der rohen Felddaten. Bei Verwendung eines Encodings entspricht dies jedoch nicht der tatsächlichen Größe der geschriebenen Daten. Es werden zwei neue Methoden definiert mit den Namen `cosc/cobc` welche als Parameter jeweils einen Chunk vom entsprechenden Typ bekommen. Der Name ist ein Akronym für *Coded Offset Simple/Bulk Chunk* und übernimmt die Aufgabe der Offset-Berechnung. Der Ablauf der beiden Methoden ist derselbe, sie unterscheiden sich nur darin ob `osc` oder `obc` aufgerufen wird um die Rohdatengröße zu berechnen.

Im ersten Schritt wird mithilfe von `osc/obc` die Größe der Rohdaten ermittelt. Diese Funktionen übernehmen zusätzlich zur Offset-Berechnung nun auch die Aufgabe zu entscheiden, ob die Felddaten nur aus Defaultwerten bestehen. Das Ergebnis wird in einem bool'schen Feld der `FieldDeclaration` gespeichert. Es kann nicht einfach ein Offset von 0 berechnet werden, da `obc` durch mehrere `osc`-Aufrufe auf disjunkten Teilen des BulkChunks implementiert ist (Abschnitt 3.1.1). Würden ein einzelner solcher Aufruf Defaultwerte feststellen, setzt er den Offset auf 0, obwohl dies nicht impliziert dass der gesamte BulkChunk aus Defaultwerten besteht. Der Teilaufruf weiß an dieser Stelle also nicht ob eine Defaultwert-Optimierung stattfindet und berechnet den korrekten Offset. Erst wenn alle Teilaufrufe abgeschlossen und festgestellt wird, dass keiner das bool'sche Feld auf `false` gesetzt hat, kann die Optimierung vorgenommen werden.

Sollten nur Defaultwerte vorhanden sein, kann an dieser Stelle abgebrochen werden. Selbst wenn ein `Encoding` existiert, werden keine Daten von diesem Feld serialisiert. Daher wird auch der Offset auf 0 gesetzt. Dieser Wert reflektiert die tatsächliche Größe des Chunks. Daher funktioniert die Serialisierung mit diesem Wert weiterhin. Der Offset kann auch 0 werden, wenn keine Instanzen in einem Block gespeichert werden. Ein solches Feld würde wegen diesem Offset als Defaultwert-optimiert erkannt werden und die Instanzen mit Default-Werten belegt werden. Weil es aber keine Instanzen gibt, ist das Verhalten für solche Felder dann trotzdem korrekt. Die Anzahl der Instanzen wird im Block in der Typdefinition gespeichert. Daher ist es ohne Probleme möglich festzustellen, wie viele Werte genau wegoptimiert wurden.

Sollte kein `Encoding` vorhanden sein, ist die Offset-Berechnung hier beendet. `osc/obc` haben den korrekten Offset bereits gesetzt. Bei vorhandenem `Encoding` muss zunächst der berechnete Offset zwischengespeichert werden. Dies ist nötig um zu wissen wie viele Daten das Feld schreiben will, wenn wir die Felddaten in einen Buffer zum komprimieren schreiben wollen. Ist diese Größe bekannt, können wir diesen Buffer präzise allokieren. Das Speichern des berechneten Offsets erfolgt im Chunk, der dafür mit einem entsprechenden Feld erweitert wurde.

Bei einem `PrepassEncoding` muss nun der Prepass durchgeführt werden. Dazu allokieren wir einen Buffer der der originalen Größe der Felddaten entspricht. Über diesem Buffer erstellen wir einen `VirtualOutStream` und rufen `wsc` auf. Der Buffer, welcher jetzt die Felddaten enthält, kann dann dem `Encoding` zur Größenberechnung übergeben werden.

Nachdem der Prepass durchgeführt wurde, wird überprüft, ob die Kompression eine Verkleinerung der Daten bewirkt. Ist dem nicht so, kann das im `Encoding` markiert werden um die Omission-Mechanik zu aktivieren. Sie sorgt dafür, dass die originalen Daten gespeichert werden. Die Restriction bleibt allerdings trotzdem erhalten. Durch ein Byte am Anfang des Chunks wird

markiert, ob hier Originaldaten vorliegen, oder komprimierte Daten. Dieses Byte erhöht die Größe des Chunks um 1, und muss daher zum berechneten Offset hinzu addiert werden. Dies gilt nur für ein `PrepassEncoding`.

Handelt es sich um ein `SimpleEncoding`, ist das Vorgehen trivial. Die `maxEncodedSize`-Methode des `Encoding` kann die komprimierte Größe direkt berechnen. Das Erkennen von Defaultwerten wird im Codegenerator implementiert, da dort `osc` implementiert ist. `obc` delegiert nur an `osc` und muss nicht geändert werden. Felder die generisch gelesen werden, sind durch das `DistributedField` implementiert, welches Subklassen von `FieldDeclaration` sind. Die Implementierung dessen `osc`-Methode wird im spezifischen Teil behandelt.

Wird ein Chunk vorgefunden, welcher die Größe 0 hat, kann er nicht normal über `rsc/rbc` gelesen werden, da es keine Daten zu lesen gibt. Für diesen Zweck wurden die Methoden `rscd` und `rbcd` hinzugefügt (Read Simple-/Bulk-Chunk Default). Diese erwarten als Parameter jeweils Chunk-Informationen. Für `rscd` ist das ein SkillID-Intervall und für `rbcd` ein BulkChunk. Dies wird benötigt um zu wissen wie viele Werte in dem Chunk wegoptimiert wurden und für welche SkillIDs. Diese Methode füllt dann den State mit Standardwerten.

Für bekannte Felder ist `rbcd` wie `rbc` definiert und delegiert für jeden Block, für den der BulkChunk Werte speichert, das Lesen der Daten für diesen Block an einen `rscd`-Aufruf. `rscd` wird durch den generierten Code implementiert und wird in Abschnitt 3.7.3 erklärt. Die Implementierung für `DistributedField` wird in den Abschnitten der jeweiligen Sprachen gezeigt und erläutert.

3.2.4. Lesen

Für diesen Abschnitt sei folgendes definiert: Die Begriffe „Coding-Restriction“ und „Encoding“ schließen auch den Null-Pointer mit ein. Ein „anderes `Encoding`“ bedeutet also, dass überhaupt kein Encoding vorhanden sein kann.

Eine wichtige Änderung im Parser-Code, der durch die Klasse `FileParser` implementiert ist, ist das Speichern der Coding-Restriction bzw. des Encodings. Es wird hier davon ausgegangen, dass eine andere Coding-Restriction als die spezifizierte in einer .sf-Datei vorgefunden werden kann. Ist der enthaltene String ein bekanntes Encoding, soll die Implementierung dieses Feld lesen können. Bei einer Write-Operation wird allerdings die spezifizierte Restriction verwendet. Bei einer Append-Operation ist die Felddefinition nicht veränderbar und damit muss die vorgefundene Restriction weiterverwendet werden. Eine Ausnahme davon tritt ein, wenn das Feld nicht bekannt ist. Das bedeutet das Feld existiert nicht in der Spezifikation aus der der verwendete generierte Code erzeugt wurde. In diesem Fall ist also nichts über die Intention bezüglich dieses Feldes bekannt. Somit wird in dem Fall die in der .sf-Datei vorgefundene Restriction auch bei einer Write-Operation weiter verwendet. Ohne diese Ausnahme würde bei einer Write-Operation keine Coding-Restriction für unbekannte Felder serialisiert werden.

Aus dieser Forderung lässt sich folgendes ableiten: Eine gelesene Coding-Restriction muss bei einem bekannten Feld niemals gespeichert werden. Entweder es findet später eine Write-Operation statt, dann wird die spezifizierte Restriction verwendet. Diese wurde durch den generierten Code in die `FieldDeclaration` hinzugefügt. Andernfalls findet ein Append statt, dann steht bereits eine Coding-Restriction in der Datei. Das Lesen der Felddaten muss aber mit dem vorgefundenen

Encoding erfolgen. Daher wird die vorgefundene Restriction zwar nicht gespeichert, aber das enthaltene Encoding wird als das Encoding des Feldes gesetzt. Nur bei einem unbekannten Feld wird die vorgefunden Restriction der FieldDeclaration hinzugefügt wie oben erläutert.

Der finale Schritt ist das Lesen der Felddaten. Dabei wird jeder Chunk parallel gelesen. Sollte der Chunk eine Länge von 0 haben, handelt es sich um wegoptimierte Standardwerte, oder dieser Block enthält keine Instanzen des Typs der dieses Feld besitzt. In diesem Fall kann die Operation schnell beendet werden. Ein Aufruf von `rscd/rbcd` mit den Chunkinformationen setzt die jeweiligen Instanzen auf Standardwerte. In diesem Fall ist der Chunk nun abgearbeitet.

Bei normalen Werten geht es folgendermaßen weiter: Wenn kein Encoding vorhanden ist, können mittels `rsc/rbc` direkt die Daten aus der Datei in den State gelesen werden. Sollte allerdings ein Encoding vorhanden sein, muss der Daten-Chunk zunächst dekomprimiert werden. Handelt es sich um ein PrepassEncoding, ist es möglich dass keine Kompression vorgenommen wurde, da diese keine Verkleinerung der Daten bewirkt hätte. Das erste Byte als `boolean` interpretiert gibt Aufschluss darüber. Wurden die Daten wirklich komprimiert, liefert uns die `decode`-Methode des `Encoding` einen Buffer mit den rohen Felddaten, der dann `rsc/rbc` über einen Stream übergeben werden kann.

3.2.5. Schreiben

Der bisherige Ablauf des Schreibvorgangs wurde in Abschnitt 3.1.5 gezeigt. In der Vorbereitungsphase, in der die zu schreibenden Strings gesammelt werden, müssen wir die Namen der `Encoding`-Instanzen berücksichtigen. Dazu wird in jedem Feld des States nach einer CodingRestriction gesucht und der Name des zugrundeliegenden `Encoding` registriert.

Für die Offset-Berechnung werden nun die Methoden `cosc/cobc` aufgerufen. Zusätzlich muss bei jedem Feld vor der Offset-Berechnung das `Encoding` entpackt werden, welches in der Coding-Restriction liegt. Das bedeutet, dass dieses `Encoding` als das aktive `Encoding` der `FieldDeclaration` gesetzt wird. Bei einer Write-Operation ist dies immer der Fall, da wir die Felddefinition komplett neu schreiben und das `Encoding` aus der spezifizierten Restriction verwenden wollen. Bei einem Append ist dies nur der Fall wenn das Feld neu hinzugefügt wurde, da bei bereits in der .sf-Datei vorhandenen Feldern die Restrictions nicht überschrieben werden können. Bei einem unbekannten Feld ist die Coding-Restriction die, welche in der Datei vorgefunden wurde.

Im letzten Schritt werden die Felddaten in die .sf-Datei geschrieben. Jedes Feld wird dabei parallel serialisiert Hat der neue Chunk des Feldes, welches geschrieben wird, eine Länge von 0 (Defaultwerte), muss nichts getan werden. Ist kein `Encoding` gesetzt, kann der `wsc/wbc`-Aufruf direkt erfolgen. Dieser schreibt in den Chunk-Bereich über einen zur Verfügung stehenden OutStream. Bei einem `PrepassEncoding` müssen wir zuerst feststellen ob die Kompression überhaupt stattfinden soll. Dies wurde während der Offset-Berechnung getan und ist im `Encoding` gespeichert. Um diese Information zu serialisieren, schreiben wir einen bool'schen Wert der `true` ist genau dann, wenn wir auf die Kompression verzichten. Soll die Kompression stattfinden, überprüfen wir ob die Daten durch das `PrepassEncoding` bereits zwischengespeichert wurden. In diesem Fall wird dieser Cache einfach in die Datei geschrieben und der Schreibvorgang ist beendet. Andernfalls lassen wir `wsc/wbc` in einen allokierten Buffer schreiben. Die Größe des Buffers ist die Größe der rohen Felddaten, die durch die Offset-Berechnung in den Chunk-Informationen zur Verfügung steht. Selbiges geschieht bei einem SimpleEncoding.

Nach dem Aufruf von `wsc/wbc` muss der Buffer komprimiert werden und dieser kann dann in die Datei geschrieben werden. Abschließend wird der State des Encodings zurückgesetzt. Das bedeutet der Cache und das Omit-Flag werden gelöscht. So ist das `Encoding` über weitere Serialisierungen aus dem gleichen SKilL-State gültig.

3.3. Bestehende Java-Implementierung

Die Java-Implementierung besteht aus 3 verschiedenen Komponenten: Die Java-Laufzeitbibliothek (*javaCommon*) enthält die bisher besprochene Implementierungslogik und die Erweiterungen. Die JVM-Bibliothek (*jvmCommon*) enthält Code den auch andere Sprachen wie z.B. Scala nutzen. Er enthält nur die später besprochene Stream API. Die beiden Bibliotheken werden von einem Programm welches SKilL verwendet eingebunden. Der dritte Teil ist der Codegenerator. Die Funktion dessen wurde in Abschnitt 3.7 genauer erklärt. Die Änderungen im Codegenerator werden später separat gezeigt.

3.3.1. Restrictions

Die `FieldRestriction` in Java besitzt die Methode `check(value)`, mit der überprüft werden kann ob die Restriction eingehalten wird. Bei einer Range-Restriction wird z.B. überprüft ob der übergebene Wert im spezifizierten Bereich liegen. Beim Lesen einer .sf-Datei kann anhand der gelesenen ID auf den Typ der Restriction geschlossen und eine entsprechende Instanz der Unterklasse erstellt werden. Folgende Restrictions sind in der Java-Implementierung implementiert: NonNull, Range und Coding. Jedoch ist Coding leer implementiert und hat nicht den spezifizierten Effekt. Bei nicht implementierten Restrictions wie z.B. Default werden noch die mitgelieferten Parameter eingelesen (der Default-Wert), damit der Lese-Stream an der richtigen Position weiterlesen kann. Es wird jedoch keine Restriction erzeugt. Die Serialisierung von Restrictions war vor dieser Arbeit noch nicht implementiert. Wie zu sehen ist, unterscheidet Java bisher nicht zwischen prüfbaren Restrictions und nicht-prüfbaren Restrictions.

3.3.2. Blocks und Chunks

Der Einfachkeit halber erstellt die Java-Implementierung beim Lesen nur BulkChunks, wenn der Chunk auch wirklich für mehrere Blöcke Daten enthält. Ein Chunk der im selben Block liegt wie die Erstdefinition des Feldes wird andernfalls als SimpleChunk gelesen, da die zusätzliche Information des BulkChunk nicht benötigt wird.

3.3.3. Felder

Die in Abschnitt 3.1.1 vorgestellte Klassenhierarchie erweitert Java um die Klasse `KnownDataField` als Unterklasse von `FieldDeclaration`. Diese Klasse dient als Basisklasse für die Felder im generierten Code. Als fortsetzendes Beispiel sei in Listing 3.1 der Code gezeigt, der für das Lesen des `matriculationNumber`-Feldes des `Student`-Typs der in Abschnitt 2.1 als Beispiel angeführten Spezifikation generiert wird. `P2` ist dabei die Unterklasse von `StoragePool`, die für `Student` erzeugt

Listing 3.1 beispielhaft generierter Code für das Lesen eines variablen Integer-Typs

```
1 @Override
2 protected final void rsc(int i, final int h, MappedInStream in) {
3         final de.test.Student[] d = ((P2) owner).data();
4         for (; i != h; i++) {
5                 d[i].matriculationNumber = in.v64();
6         }
7
8 }
```

wird. Der Typ des Daten-Arrays ist die Repräsentation einer Objektinstanz von `Student` im Java-Typsystem. Siehe dazu auch Abschnitt 2.2 und Abschnitt 2.6. `i` und `h` beschreiben das Intervall der SkillIDs der Objektinstanzen, deren Feldwerte verarbeitet werden sollen.

3.3.4. Streams

Die Bibliothek *jvmCommon* bietet eine eigene Stream API, die von der Implementierung der Java-Laufzeitbibliothek genutzt wird. Die Schnittstelle der Streams bietet Methoden um SKilL-spezifische Datentypen zu behandeln. Zum Beispiel kann damit ein V64-Wert serialisiert werden. Dieser verwendet eine von SKilL definierte Kodierung um eine variable Länge zu ermöglichen. Das Stream-Interface implementiert die Umwandlung einer Java Long-Variable zur binären Darstellung in der .sf-Datei und umgekehrt. Die Basis der Streams bilden der `InStream` und der `OutStream`. Beide definieren die grundlegenden Operationen auf den SKilL-Datentypen und die Implementierung über einen `ByteBuffer`. Beide Streams besitzen jeweils eine Unterklasse, den `Mapped[In|Out]Stream` und den `File[Input|Output]Stream`. Der File-Stream ist beim Lesen eine in den Hauptspeicher gemappte Datei und beim Schreiben ein Buffer, der in eine Datei geschrieben werden kann (flush). Die File-Streams bieten die Möglichkeit aus einem Teilbereich der Datei einen Mapped-Stream zu erzeugen. Dies wird in der SKilL-Implementierung z.B. verwendet um beim Lesen Streams für die einzelnen Feld-Chunks zu erzeugen, welche parallel gelesen werden. Beim Schreiben erfolgt ein Mapping erst dann wenn ein `MappedOutStream` aus dem `FileOutputStream` gemappt wird. Beim parallelisierten Schreiben der Felddaten werden so die Bereiche der einzelnen Chunks in den Speicher abgebildet.

3.3.5. Index-Wertebereich

Wie in [Fel17, p. 60] erläutert, erlaubt Java nur Array-Indizes bis ungefähr 2^{31}. Daher deklariert SKilL es als gültiges Verhalten, wenn die Serialisierung bei Arrays, welche größer als 2^{30} sind, scheitert. Dies ist z.B. der Fall wenn mehr als 2^{30} SkillIDs vergeben werden müssen. Da Java keine größeren Indizes erlaubt, passt jeder Index in einen `int`. Dennoch wird in der bisherigen Implementierung oft `long` verwendet. An einigen Stellen wird aber, auch in den Erweiterungen durch diese Arbeit, `int` verwendet. Dies stellt, wenn die Semantik der Variable in den Gültigkeitsbereich dieser Beschränkung fällt, keine Einschränkung des Wertebereiches dar.

3.4. Java-Implementierung der Kompression von Felddaten

Hier wird die Umsetzung der in Abschnitt 3.2 vorgestellten Konzepte und Prozeduren in Java gezeigt.

3.4.1. Restrictions

Da die Coding-Restriction in der Java-Bibliothek nur eine leere Implementierung besaß, waren nur NonNull und Range funktionsfähig. Mit der korrekten Realisierung einer Coding-Restriction kommt eine Restriction hinzu, welche keiner Überprüfung bedarf (näheres siehe Abschnitt 2.3). Daher wurde die Klassenhierarchie geändert, um diesen Zustand korrekt darzustellen. Eine Subklasse `CheckableRestriction` beinhaltet nun die `check(value)`-Methode und ist Superklasse der Range- und NonNull-Restrictions. Mittels einer dynamischen Typprüfung wird bei der Überprüfung des States dann entschieden welche Restriction überprüft werden muss.

Jede Restriction ist dafür verantwortlich, sich zu schreiben. Dafür definiert die `FieldRestriction` die Methode `write(OutStream, StringPool)`, welche die Restriction-ID in den `OutStream` schreibt. Einzelne Restrictions können diese dann überschreiben. Der `StringPool` wird von der Coding-Restriction benötigt um ihren Namen in die Datei zu schreiben. Über den Pool ist es möglich die ID dieses Strings herauszufinden, um ihn referenzieren zu können. Die Serialisierungsmechanik macht dem Pool die Namen der existierenden Coding-Restrictions bekannt, damit diese auch in den Stringblock geschrieben werden.

Abbildung 3.2 zeigt die Klassenhierarchie der Restrictions als UML-Diagramm. Der Einfachheit halber wurden triviale Getter- und Setter-Methoden und überschriebene virtuelle Methoden weggelassen. `Range` ist aufgrund von Beschränkungen des generischen Typ-Systems in Java eigentlich für jeden (Wert-)Feldtypen separat definiert.

3.4.2. Streams

In dieser Arbeit wurde die Stream API um zwei wichtige Features ergänzt. Es gibt nun die Möglichkeit einzelne Bits zu serialisieren. Dies ist über eine Methode namens `i1` möglich. Dies geschieht nur auf der logischen Ebene. Es muss dafür das nächste Byte zwischengespeichert werden bzw. ein temporäres Byte beim Schreiben verwendet werden, da einzelne Bits nicht in eine Datei geschrieben werden können. Bei einem OutStream muss der Benutzer manuell eine Flush-Operation durchführen, falls keine durch 8 teilbare Bit-Anzahl geschrieben wurde. Dann wird das temporäre Byte mit abschließenden Nullen geschrieben. In Listing A.1 ist die Implementierung für das Lesen (InStream) zu sehen. Die Implementierung speichert auch die Position des Bytes, das sie zuletzt gecached hat. Dadruch kann sie beim nächsten Aufruf erkennen, ob dazwischen weitere Bytes über eine andere Operation gelesen wurden. Das invalidiert das zwischengespeicherte Byte. Die Invalidierung wird nicht durch die anderen Operationen ausgeführt, da der dadurch entstehende Overhead durch die seltene Verwendung der i1-Operation nur bei Bitvector-Encodings überwiegt.

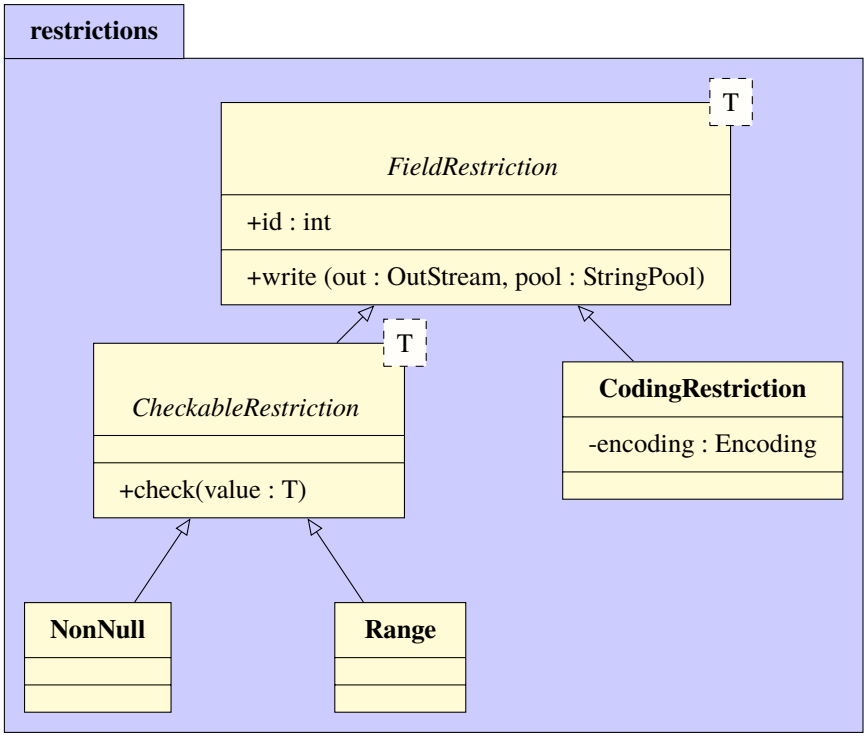


Abbildung 3.2.: Klassendiagramm der Restriction-Hierarchie

Als zweite Änderung wurde eine neue Klasse eingeführt, der `VirtualOutStream`. Er ist eine Überklasse des `MappedOutStream` und eliminiert die Notwendigkeit eines `MappedByteBuffer` als zugrundeliegenden Buffer des Streams, welchen der `MappedOutStream` verwendet. Somit kann auch in einen rein im Hauptspeicher liegenden Buffer geschrieben werden, der keinen gemappten Bereich aus einer Datei darstellt. Die in Abschnitt 3.1.1 vorgestellten Lese- und Schreibmethoden verwenden die Mapped-Streams als Parameter. Durch die Realisierung der Encodings besteht die Notwendigkeit, Felddaten auch in den Hauptspeicher schreiben zu lassen, ohne dabei die Implementierung der Serialisierungslogik des Feldes zu ändern. Die Virtualisierung des Mapped-Streams macht die Serialisierung unabhängig vom tatsächlichen Ziel des Streams. Dadurch wird das benötigte Verhalten über eine Änderung der Methodensignatur realisierbar.

Es sei erwähnt, dass der `VirtualOutStream` eine Generalisierung des `MappedOutStream` darstellt, da er die Buffer-Implementierung nicht spezifiziert. Der Typ des Buffers und damit das Ziel des Streams wird bei der Erstellung angegeben. Es ist keine Unterklasse nötig, die auf einen `DirectByteBuffer` spezialisiert. Der `MappedOutStream` als Unterklasse ist nötig, da bei einer Close-Operation der gemappte Bereich wieder in die Datei zurückgeschrieben werden muss. Für den `MappedInStream` ist diese Änderung nicht nötig, da dieser keinen `MappedByteBuffer` erzwingt.

3.4.3. Encodings

Listing A.2 zeigt die Implementierung der `encodedSize`-Methode des `PrepassEncoding`. Dort findet auch der `VirtualOutStream` Verwendung. Mit ihm können die Felddaten in einen Buffer geschrieben werden, ohne die Logik der `wsc/wbc`-Methode zu ändern. Das `Encoding` überschreibt zusätzlich die Methode `toString`, welche den Namen des Encodings für die Serialisierung der Coding-Restriction zurückgibt.

Die Klasse `BitvectorEncoding` ist ein Singleton, da sie zustandsfrei ist und daher mehrere Instanzen unnötig sind. `encodedSize` wird nicht näher erläutert, da die Formel dazu in Abschnitt 3.2.2 schon gezeigt wurde. Listing A.3 zeigt die Implementierung von `encode`. `decode` ist in Listing A.4 zu finden. Mit der neuen `i1`-Methode des Streams kann ein einzelnes Bit geschrieben oder gelesen werden. Beim Schreiben muss ein Flush erfolgen, damit ein eventuell unvollständiges Byte mit einem Null-Padding geschrieben wird. `i1` gibt beim Lesen ein Byte zurück welches mit SKilL-`bool` binärkompatibel ist und kann so ohne Konvertierung direkt geschrieben werden.

Die Implementierungsdetails der LZ4-Encodings werden hier genauer gezeigt. Es wird eine quelloffene LZ4-Implementierung für Java verwendet [1]. Diese unterstützt den normalen LZ4-Modus, sowie das hier auch unterstützte LZ4HC. Die Implementierung der `encode`-Methode ist in Listing A.5 zu sehen. Welcher Kompressor genau verwendet wird hängt von der Implementierung von `getCompressor` ab, und ist entweder der `LZ4FastCompressor`, oder der `LZ4HCCompressor`.

Wie die Daten dekomprimiert werden, zeigt Listing A.6. Der erste v64 enthält die Größe der originalen Daten. Damit ist es möglich einen passenden Buffer zu erzeugen. Diese Information erlaubt uns außerdem einen `FastDecompressor` zu verwenden. Wie in der Dokumentation der Bibliothek zu lesen ist, ist dieser schneller als der `SafeDecompressor`.

3.4.4. Felder

Die Signatur aller in Abschnitt 3.1.1 erklärten Methoden wurde so geändert, dass statt einem `MappedOutStream` ein `VirtualOutStream` erwartet wird. Der Vorteil davon ist in Abschnitt 3.4.2 beschrieben. Die genaue Umsetzung für die generierten Methoden von Subklassen ist in Abschnitt 3.7.2 beschrieben. Die neue Offset-Berechnung wurde bereits in Abschnitt 3.1.1 beschrieben. Listing A.7 zeigt die Implementierung von `cosc` in Java. `cobc` ist gleich implementiert, enthält aber einen `wbc`-Aufruf.

Die Implementierung für `DistributedField` ist in Listing A.8 zu finden. Der Offset für einen Wert wird über `FieldType.singleOffset(value)` berechnet. Wir gehen zunächst davon aus, dass es nur Defaultwerte gibt. Für jeden Wert muss jetzt festgestellt werden, ob dieser ein Standardwert ist. Falls nicht, kann `defaults` auf `false` gesetzt werden. Wir können dann zwar weitere Vergleiche auf Defaultwerte einstellen, jedoch die Berechnung nicht komplett abbrechen. Dies liegt daran, dass der Offset-Wert komplett berechnet werden, und daher über alle Instanzen iteriert werden muss. Der berechnete Wert wird auf den Offset addiert. Sind hier nicht nur Defaultwerte vorhanden, wird das Flag `defaultsOnly` der `FieldDeclaration` auf `false` gesetzt.

[1] https://github.com/lz4/lz4-java

Die Implementierung der neu eingeführten Methoden `rscd/rbcd` zeigt Listing A.9. Das `DistributedField` speichert die Felddaten selbst und nicht im `StoragePool`. `rscd` fügt einen Eintrag in der `HashMap` des `DistributedField` hinzu. Dieser hat als Key das Objekt im Daten-Array des `StoragePool` und als Wert den Standardwert des Typ des Feldes. Jeder `FieldType` implementiert die Methode `getDefaultValue`. Die Implementierung einer funktionierenden Defaultwert-Logik ist nicht Thema dieser Arbeit, aber trotzdem nötig für die Defaultwert-Optimierung. Daher wurde eine behelfsmäßige Lösung implementiert, welche mit Nullwerten funktioniert. Dazu gehört unter anderem diese Abfrage des Standardwertes eines Typs. Die hier implementierte Defaultwert-Logik funktioniert nicht für Container-Typen, da eine Nullreferenz und ein leerer Container binär identisch sind. Es wird jedoch immer ein leerer Container gelesen, der keinen Defaultwert darstellt. Die hier vorgestellte Defaultwert-Optimierung funktioniert aber wenn eine robuste Defaultwert-Logik implementiert wird. `rbcd` ist ähnlich implementiert, hier erfolgt nur eine kompliziertere Berechnung der SkillIDs, da die Informationen aus mehreren Blöcken abgefragt werden müssen.

Listing A.10 zeigt einen Ausschnitt aus dem Code, der einen Chunk aus einer .sf-Datei liest. Der Ablauf wurde bereits erläutert.

3.4.5. Serialisierung

Der Java-Code für das Lesen und Schreiben eines Chunks mit Encodings ist in Listing A.10 und Listing A.11 zu finden.

3.5. Bestehende C++-Implementierung

Die C++-Implementierung setzt sich aus der C++-Laufzeitbibliothek (*cppCommon*) und dem Codegenerator für C++ zusammen. Ein wichtiger Unterschied im Vergleich zur Java-Implementierung ist, dass C++ keine Append-Operationen unterstützt.

3.5.1. Restrictions

In der C++-Implementierung existiert bereits eine Unterscheidung zwischen prüfbaren und nichtprüfbaren Restrictions. Prüfbare Restriction sind durch eine Unterklasse `CheckableRestriction` von `FieldRestriction` realisiert. Die Klasse für die Coding-Restriction heißt `Coding` und ist bisher leer implementiert.

3.5.2. Strings

Um in C++ mit Strings umzugehen, die serialisiert werden müssen, wurde eine eigene Klasse geschaffen. `string_t` ist eine Unterklasse von `std::string`. Diese Strings enthalten eine zusätzliche ID und können nicht vom Benutzer direkt erzeugt werden. Ein `StringPool` verwaltet alle Strings. Über diesen kann ein neuer String erzeugt werden. Falls der String bereits existiert wird er zurückgegeben um Duplikate zu vermeiden. Bei der Serialisierung werden allen String-Instanzen korrekte IDs zugewiesen, sodass ein String serialisiert werden kann, indem die ID geschrieben wird.

Listing 3.2 beispielhaft generierte rsc-Methode

```
1 void ::test::internal::KnownField_Student_matriculationNumber::rsc(::skill::SKilLID i,
2 const ::skill::SKilLID h, ::skill::streams::MappedInStream *in) {
3         auto d = ((StudentPool *) owner)->data;
4         for (; i != h; i++) {
5                 d[i]->_matriculationNumber = in->v64();
6         }
7 }
```

Im Gegensatz zur Java-Implementierung kann der String so serialisiert werden ohne beim StringPool die ID abfragen zu müssen. Da keine Kopien von Strings erlaubt sind und dieser (außer vom Pool) auch nicht verändert werden soll, erfolgt der Zugriff über einen Pointer auf einen konstanten String.

3.5.3. Felder

Die Funktionen `osc` und `wsc` benötigen in C++ keine Version für BulkChunks, da solche beim Schreiben nicht vorhanden sein können. Die SimpleChunk-Methoden implementieren jedoch eine veraltete Fallunterscheidung für Simple- und BulkChunks. Die Implementierungen dieser Arbeit wurden so gestaltet, dass sie sich auch im nicht auftretenden Fall eines BulkChunks korrekt verhalten. Die `rsc`-Methode für das Feld `matriculationNumber` des Typs `Student` in dem in Abschnitt 2.1 angeführten Beispiel, ist in Listing 3.2 zu sehen. `SKilLID` ist ein Typedef für einen integralen Typ.

3.5.4. Streams

In der C++-Implementierung ist ein Stream ein Bereich im Adressraum des Programms. Dazu existiert eine Basisklasse `Stream`. Die jeweiligen Stream-Unterklassen implementieren Methoden um SKilL-Typen schreiben zu können. Für den Input gibt es `InStream`, der diese Operationen definiert, und dessen Unterklassen `FileInputStream` und `MappedInStream`. Der `FileInputStream` bildet eine Datei in den Adressraum des Programms ab und stellt diese dar. Der `MappedInStream` kann aus dem `FileInputStream` erstellt werden und operiert über einem Teilbereich der Datei. Für den Output existieren jeweils der `FileOutputStream` und der `MappedOutStream` (ohne eine Klasse `OutStream`). Der `FileOutputStream` stellt einen Buffer dar, der regelmäßig in eine Datei schreibt. Das Mappen aus Teilbereichen der Datei findet statt, wenn ein `MappedOutStream` erstellt wird.

3.5.5. Box

Ein Feld kann je nach Typ verschiedene Werte im State annehmen. Integer, Gleitkommazahlen, oder ein Zeiger auf ein anderes Objekt bzw. eine Datenstruktur wie eine Liste. Um den Typ des Wertes eines Feldes zu generalisieren, wird eine `Box` verwendet. Diese ist eine `union`, welche jeden Möglichen Feldwert-Typ enthält. Die Zugriffsfunktionen `box` und `unbox` sind für jeden Typ und auch mit Typparameter überladen. So kann z.B. die Range-Restriction, welche eine Template-Klasse ist

und als Typparameter den Typ des Feldwertes für die Intervallgrenzen enthält in ihrer `check`-Methode eine `Box` annehmen. `unbox` löst mithilfe der Template-Funktion je nach Typparameter so auf, dass der Wert der `union` korrekt abgerufen wird.

3.6. C++-Implementierung der Kompression von Felddaten

Hier wird die Umsetzung der in Abschnitt 3.2 vorgestellten Konzepte und Prozeduren in C++ gezeigt.

3.6.1. Restrictions

Die Serialisierung von Restrictions war hier ebenfalls nicht implementiert. Wie auch in Java schreibt eine Restriction sich selber. Dafür existiert die virtuelle Methode `write(FileOutStream&, FieldType*)`. Der `FieldType` wird von der Range-Restriction benötigt um festzustellen welchen Typ das Feld hat, und welchen Typ damit die Intervallgrenzen der Restriction haben.

3.6.2. Streams

Die Stream API von C++ kann nun einzelne Bits lesen/schreiben. Da es für Out-Streams keine gemeinsame Oberklasse gibt, müsste diese Funktion sowohl für `MappedOutStream` als auch für `FileOutputStream` implementiert werden. Sie wurde jedoch neben dem `InStream` nur für den `MappedOutStream` implementiert, da sie nur damit verwendet wird. Die Implementierung entspricht logisch der in Abschnitt 3.4.2 vorgestellten Java-Implementierung. Die Ausnahme stellt die Implementierung im `MappedOutStream` dar. Während wir in Java einzelne Bytes oder Worte in den Buffer schreiben mussten, können wir in C++ auf den Inhalten des Streams dynamisch mittels Pointerarithmetik operieren. Hier liegt nämlich jedes Byte adressierbar im Speicher. Somit ist ein temporäres Byte unnötig um mehrere i1-Operationen zu sammeln. Wir können jedes Bit sukzessive in den abgebildeten Speicher reinmaskieren. Listing A.12 zeigt die Implementierung. Da beim Beginn eines Bytes die Position des Streams inkrementiert wird, schreiben andere Operationen erst in das nächste Byte. Verglichen mit der Java-Umsetzung befindet sich der C++-Stream nach jeder i1-Operation in demselben Status wie der Java-Stream nach eine Flush-Operation. Das eliminiert die Notwendigkeit einer solchen in C++.

Ebenfalls ist die Implementierung eines `VirtualOutStream` wie in Java nicht nötig. Der `MappedOutStream` ist nicht von einer Datei abhängig. Es ist möglich einen `MappedOutStream` auf einem beliebigen Bereich im Adressraum zu erstellen. Map- und Unmap-Operationen finden im `FileOutputStream` statt und werden bei manuellem Erstellen eines `MappedOutStream` komplett umgangen.

3.6.3. Encodings

Eine Besonderheit der C++-Encodings sind ihre Namen. Jedes `Encoding` enthält einen verwalteten String (s. Abschnitt 3.5.2). Wird das `Encoding` durch eine gelesene Coding-Restriction aus einer .sf-Datei erstellt, existiert der verwaltete String beim Lesen schon und wird dem `Encoding` zugewiesen. Wenn die Coding-Restriction mit dem `Encoding` allerdings in dem generierten Code erstellt wird, existiert der verwaltete String zunächst nicht. Erst beim Schreiben wird dieser dem `Encoding` zugewiesen, wenn alle Strings registriert werden. Der StringPool weiß wie das `Encoding` heißt, weil ein C-String abgefragt wird. Dann kann die besitzende Coding-Restrictions sich auch schreiben, da sie über den verwalteten String seine ID schreiben kann.

Eine Problemstellung in C++ ist die Speicherverwaltung, da jeder Buffer wieder gelöscht werden muss und kein Garbage Collector vorhanden ist. `encode` bekommt nur zwei Streams übergeben. Die Speicherverwaltung übernimmt hier also offensichtlich der Aufrufer, genauso wie bei `encodedSize`. `decode` erzeugt einen Buffer mit den dekodierten gelesenen Daten und gibt ihn zurück. Dafür wird ein Unique-Pointer verwendet, damit der Buffer seine Zerstörung selber übernimmt. So wird verhindert, dass ein Aufrufer mangels sichtbarer Allokation dies vergisst.

Die Bitvector-Klasse wird nicht mehr als Singleton realisiert, da sie mit dem verwalteten String einen State besitzt. Es wird die offizielle C-Bibliothek des Erfinders von LZ4 verwendet [2]. Im Java-Abschnitt wurde angeführt, dass es zwei Möglichkeiten gibt, Daten mit der LZ4-Bibliothek zu dekomprimieren. Im Gegensatz zur Java-Bibliothek dokumentiert die C-Bibliothek dass der `SafeDecompressor` gleich schnell bis schneller als der `FastDecompressor` sein kann. Die Verwendung des letzteren wird nicht mehr empfohlen. Daher verwendet die C++-Implementierung den `SafeDecompressor`. Die Implementierung der `Encoding`-Methoden für C++ befindet sich in Listing A.13, Listing A.14, Listing A.15, Listing A.16 und Listing A.17. Da die C-LZ4-Implementierung keine Compressor-Klassen besitzt, sondern nur unterschiedliche Methoden, wird die Abstraktion der Komprimierung dadurch realisiert, dass es eine virtuelle `compress`-Methode gibt, die von LZ4 und LZ4HC unterschiedlich implementiert ist.

3.6.4. Felder

Die Implementierung von `cosc` ist in Listing A.18 zu finden. `cobc` existiert nicht, da keine BulkChunks geschrieben werden können. In Listing A.19 sieht man, wie die Default-Werte im `DistributedField` erkannt werden. Bei jedem Schleifendurchlauf wird überprüft ob die Instanz einen Defaultwert hat. Der Defaultwert kann hier verallgemeinert werden mit einer Box die den Wert 0 hat. Das ist binärkompatibel zu allen Standardwerten. Stellen wir fest dass es kein Standardwert ist, setzen wir das Flag auf `true` und müssen die Überprüfung nicht mehr weiter durchführen. Sind allerdings nur Defaultwerte vorhanden können wir 0 zurückgeben. Dies ist in C++ möglich, da der Fall nicht auftreten kann, dass `osc` einen Teilbereich eines Chunks überprüft wie in der Java-Implementierung von `obc`.

[2] https://github.com/lz4/lz4

3.6.5. Serialisierung

In Listing A.20 ist ein Ausschnitt aus dem Code zu sehen, der einen Chunk liest. Wie ein Chunk serialisiert wird, ist in Listing A.21 zu sehen. Der Ablauf entspricht dem, der in Abschnitt 3.1.5 beschrieben ist.

3.7. Codegenerator

Zusammen mit den bisher vorgestellten Änderungen an den Implementierungen müssen einige Änderungen am Codegenerator vorgenommen werden. Der hier relevante Teil ist in Scala implementiert [Fel18, p. 124]. Einen Überblick über dessen Implementierung geht über den Rahmen dieser Arbeit hinaus, es wird nur auf die Stellen eingegangen an denen Änderungen erfolgen. Diese finden ausschließlich im `FieldDeclarationMaker` statt, welcher Subklassen von `KnownDataField` (Java) bzw. `FieldDeclaration` (C++) erstellt und implementiert. Das Projekt wurde zwar um einige Tests zu den neuen Features erweitert, auf diese wird allerdings nicht weiter eingegangen, da sie nur zur Verifizierung der Funktionalität der vorgestellten Änderungen an der gesamten Implementierung dienen.

3.7.1. Generierung spezifizierter Restrictions

Wird bei einem Feld eine Restriction spezifiziert, so soll diese im Konstruktor der generierten Feld-Klasse in das Restriction-Set hinzugefügt werden. Restrictions werden zwar auch aus einer .sf-Datei mit existierenden Daten gelesen, jedoch können die dort serialisierten Restrictions abweichen.

Die Generierung dieser Befehle erfolgt während der Generierung des Konstruktors für die Unterklasse die das Feld repräsentiert und ist in Listing A.22 für Java und in Listing A.23 für C++ zu sehen. Für jede Restriction des Feldes wird ein Befehl erzeugt. `makeRestriction(fieldType, restriction)` bildet die Restriction auf einen Konstruktor- / Factoryaufruf ab. Wie die Coding-Restrictions erzeugt werden, ist in einem Ausschnitt aus `makeRestriction` im selben Listing zu sehen. Der Rückgabewert ist eine `Option`. Diese enthält entweder die Erzeugung der Restriction, oder ist leer wenn die Restriction nicht unterstützt wird. Die erzeugten Aufrufe müssen nur noch in ein `addRestriction()` eingefügt werden. Es werden ebenfalls entsprechende Includes bzw. Imports erzeugt um die Encodings einzubinden.

3.7.2. Umstellung auf VirtualOutStream

Da die Signatur von `wsc/wbc` geändert wurde, um statt einem MappedInStream einen VirtualOutStream zu empfangen, müssen die Überschreibungen der Methode `wsc` (`wbc` wird nicht überschrieben) im generierten Code ebenfalls mit veränderter Signatur erzeugt werden. Diese Änderung ist trivial da die Signatur statisch ist und nur der Typ des Parameters im String der Signatur geändert werden muss. Diese Änderung findet nur für Java statt.

Listing 3.3 Generierung der rscd-Methode

```
s""" ...
@Override
protected final void rscd(int i, final int h) {
        final ${mapType(t.getBaseType)}[] d = ((${access(t.getBaseType)}) owner.basePool).data();
        for(; i != h; i++) {
                $fieldAccess = ${defaultValue(f)};
        }
}
... """
```

Listing 3.4 Generierung der rscd-Methode

```
s""" ...
void $fieldName::rscd(::skill::SKilLID i, const ::skill::SKilLID h) {${
        if(f.isConstant) "\n        // reading constants is O(0)"
        else s"""
        auto d = ((${storagePool(t)} *) owner)->data;
        for(; i != h; i++) {
                $accessI = ${defaultValue(f)};
        }
        """
}}
... """
```

3.7.3. Lesen von Default-Werten

Die Methode `rscd` muss auch im generierten Code implementiert werden (`rbcd` ist in der Laufzeitbibliothek implementiert). Die Implementierung für gewöhnliche Felder wird in Listing 3.3 (Java) und Listing 3.4 (C++) gezeigt. Bei einem konstanten Feld wird die Methode leer implementiert, da das Feld immer den gleichen Wert hat. Auf konstante Felder wird nicht näher eingegangen. Zunächst wird eine lokale Variable für das Daten-Array des besitzenden `StoragePool` erzeugt. Bei Java gibt `mapType` den Namen des Typs des Arrays zurück. Dabei handelt es sich um die Repräsentation des Basistyps der Hierarchie in Java. Mit `access` wird der Name der Klasse des `BasePool` für den Cast generiert. Von diesem wird dann das Daten-Array abgerufen. Bei C++ kann man mit `auto` den Typ automatisch bestimmen lassen. `storagePool` generiert den Typ des besitzenden Pools.

Wir können nun für jede Instanz mit einer for-Schleife dem Eintrag im Array einen Standardwert zuweisen. `fieldAccess` bzw. `accessI` erzeugt einen indizierten Zugriff auf das Objekt der Instanz im Daten-Array und greift auf das Feld der Instanz zu das dem hier implementierten Feld entspricht. Diesem Feld weisen wir einen Standardwert zu, der von `defaultValue` generiert wird. Diese Methode erzeugt für Werttypen den Wert `0`, für Referenztypen und Container den Wert `null` und für bool'sche Werte den Wert `false`.

3.7.4. Offset-Berechnung

Abschließend muss die Generierung von `osc` geändert werden, da diese Methode feststellen muss, ob die Daten des Chunks nur aus Standardwerten bestehen. Listing 3.5 enthält die Erzeugung der `osc`-Methode für Java und Listing 3.6 für C++. `offsetCode` generiert ein Paar. Das erste Element ist Code, der die korrekte Größe der Felddaten berechnet. Das zweite Element gibt an, ob es sich dabei um schnellen Offset-Code handelt. Für manche Datentypen kann die Größe ohne for-Schleife berechnet werden, so ist z.B. der Offset eines i8 in Byte die Anzahl der Instanzen. In C++ steht dafür ein `bool` zur Verfügung (`fastOffset`). Bei der C++-Implementierung wurde zugunsten der Übersichtlichkeit auf die unnötige Fallunterscheidung zwischen SimpleChunks und BulkChunks verzichtet. Nur die Implementierung für SimpleChunks ist zu sehen.

Bei schnellem Offset-Code können wir einiges weglassen. Zunächst wird die bereits aus `rscd` bekannte Variable für das Daten-Array generiert. Um festzustellen, ob nur Standardwerte vorhanden sind, brauchen wir trotzdem eine for-Schleife. Diese läuft in Java mit dem Index `j`, da `i` für die schnelle Offset-Berechnung am Ende unverändert bleiben muss. Daher wird für den Zugriff auf das Array statt `fieldAccess` der Ausdruck `fieldAccessAlt` verwendet, welcher den Index `j` für den Zugriff benutzt. Der Vergleich findet mit dem generierten Defaultwert statt (`defaultValue`). Weicht der Wert ab wird `defaults` auf `false` gesetzt und wir können die Schleife abbrechen. Dies ist eine wichtige Optimierung, da bei großen Chunks die Iteration früh beendet werden kann. Bei Daten, die nur selten Standardwerte haben, kann das relativ schnell passieren und reduziert somit den Overhead durch die neu eingeführte for-Schleife. Die `defaultsOnly`-Variable der `FieldDeclaration` wird danach entsprechend geändert. Diese existiert in C++ nicht in der `FieldDeclaration` sondern ist eine lokale Variable. Die logische Implementierung der Defaultwert-Erkennung sollte aus dem `DistributedField` bekannt sein. Abschließend wird der vorher berechnete Offset-Code eingefügt.

Kann kein schneller Offset-Code verwendet werden, ist die Methode etwas größer. `prelude` (Java) generiert lokale Variablen für verschiedene `FieldTypes`. Diese werden z.B. beim String benötigt (StringPool), um einen String aus der gelesenen ID zu bekommen. Bei den Containern erhält man damit Zugriff auf den/die zugrundeliegenden Typ/-en um den Offset eines einzelnen Elements zu berechnen zu lassen. In C++ ist das nicht nötig. Der Grund dafür liegt in der Generierung des Offset-Codes durch `offset`, welche nicht genauer betrachtet wird. Die darauffolgende Erzeugung der Variable für das Daten-Array sollte schon bekannt sein. Die Variable `result` die definiert wird, kumuliert den Offset über die Iterationen der for-Schleife. Des weiteren findet in der Schleife die bereits bekannte Erkennung von Defaultwerten statt. Am Ende wird der kumulierte Offset zur entsprechenden Variable in der `FieldDeclaration` addiert (Java) bzw. zurückgegeben (C++).

Listing 3.5 Generierung der osc-Methode (Java)

```
s""" ...
@Override
protected final void osc(int i, final int h) {${
        val (code, isFast) = offsetCode(t, f, originalF.getType, fieldActualType);
        if (isFast)
                s"""
                final ${mapType(t.getBaseType)}[] d = ((${access(t.getBaseType)})
                        owner.basePool).data();
                boolean defaults = true;
                for(int j = i; j != h; j++) {
                        if($fieldAccessAlt != ${defaultValue(f)}) {
                                defaults = false;
                                break;
                        }
                }
                defaultsOnly = defaultsOnly && defaults;
                $code"""
        else s"""${prelude(originalF.getType)}
        final ${mapType(t.getBaseType)}[] d = ((${access(t.getBaseType)}) owner.basePool).data();
        long result = 0L;
        boolean defaults = true;
        for (; i != h; i++) {
                $code
                if(defaults && $fieldAccess != ${defaultValue(f)})
                        defaults = false;
        }
        defaultsOnly = defaultsOnly && defaults;
        offset += result;"""
}}
... """
```

Listing 3.6 Generierung der osc-Methode (C++)

```
s"""...
if (fastOffset(f.getType)) {
        s"""${mapType(t)}* d = ((${storagePool(t)}*) owner)->data;
        const ::skill::internal::Chunk *target = dataChunks.back();
        bool defaultsOnly = true;

        for (::skill::SKilLID i = 1 + ((const ::skill::internal::SimpleChunk *) target)->bpo,
        high = i + target->count; i != high; i++) {
                if($accessI != ${defaultValue(f)}) {
                        defaultsOnly = false;
                        break;
                }
        }

        if(defaultsOnly)
                return 0;
        ${offsetCode(f.getType)}"""

} else {
        s"""
        ${mapType(t)}* d = ((${storagePool(t)}*) owner)->data;
        const ::skill::internal::Chunk *target = dataChunks.back();
        std::size_t result = 0L;
        bool defaultsOnly = true;

        for (::skill::SKilLID i = 1 + ((const ::skill::internal::SimpleChunk *) target)->bpo,
        high = i + target->count; i != high; i++) {
                ${offsetCode(f.getType)}
                if(defaultsOnly && $accessI != ${defaultValue(f)})
                        defaultsOnly = false;
        }

        if(defaultsOnly)
                return 0;
        return result;"""
}
... """
```

4. Evaluierung

In diesem Kapitel wird die Performance der veränderten SKilL-Implementierungen evaluiert. Dabei wird vor allem auf Lese- und Schreibperformance eingegangen, sowie der durch diese Arbeit entstandene Overhead bei Anwendungsfällen ohne Encodings. Die Evaluierung findet dabei auf Daten statt, die durch andere Anwendungsfälle entstanden und nicht artifiziell sind und damit praktische Aussagekraft haben.

4.1. Testdaten

Als Grundlage für die Evaluierung dient ein Datensatz aus *Bauhaus Intermediate Language (IML) -Graphen*. Die Bauhaus-Werkzeugkette kann auf Programmen in C oder C++ verschiedene Analysen durchführen. Dazu werden syntaktische und semantische Informationen aus dem Quelltext in einem IML-Graphen gespeichert. [Prz17, p. 9]

Der Datensatz besteht aus den IML-Graphen für verschiedene C-Programme, welche mittels SKilL serialisiert wurden. Dafür wurde die IML-Spezifikation auf eine SKilL-Spezifikation abgebildet und der Graph in diesem Typsystem in eine .sf-Datei serialisiert. Die C-Programme sind z.B. *bash*, *php*, *sed* oder *python*. Dazu gesellen sich selbstgeschriebene Programme anderer, an SKilL beteiligter Entwickler. [Fel18, p. 162]

4.2. Testumgebung

Die Tests wurden auf einem Rechner der Abteilung für Programmiersprachen und Übersetzerbau durchgeführt. Dieser war folgendermaßen ausgestattet:

- **CPU** 4x Intel Xeon ES-4640 v4 ($\sum$ 96x 2.1GHz)
- **RAM** 1,5 TB DDR4 @ 2400 MHz
- **HDD** PERC H730 (Raid 1)
- **Betriebssystem** Debian 8

Die Java Tests wurden mit dem Java 1.8 JDK übersetzt und ausgeführt.

Für C++ wurde gcc 4.9.2 verwendet. Die C++-Tests wurden mit dem Debug-Flag `-g` und ohne Optimierungen durchgeführt. Dies geschah, da zwischendurch Debug-Informationen benötigt wurden, jedoch vergessen wurde den Build für die Tests zu verändern. Als dieser Fehler auffiel war nicht mehr genug Zeit übrig um die Tests erneut durchzuführen. Die C++-Tests welche die Originalimplementierung verwenden wurden mit `-O3` und ohne `-g` kompiliert.

4.3. Testaufbau

Der atomare Testvorgang wird durch einen *Executor* durchgeführt. Ein atomarer Vorgang ist das Lesen einer .sf-Datei (Read). Dabei wird die .sf-Datei komplett eingelesen und die Zeit dafür gemessen, sowie die Dateigröße und die Anzahl der Objekte in der .sf-Datei bestimmt. Die andere atomare Operation ist das Schreiben einer .sf-Datei (Write) unter Verwendung eines bestimmten Encodings auf allen Feldern. Dabei wurde die angegebene .sf-Datei zunächst eingelesen. Nachdem bei allen Feldern eine Coding-Restriction hinzugefügt wird, wird der State wieder serialisiert. Dabei wurde die Schreibzeit gemessen. Der Executor besitzt dabei den generierten Code aus der Spezifikation, die den .sf-Dateien entsprach.

Ein Dispatcher startete dann einen Executor auf verschiedenen Dateien. Dabei wurde im ersten Schritt jede Datei reserialisiert, damit sie nur noch einen Block enthält und die Standardwert-Optimierung angewendet wird. Dadurch werden bei der Messung der originalen Dateigröße die Standardwerte nicht mit gemessen. So wird das Kompressionsverhältnis nicht durch plötzlich verschwundene Standardwerte verfälscht. Die Executor wurden in jeweils einem eigenen Prozess ausgeführt, um Verfälschungen durch einen aufgewärmten JIT-Compiler oder Garbage Collector zu vermeiden.

Auf einer höheren Ebene lässt sich ein Testschritt durch ein Encoding definieren. Dabei werden alle .sf-Dateien zunächst mit dem atomaren Write-Vorgang mit dem Encoding auf allen erlaubten Feldern reserialisiert, bevor sie wieder mit dem atomaren Read-Vorgang eingelesen werden um Lesezeit und neue Dateigröße festzustellen. Jeder Testschritt wurde auf jeder Datei 10 mal durchgeführt.

Für die Caching-Evaluation wurden Tests mit `valgrind --tool=massif` durchgeführt. Dabei wurde die Heap-Nutzung gemessen. Dabei wurden drei Werte gemessen und addiert: Angeforderte Heap-Bytes, zusätzliche Heap-Bytes und Stack-Bytes. Es wurde nur der maximale Speicherverbrauch gemessen.

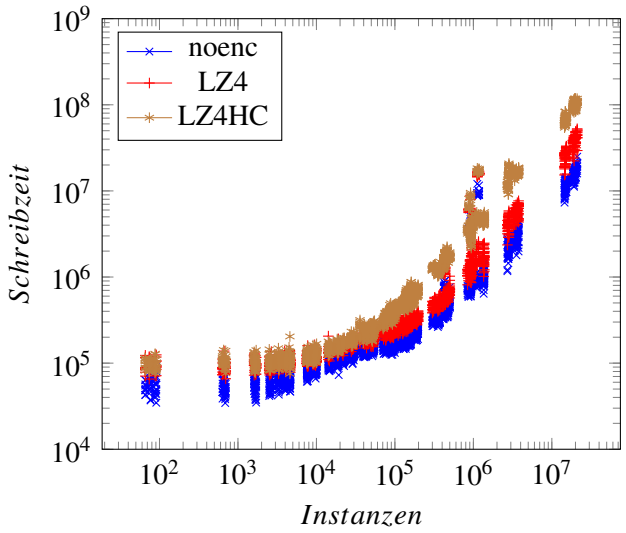
Die Evaluierung besteht aus drei Testdurchläufen:

- LZ4-Tests
- Bitvector-Tests
- Overhead-Tests

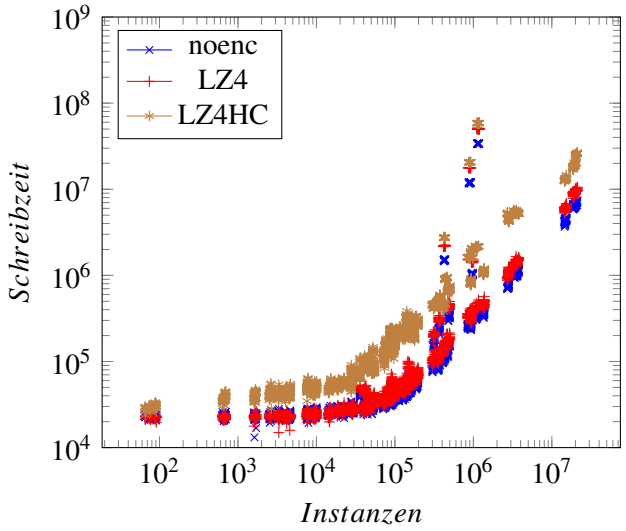
Die LZ4-Tests bestanden aus 5 Schritten. Das Serialisieren ohne Encoding, sowie das Serialisieren mit jeweils einem der anderen 4 LZ4-Encodings (LZ4(HC)(Cached)). Bei den Bitvector-Tests wurden alle bool'schen Werte in einem Testschritt mit dem Bitvector-Encoding serialisiert. Der Overhead-Test besteht aus einem Schritt ohne ein Encoding. Dabei wird als Laufzeitbibliothek die SKilL-Implementierung ohne die Änderungen dieser Arbeit verwendet. Auch der generierte Code wurde mit dem originalen Codegenerator erstellt. Dieses Ergebnis lässt sich mit dem No-Encoding-Schritt aus den LZ4-Tests vergleichen, um festzustellen, welchen Overhead diese Arbeit verursacht.

Die Schreib- und Lesezeiten sind in Mikrosekunden angegeben. Die Dateigröße ist in Bytes angegeben. Durch die logarithmische Skalierung lassen sich diese Werte gut interpretieren.

4.4. LZ4-Tests



(a) Java-Implementierung



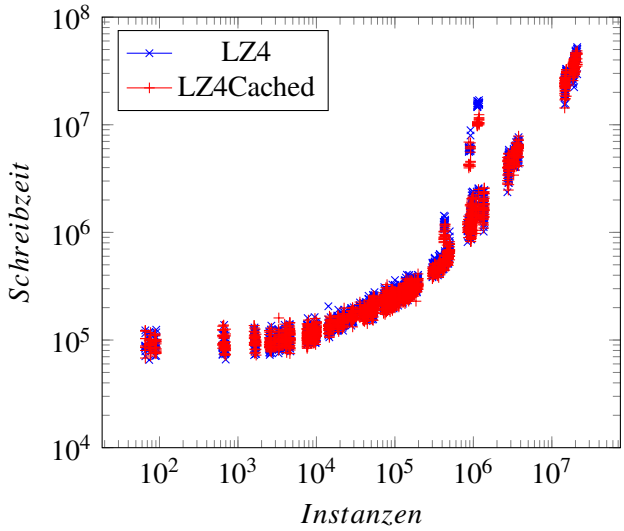
(b) C++-Implementierung

Abbildung 4.1.: Schreibzeit mit LZ4, LZ4HC und ohne Encoding

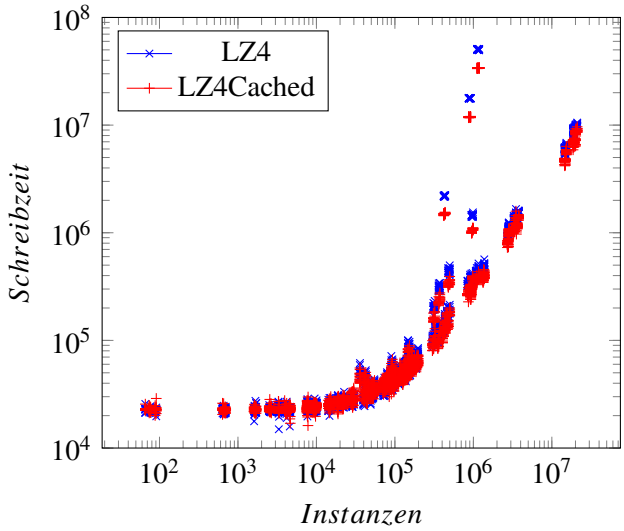
Abbildung 4.1a zeigt die Schreibzeiten mit dem LZ4- und dem LZ4HC-Encoding sowie ohne Encoding. Dabei wurde die Java-Implementierung gemessen. Auffallend sind einzelne Ausreißer, die eine ungewöhnlich hohe Zeit für ihre Objektzahl brauchen. Die Dateigröße ist auffallend groß und die Ausreißer treten auch bei anderen Werten an der gleichen Stelle auf, somit ist ein Messfehler unwahrscheinlich.[Fel18, p. 169f.] beschreibt diese Ausreißer ebenfalls und weist auf ein ungewöhnlich hohes Kanten-/Knotenverhältnis hin. Wie erwartet braucht LZ4HC länger als LZ4, welches wiederum länger braucht als No-Encoding. LZ4HC wird mit zunehmender Instanzanzahl teurer genauso wie LZ4. LZ4 macht bei mittlerer Größe den wenigsten Unterschied.

Die Ergebnisse der C++-Implementierung sind in Abbildung 4.1b zu sehen.Die gleichen Ausreißer sind vorhanden. Diese zeigen sich jedoch um eine Größenordnung deutlicher als bei Java. Die LZ4-Laufzeit unterscheidet sich kaum von der No-Encoding-Laufzeit. Bei kleinen Dateien ist LZ4 sogar teilweise schneller. LZ4HC dagegen hebt sich von Beginn an deutlich von den beiden anderen ab. Im Gegensatz zu den Ergebnissen in [Fel18, p. 183] zeigt sich hier keine alternierende Laufzeit bei kleinen Dateien in C++.

Im direkten Vergleich der beiden Implementierungen fällt auf, dass Java im allgemeinen langsamer ist als die C++-Implementierung. Laut [Fel18, p. 185f.] liegen die höheren Fixkosten von Java erfahrungsgemäß am JIT-Compiler.



(a) Java-Implementierung



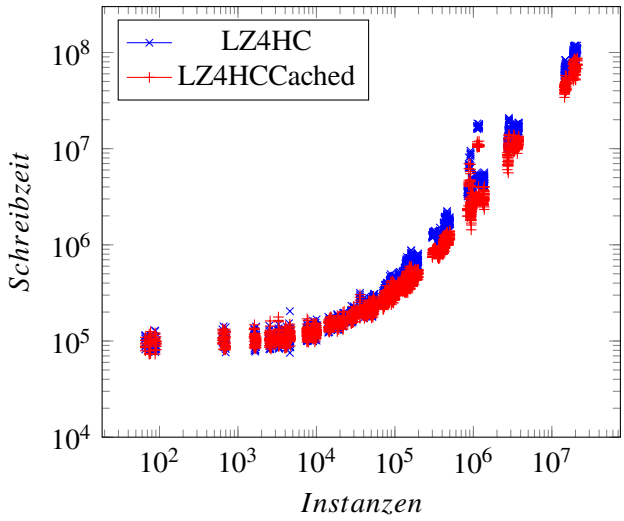
(b) C++-Implementierung

Abbildung 4.2.: Schreibzeit von LZ4Cached im Vergleich

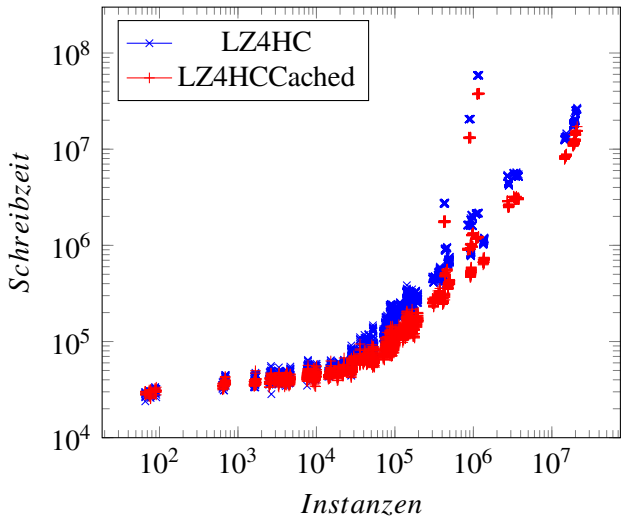
Im Folgenden untersuchen wir den Effekt des Cachings. Abbildung 4.2a und Abbildung 4.2b zeigen den Vergleich zu LZ4 ohne Caching. Es ist klar, dass die Einsparung durch das Vermeiden eines zweiten Kompressionsvorgang umso höher ist, je höher die Kosten des Kompressionsvorgang sind. Bei beiden Implementierungen ist kaum ein Unterschied durch das Caching zu sehen. Erst bei den großen Dateien sieht man eine leichte Verbesserung. Die Ausreißer werden durch das Caching deutlich schneller serialisiert. Die Effizienz des Cachings scheint also durch ein hohes Kanten-/Knoten-Verhältnis erhöht zu werden. Abbildung 4.3a und Abbildung 4.3b zeigen den Caching-Vergleich für LZ4HC. Hier ist auch bei Dateien mittlerer Größe schon eine deutliche Optimierung zu erkennen. Bei C++ ist diese stärker als bei Java. Die Speichernutzung wurde auch gemessen. Das Caching erhöht die maximale Speichernutzung leicht bei großen Dateien. Diese Erhöhung ist allerdings sehr gering und auf einem Plot nicht sichtbar, daher wurde auf diesen verzichtet. Das Caching erhöht den Speicherverbrauch zwischen der Offset-Berechnung und der Serialisierung, da in diesem Zeitraum alle komprimierten Daten zwischengespeichert sind. Zum Zeitpunkt des tatsächlichen Schreibens der Daten sind die komprimierten Daten auch ohne Caching im Speicher. Daher ist es nicht zu erwarten, dass sich die maximale Speichernutzung beim Caching deutlich abhebt. Beim parallelen Schreiben der Chunks ist es zu erwarten, dass kleine Chunks oft schneller fertig sind als große Chunks. Dann kann es passieren, dass bei kleinen Chunks der Speicher früh freigegeben wird. Währenddessen sind große Chunks noch am komprimieren. In diesem Fall sind im Gegensatz zum Caching zu keinem Zeitpunkt alle komprimierten Daten im Speicher. Dies könnte erklären, warum trotzdem kleine Unterschiede bei der maximalen Speichernutzung vorhanden sind.

Das Kompressionsverhältnis, also der Quotient aus komprimierter Dateigröße und originaler Dateigröße, ist in Abbildung 4.4a und Abbildung 4.4b zu sehen. Das Verhältnis ist für beide Sprachen fast identisch. Die Werte größer als 1 sind dadurch zu erklären, dass es viele Felder gibt, die durch das Byte, welches die Omission serialisiert, größer sind als ohne Encoding. Mit steigender Dateigröße erkennt man, dass das LZ4HC-Encoding eine leicht besseres Kompressionsverhältnis erzeugt. Zu beachten ist, dass nicht die gesamte Datei komprimiert wird, das heißt es handelt sich hier nicht um das tatsächliche Kompressionsverhältnis, denn ein konstanter Anteil bleibt unverändert erhalten. Der Plot zeigt auch dieselben Ausreißer, welche durch eine sehr hohe Komprimierbarkeit auffallen. Die fast identischen Diagramme sind definitiv zu erwarten, da der LZ4-Algorithmus in beiden Implementierungen gleich funktioniert und die Daten auf denen operiert wird, ebenfalls die gleichen sind.

Abbildung 4.5a und Abbildung 4.5b plotten die Lesezeit abhängig von der Anzahl der Instanzen in der Datei. Die Lesezeit ändert sich durch die Dekompression fast nicht. Bei Java verbessert sie sich bei den Ausreißern sogar punktuell. Das lässt sich darauf zurückführen, dass durch die hohe Komprimierbarkeit der Ausreißer der Lesevorgang aus der Datei an sich beschleunigt wird, da weniger Daten zu lesen sind und dies die zusätzlichen, wenigen Kosten durch die Dekompression amortisiert. Dieser Effekt lässt sich bei Java sehr deutlich beobachten. Die Betrachtung des Cachings im Kontext der Lesezeit ist nicht sinnvoll, da das Caching nur Auswirkungen auf die Serialisierung hat.



(a) Java-Implementierung



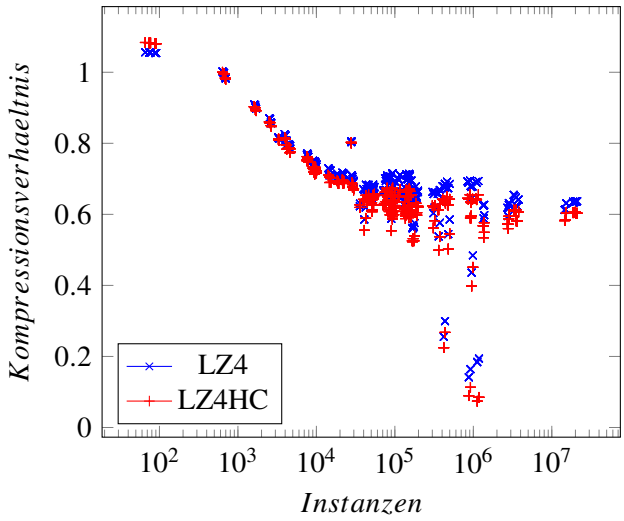
(b) C++-Implementierung

Abbildung 4.3.: Schreibzeit von LZ4HCCached im Vergleich

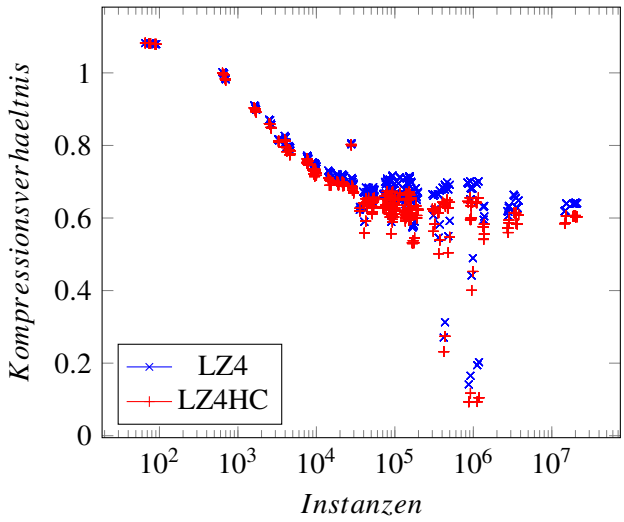
4.5. Bitvector-Tests

Bei den Bitvector-Tests wurden alle bool-Felder in einem Testschritt mit dem Bitvector-Encoding serialisiert. Um die Ergebnisse besser beurteilen zu können, betrachten wir in Abbildung 4.6 zunächst die Anzahl der Bool-Werte in Abhängigkeit zur Instanzanzahl. Die Anzahl der Bool-Werte ist die Summe der Werte aller Bool-Felder. Wir sehen, dass das log-log-Schaubild annähernd eine Gerade formt. Eine Potenz-Regression ergibt einen Exponenten von ungefähr 0,8. Das Verhältnis ist somit nicht ganz linear, die Wachstumsrate sinkt bei größeren Dateien.

Um aussagekräftigere Werte zu erhalten, plotten wir die Laufzeiten in Abhängigkeit der Anzahl der Bool-Werte. Dabei ist zu beachten, dass die Daten einiger Felder eventuell durch die Defaultwert-Optimierung wegoptimiert wurden und damit keine Kompression durchlaufen.



(a) Java-Implementierung

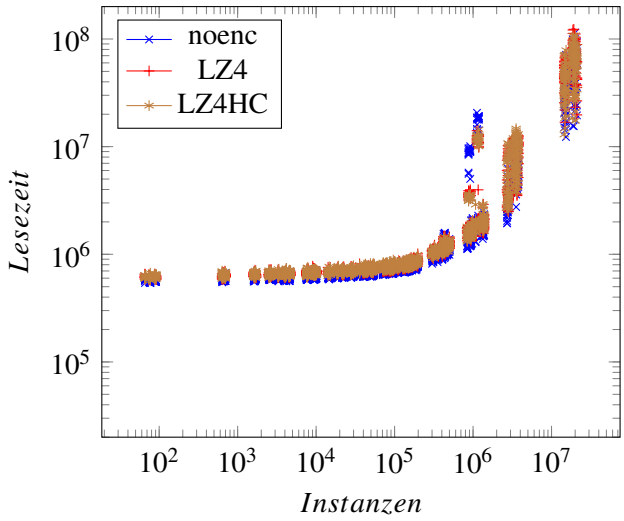


(b) C++-Implementierung

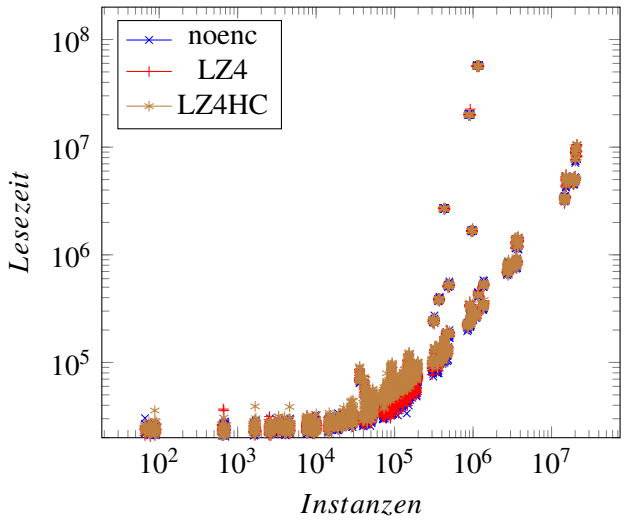
Abbildung 4.4.: Kompressionsverhältnis von LZ4 und LZ4HC

Die Abbildung auf die Schreibzeit befindet sich in Abbildung 4.7a und Abbildung 4.7b. Vor allem in C++ lässt sich kaum eine Veränderung der Serialisierungszeit beobachten. In Java tendiert es zu einer leichten Verschlechterung, punktuell aber auch zu einer Verbesserung. Dass das Bitvector-Encoding kaum teurer als die normale Serialisierung ist, war zu erwarten. Die Kompression besteht aus einer Iteration über den Daten und Bit-Operationen durch den Stream, was nicht groß ins Gewicht fällt.

Die Lesezeiten sind in Abbildung 4.8a und Abbildung 4.8b aufgetragen. Auch hier sieht man keine eindeutige Veränderung gegenüber der Lesezeit ohne Encoding. Bei Java findet man deutliche Schwankungen bei großen Dateien in beide Richtungen.



(a) Java-Implementierung



(b) C++-Implementierung

Abbildung 4.5.: Lesezeit mit LZ4, LZ4HC und ohne Encoding

4.6. Overhead-Tests

Hier vergleichen wir die Performanz der originalen Implementierung mit der, die die Änderungen im Zuge dieser Arbeit enthält. Dabei wird kein Encoding angewandt, dies ist bei der originalen Implementierung auch nicht möglich. Die Schreibzeit der beiden Implementierungen sind für Java und C++ in Abbildung 4.9a aufgetragen. Bei kleinen Dateien ist die originale Implementierung leicht schneller. Bei großen Dateien ist sie deutlich schneller, vor allem in Java. Der Overhead durch die Implementierung der Kompression ist, wenn kein Encoding verwendet wird, nicht hoch. Es werden lediglich ein paar zusätzliche Überprüfungen durchgeführt. Den größten Einfluss besitzt die Defaultwert-Optimierung. Diese greift invasiv in die Offset-Berechnung ein und verhindert bei primitiven Datentypen eine schnelle Offset-Berechnung durch eine arithmetische Operation. Stattdessen existiert dort eine Schleife die Vergleiche durchführt. Wenn eine Optimierung stattfindet,

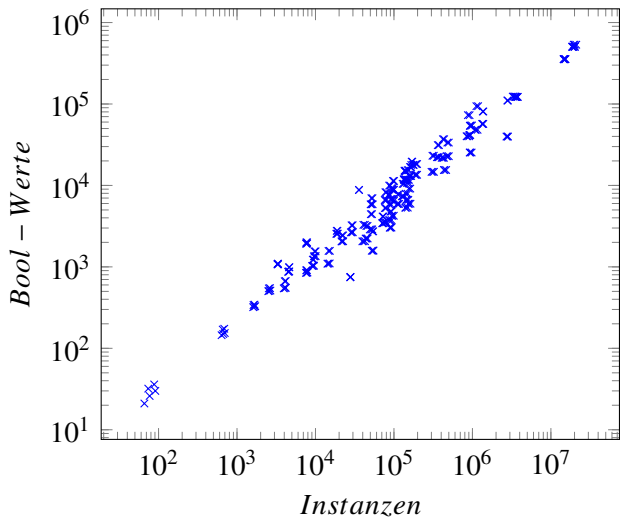
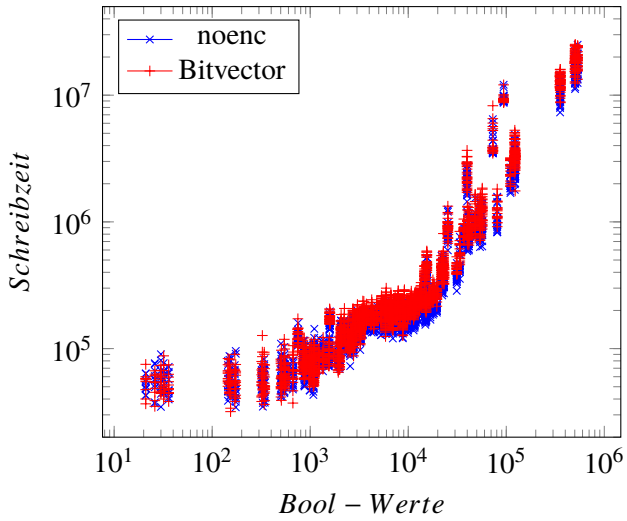


Abbildung 4.6.: Anzahl an Bool-Werten in Abhängigkeit zu der Anzahl der Instanzen

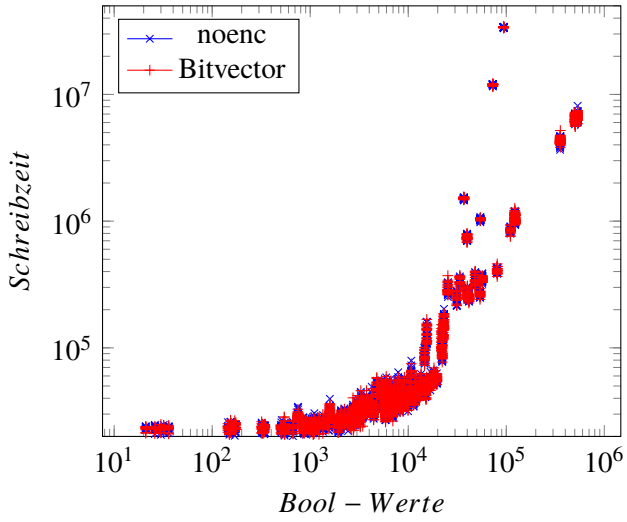
läuft diese Schleife über alle Instanzen durch. Auch bei Offset-Berechnungen die eine Schleife enthalten wird der Aufwand durch die Vergleiche vergrößert. Bei C++ ist zu beachten, dass die originale C++-Implementierung ohne Debug-Flag und mit Optimierungen kompiliert wurde.

Den Vergleich der Dateigrößen findet man in Abbildung 4.10a und Abbildung 4.10b. Über alle Dateigrößen findet man einen konstanten Unterschied auf dem log-log-Plot, also einen linear mit der Dateigröße wachsenden Unterschied. Dabei ist die Datei, welche mit Defaultwert-Optimierung serialisiert wurde erwartungsgemäß kleiner. Dem entgegen wirken die zusätzlichen Restrictions, die jetzt serialisiert werden können. Die Originaldateien sind bei großen Dateien rund 12% - 25% größer, was auf die fehlende Defaultwert-Optimierung zurückzuführen ist. Die einzelnen Dateien gruppieren sich dabei an den beiden Grenzen. Beide Diagramme sind wie erwartet identisch, da die Optimierung in beiden Implementierungen deterministisch gleich erfolgen muss.

Wie Abbildung 4.11a und Abbildung 4.11b zeigen, ist auch das Lesen etwas teurer geworden. Der vor allem bei Java auftretende Schreib-Overhead bei großen Dateien ist hier nicht vorhanden. Bei Java ist sogar zu beobachten, dass die originale Implementierung punktuell gleich schnell ist. Dafür beobachtet man bei C++ bei den Ausreißern einen deutlichen Unterschied. Der Einfluss auf das Lesen ohne Encodings ist nicht so hoch wie auf das Schreiben. Es existieren nur die schon beschriebenen zusätzlichen Abfragen. Defaultwerte werden mit einem im Code generierten Wert gefüllt, was schneller ist als die Daten aus der Datei zu lesen. Das spiegelt auch der Durchschnitt der Diagramme wider.

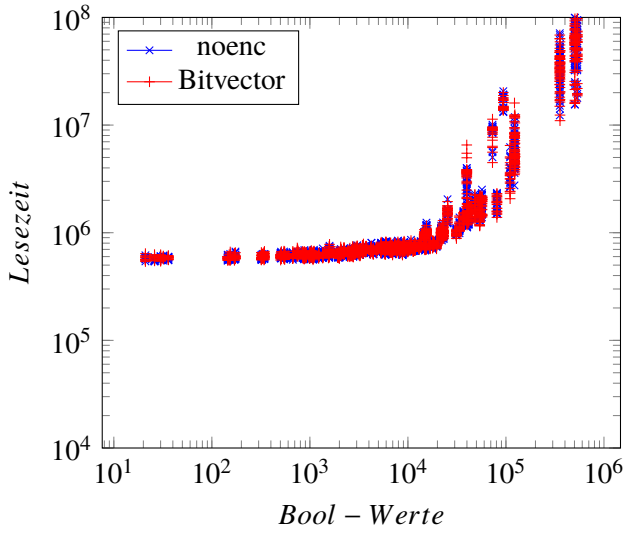


(a) Java-Implementierung

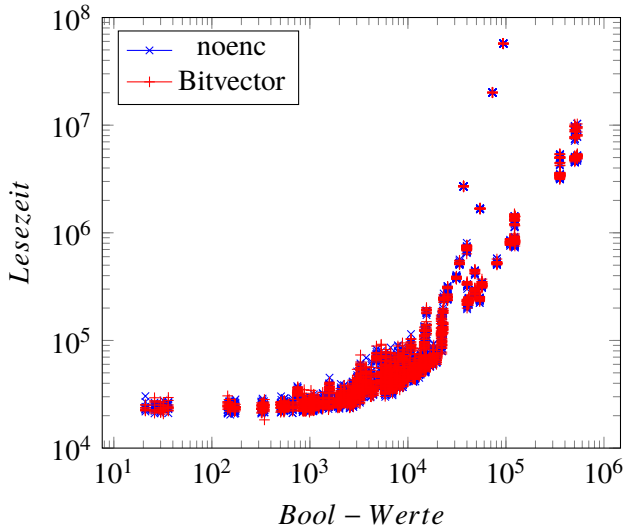


(b) C++-Implementierung

Abbildung 4.7.: Schreibzeit mit Bitvector-Encoding und ohne Encoding

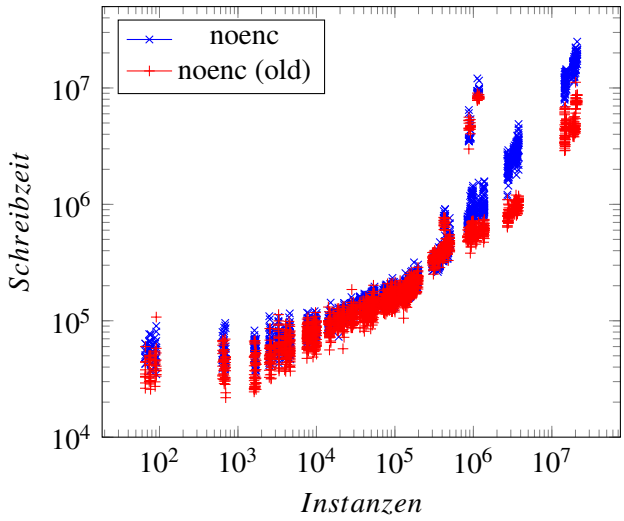


(a) Java-Implementierung

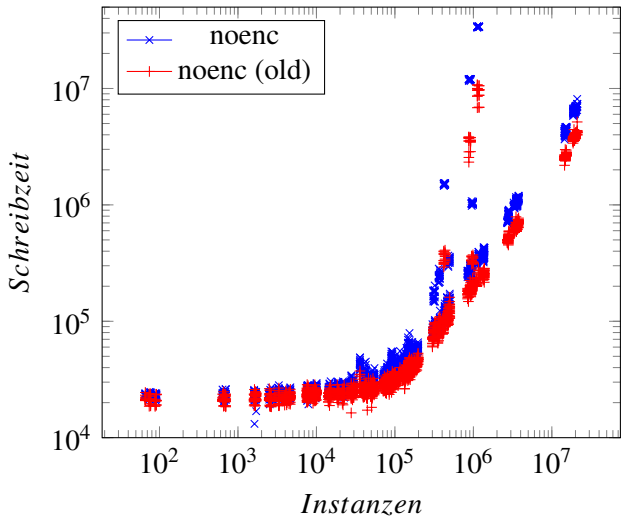


(b) C++-Implementierung

Abbildung 4.8.: Lesezeit mit Bitvector-Encoding und ohne Encoding

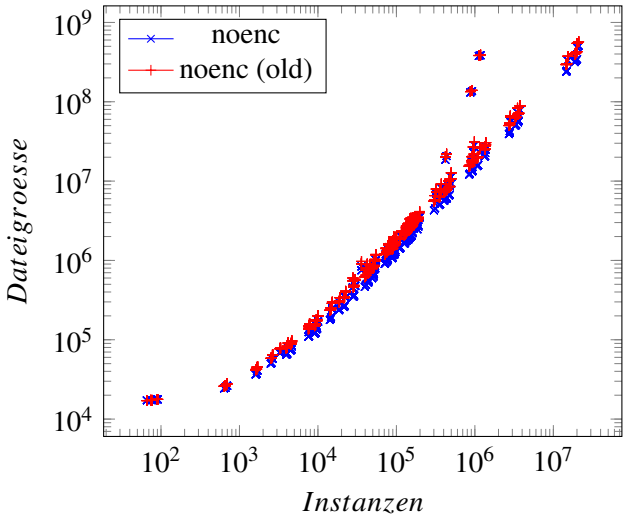


(a) Java-Implementierung

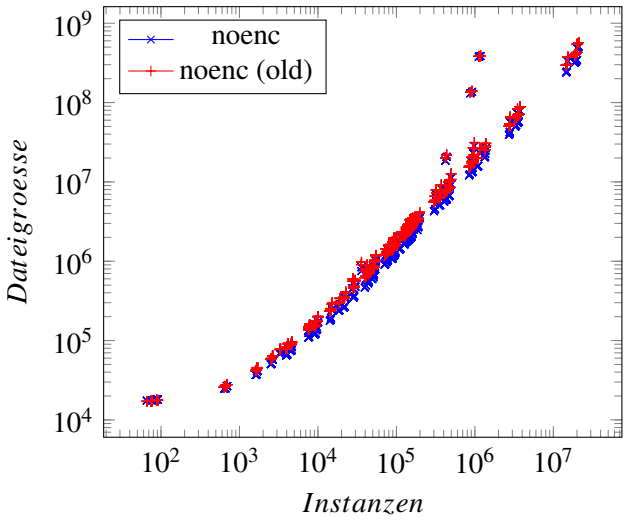


(b) C++-Implementierung

Abbildung 4.9.: Schreibzeit-Vergleich zur Original-Implementierung

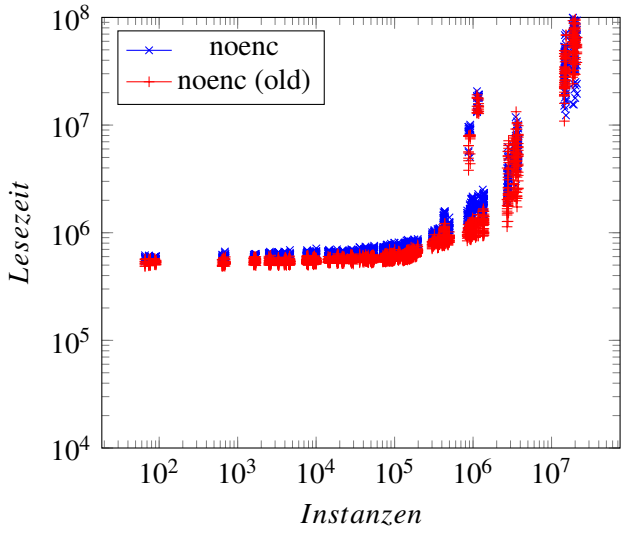


(a) Java-Implementierung

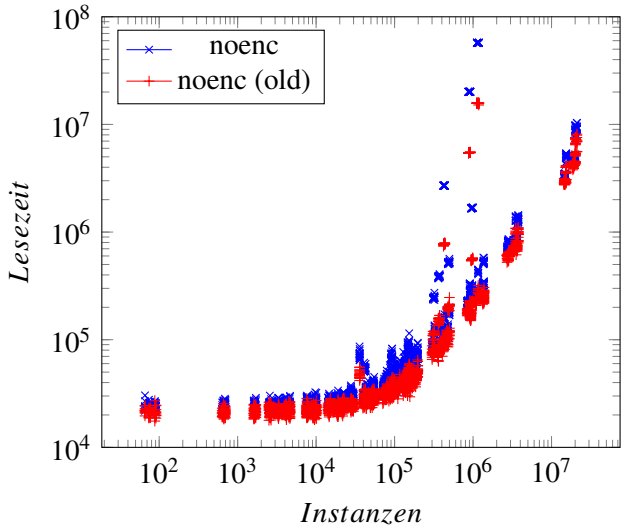


(b) C++-Implementierung

Abbildung 4.10.: Dateigrößen-Vergleich zur Original-Implementierung



(a) Java-Implementierung



(b) C++-Implementierung

Abbildung 4.11.: Lesezeit-Vergleich zur Original-Implementierung

5. Fazit

In dieser Arbeit wurde eine Kompression von Felddaten und eine Defaultwert-Optimierung in SKilL implementiert und evaluiert. Wie gezeigt wurde, ließen sich die Encodings gut minimal-invasiv in die bisherige Infrastruktur einbauen, ohne dabei den bisherigen Lese- und Schreibvorgang der Felder zu ändern. Eine Herausforderung stellte dabei die Anforderung dar, dass die Größe der Daten schon beim Schreiben der Felddefinitionen bekannt sein muss. Es hat sich die Frage gestellt ob ein Caching der dabei entstehenden komprimierten Daten die Laufzeit verbessert. Die Evaluation konnte dies für große Dateien bestätigen. Das Umsetzen der Defaultwert-Erkennung gestaltete sich als schwierig, da keine bestehende Logik für Defaultwerte vorhanden war. Mithilfe einer behelfsmäßigen Implementierung wurde ein Grundstein gelegt, der unter einer robusten Defaultwert-Logik zu voller Entfaltung kommen kann. Die Implementierung hatte wie erwartet Auswirkungen auf die Laufzeit, wenn eine Kompression verwendet wird. Mittels LZ4HC konnte die Kompression für mittelgroße und große Dateien verbessert werden, allerdings auf Kosten der Laufzeit. Es wird hier keine Aussage darüber getroffen, ob sich der Mehraufwand von LZ4HC lohnt. Das ist für jeden Anwendungsfall individuell zu bewerten. Des weiteren wurde festgestellt, dass es durch die Implementierung merkbare Veränderungen der allgemeinen Performanz des Serialisierungsvorganges gibt, was vermutlich durch die Defaultwert-Erkennung verursacht wird. Diese konnte aber eine Reduktion der Dateigröße bewirken.

Ausblick

Die Klassenhierarchie der Encodings ist dafür ausgelegt auch weitere Kompressionsverfahren zu unterstützten. Dieser Erweiterung steht nichts im Weg, so können beliebte Verfahren wie zip eingebaut werden. Da die Defaultwert-Optimierung sich auf die Performanz auswirkt, kann die Möglichkeit geschaffen werden, diese optional abzuschalten, sodass sie nicht in den Code generiert wird. Dadurch lässt sich eine Optimierung der Laufzeit erzielen. Mit einer korrekten Defaultwert-Logik lässt sich diese sogar noch effektiver gestalten.

A. Code-Ausschnitte

In diesem Anhang befinden sich verschiedene, in der Arbeit referenzierte, Code-Ausschnitte die nicht zum unmittelbaren Verständnis an der jeweiligen Stelle notwendig sind und nur zur Ergänzung dienen.

Listing A.1 Implementierung der i1()-Methode im InStream (Java)

```
/**
* Reads the next bit
* @return 0x00 if bit not set, 0xFF else
* @note Intermediate full-byte reads will read from the next byte,
* so the remaining unread bits are lost in that case
*/
public final byte i1() {
// We need to start with a new byte if either there were intermediate full-byte read operations,
// or if we are not in the middle of another byte.
// bitIndex determines the position in the byte to read next,
// so 0 is default and also means the byte is not cached yet
        if(bitIndex == 0 || lastCachedPosition != position()) {
                // Reset index in case it was not 0
                bitIndex = 0;
                cachedByte = i8();
                // Cache position so we can detect if there were intermediate full-byte
                // read operations between i1()-calls
                lastCachedPosition = position();
        }

        // read the specified bit
        int retVal = cachedByte & (0x80 >> bitIndex++);

        // reset offset if we finished a byte
        if(bitIndex >= 8) {
                bitIndex = 0;
        }

        return (byte)(retVal == 0 ? 0x00 : 0xFF);
}
```

Listing A.2 Implementierung der encodedSize-Methode in PrepassEncoding (Java)

```
public long encodedSize(ByteBuffer data) {
        try {
                // Is most likely greater than actual data to avoid dynamic resizing buffer
                ByteBuffer b = ByteBuffer.allocate(maxEncodedSize(data.remaining()));
                VirtualOutStream out = new VirtualOutStream(b);
                encode(new MappedInStream(data), out);
                long position = out.position();
                if (cachePrepass) {
                        cachedBuffer = b;
                        // Reset position so we start reading at the beginning of the cache later
                        b.flip();
                }

                return position;
        } catch (IOException e) {
                e.printStackTrace();
        }

        return -1;
}
```

Listing A.3 Implementierung der encode-Methode des Bitvector-Encodings (Java)

```
public void encode(MappedInStream in, VirtualOutStream out) throws IOException {
        while (in.has(1)) {
                out.i1(in.i8());
        }
        out.flushByte();
}
```

Listing A.4 Implementierung der decode-Methode des Bitvector-Encodings (Java)

```
public ByteBuffer decode(MappedInStream in, int numElements) throws IOException {
        ByteBuffer b = ByteBuffer.allocate(numElements);
        for(int i = 0; i < numElements; i++) {
                // Read every bit and output it as a byte
                b.put(in.i1());
        }

        return b;
}
```

Listing A.5 Implementierung der encode-Methode für das LZ4-Encoding (Java)

```
public void encode(MappedInStream inData, VirtualOutStream out) throws IOException {
        out.v64(inData.asByteBuffer().remaining());
        LZ4Compressor compressor = getCompressor();
        compressor.compress(inData.asByteBuffer(), out.buffer());
}
```

Listing A.6 Implementierung der decode-Methode für das LZ4-Encoding (Java)

```
1  public ByteBuffer decode(MappedInStream in, int numElements) throws IOException {
2          int decodedSize = (int)in.v64();
3          ByteBuffer b = ByteBuffer.allocate(decodedSize);
4
5          LZ4Factory factory = LZ4Factory.fastestInstance();
6          LZ4FastDecompressor decompressor = factory.fastDecompressor();
7
8          ByteBuffer bIn = in.asByteBuffer();
9          int read = decompressor.decompress(bIn, bIn.position(), b, b.position(), decodedSize);
10         // Set byte buffers proper position
11         b.position(b.limit());
12         bIn.position(bIn.position() + read);
13
14         return b;
15 }
```

Listing A.7 Implementierung der Offset-Berechnung für einen SimpleChunk (Java)

```
1  /**
2  * calculates offset and takes CodingRestriction into account
3  * @see {@link #osc(int, int)}
4  */
5  protected void cosc(SimpleChunk c) {
6          int i = (int)c.bpo;
7          // We need to prove it's false
8          defaultsOnly = true;
9          osc(i, i + (int)c.count);
10         if(defaultsOnly) {
11                 // Only default values
12                 offset = 0;
13                 return;
14         }
15         if(encoding != null) {
16                 c.rawSize = offset;
17                 if(encoding instanceof PrepassEncoding) {
18                         offset = offsetPrepassSC(c);
19                         // Marks if encoding is actually applied
20                         // Used to optimize size by omitting the coding
21                         if(c.rawSize <= offset) {
22                                 offset = c.rawSize;
23                                 // Setting omitted and deleting cache
24                                 ((PrepassEncoding)encoding).omit();
25                         }
26                 // Mark byte
27                 ++offset;
28                 } else {
29                         // Set actual offset after encoding
30                         offset = ((SimpleEncoding)encoding).encodedSize(offset);
31                 }
32         }
33 }
34
35 private long offsetPrepassSC(SimpleChunk c) {
36         // Current implementation: Encode and throw away
37         try {
38                 ByteBuffer b = ByteBuffer.allocate((int) offset);
39                 VirtualOutStream outStream = new VirtualOutStream(b);
40                 int i = (int)c.bpo;
41                 wsc(i, i + (int) c.count, outStream);
42                 b.flip();
43                 return ((PrepassEncoding)encoding).encodedSize(b);
44         } catch(IOException e) {
45                 throw new IllegalArgumentException();
46         }
47 }
```

Listing A.8 Implementierung von osc im DistributedField (Java)

```
@Override
protected void osc(int i, int h) {
        final SkillObject[] d = owner.basePool.data;
        boolean defaults = true;
        long rval = 0;
        for (; i < h; i++) {
                rval += type.singleOffset(data.get(d[i]));
                if(defaults) {
                        T val = data.get(d[i]);
                        // equals() not callable on null
                        if(val == null) {
                                if(null != type.getDefaultValue()) {
                                        defaults = false;
                                }
                        } else {
                                if(!val.equals(type.getDefaultValue())) {
                                        defaults = false;
                                }
                        }
                }
        }
        offset += rval;
        defaultsOnly = defaults && defaultsOnly;
}
```

Listing A.9 Implementierung von rscd/rbcd für Distributed-Fields (Java)

```
@Override
protected void rscd(int i, int end) {
        final SkillObject[] d = owner.basePool.data;
        for (; i != end; i++) {
                data.put(d[i], type.getDefaultValue());
        }
}

@Override
protected void rbcd(BulkChunk c) {
        final SkillObject[] d = owner.basePool.data;
        ArrayList<Block> blocks = owner.blocks();
        int blockIndex = 0;
        final int endBlock = c.blockCount;
        while(blockIndex < endBlock) {
                Block b = blocks.get(blockIndex++);
                int i = b.bpo;
                for(final int h = i + b.count; i != h; i++) {
                        data.put(d[i], type.getDefaultValue());
                }
        }
}
```

Listing A.10 Ausschnitt aus dem Code der einen Chunk für ein Feld liest (Java)

```
// Default values
if(c.begin == c.end) {
        if(c instanceof BulkChunk)
                f.rbcd((BulkChunk)c);
        else {
                int i = (int)((SimpleChunk)c).bpo;
                f.rscd(i, i + (int)c.count);
        }

        return;
}

// check that map was fully consumed and remove it
MappedInStream map = in.map(0L, c.begin, c.end);
if(encoding != null) {
        boolean omit = false;
        if(encoding instanceof PrepassEncoding) {
                // Was the encoding omitted due to size reasons?
                omit = map.bool();
        }

        if(!omit) {
                ByteBuffer b = encoding.decode(map, (int) c.count);
                b.rewind();
                map = new MappedInStream(b);
        }
}

if (c instanceof BulkChunk)
        f.rbc((BulkChunk) c, map);
else {
        int i = (int) ((SimpleChunk) c).bpo;
        f.rsc(i, i + (int) c.count, map);
}
```

Listing A.11 Ausschnitt aus der Serialisierung eines Chunk aus Felddaten (Java)

```
if(begin == end) {
        // Exit early if there is nothing to serialize
        return;
}


Chunk c = f.lastChunk();

// collect written chunk with virtual out stream
VirtualOutStream localOut = outMap;
ByteBuffer b = null;

boolean omit = false;

if (f.encoding != null) {
        if (f.encoding instanceof PrepassEncoding) {
                // Write mark byte
                omit = ((PrepassEncoding)f.encoding).shouldOmit();
                localOut.bool(omit);
                if(!omit) {
                        ByteBuffer cache = ((PrepassEncoding) f.encoding).getCachedBuffer();
                        if (cache != null) {
                                // If prepass should be cached and is cached write and return
                                // We can omit writing and encoding since we did this already
                                outMap.put(cache);
                                f.encoding.reset();
                                return;
                        }
                b = ByteBuffer.allocate((int) c.rawSize);
                localOut = new VirtualOutStream(b);
                }
        } else {
                b = ByteBuffer.allocate((int) c.rawSize);
                localOut = new VirtualOutStream(b);
        }
}


if (c instanceof SimpleChunk) {
        int i = (int) ((SimpleChunk) c).bpo;
        f.wsc(i, i + (int) c.count, localOut);
} else
        f.wbc((BulkChunk) c, localOut);

if (f.encoding != null) {
        if (b != null) {
                b.rewind();
                // Encode written data
                f.encoding.encode(new MappedInStream(b), outMap);
        }
        f.encoding.reset();
}
```

Listing A.12 Implementierung der i1-Operation für den MappedOutStream (C++)

```
void i1(uint8_t v) {
        if(position != cachedPosition || byteOffset >= 8) {
                // Restart
                *(position) = 0x00;
                byteOffset = 0;
                // We need to do this, so if an intermediate write-operation gets called,
                // it does not overwrite our byte,
                // but continues on the next one
                // Also, if we stop writing in the middle of the byte the byte counts as written
                // which imitates correct flush behaviour
                position++;
                cachedPosition = position;
        }

        if(v != 0x00)
                *(position-1) |= 0x80 >> byteOffset;
}
```

Listing A.13 Implementierung der encodedSize-Methode des Prepass-Encodings (C++)

```
size_t skill::encodings::PrepassEncoding::encodedSize(uint8_t *data, size_t size) {
        size_t maxSize = maxEncodedSize(size);
        auto* encodedData = new uint8_t[maxSize];
        streams::MappedOutStream out(encodedData, encodedData + maxSize);
        streams::MappedInStream in(data, data, data + size);
        encode(&in, &out);

        if(cachePrepass) {
                cache = encodedData;
                cachedSize = maxSize;
        } else {
                delete[] encodedData;
        }

        return (size_t)out.getPosition();
}
```

Listing A.14 Implementierung der encode-Methode des Bitvector-Encodings (C++)

```
void skill::encodings::BitvectorEncoding::encode(skill::streams::MappedInStream *stream,
        skill::streams::MappedOutStream *outStream) {
        while(stream->has(1)) {
                outStream->i1((uint8_t)stream->i8());
        }
}
```

Listing A.15 Implementierung der decode-Methode des Bitvector-Encodings (C++)

```
std::pair<std::unique_ptr<uint8_t[]>, size_t> skill::encodings::BitvectorEncoding::decode(
        skill::streams::MappedInStream *in, skill::SKilLID numElements) {
        auto ptr = new uint8_t[numElements];
        for(int i = 0; i < numElements; i++) {
                ptr[i] = in->i1();
        }

        return {std::unique_ptr<uint8_t[]>(ptr), (size_t)numElements};
}
```

Listing A.16 Implementierung der encode-Methode des LZ4-Encodings (C++)

```
void skill::encodings::LZ4Encoding::encode(skill::streams::MappedInStream *in,
        skill::streams::MappedOutStream *out) {
        out->v64(in->remaining());
        out->jump(compress((uint8_t*)in->getBase() + in->getPosition(),
                (uint8_t*)out->getBase() + out->getPosition(), (int)in->remaining(),
                (int)out->remaining()));
        in->jump(in->remaining());
}
```

Listing A.17 Implementierung der decode-Methode des LZ4-Encodings (C++)

```
std::pair<std::unique_ptr<uint8_t[]>, size_t>
        skill::encodings::LZ4Encoding::decode(skill::streams::MappedInStream *in,
        skill::SKilLID numElements) {
        // Original size is serialized first
        int64_t decodedSize = in->v64();
        auto* buf = new uint8_t[decodedSize];

        // Use fast or safe? Java uses fast bc we have decodedSize available.
        // But C documentation states, that safe can be as fast or even faster
        // So we will test usage with safe

        // Assumes that end points to end of chunk and this is a precisely mapped buffer

        int compressedSize = (int)in->remaining();

        LZ4_decompress_safe((char*)in->getBase() + in->getPosition(), (char*)buf,
                compressedSize, (int)decodedSize);

        in->jump(compressedSize);

        return { std::unique_ptr<uint8_t[]>(buf), (size_t)decodedSize };
}
```

Listing A.18 Implementierung von cosc (C++)

```
size_t skill::internal::FieldDeclaration::cosc(Chunk* const c) const {
        size_t calculatedOffset = osc();

        if(calculatedOffset == 0) {
                // Only default values
                return 0;
        }

        if (encoding != nullptr) {
                c->rawSize = calculatedOffset;

                if (auto *penc = dynamic_cast<encodings::PrepassEncoding *>(encoding)) {
                        calculatedOffset = offsetPrepass(c, calculatedOffset);
                        // Check efficiency
                        if (c->rawSize <= calculatedOffset) {
                                // Compression is useless
                                calculatedOffset = c->rawSize;
                                penc->omitEncoding();
                        }

                        // Mark byte for omit
                        ++calculatedOffset;
                } else if (auto *senc = dynamic_cast<encodings::SimpleEncoding *>(encoding)) {
                        return senc->encodedSize(calculatedOffset);
                } else {
                        // Invalid encoding
                        throw SkillException("Inavlid encoding type in offset calculation");
                }

        }

        return calculatedOffset;
}

size_t skill::internal::FieldDeclaration::offsetPrepass(
        skill::internal::Chunk *const c, size_t rawOffset) const {
        auto* buffer = new uint8_t[rawOffset];
        streams::MappedOutStream out(buffer, buffer + rawOffset);

        // Write data to buffer to determine its encoded size
        wsc(&out);
        // Has to be PrepassEncoding
        size_t rval = static_cast<encodings::PrepassEncoding*>
                (encoding)->encodedSize(buffer, rawOffset);
        delete[] buffer;
        return rval;
}
```

Listing A.19 Implementierung von osc im DistributedField (C++)

```
size_t DistributedField::osc() const {
        size_t result = 0;

        auto c = dynamic_cast<const ::skill::internal::SimpleChunk *>(dataChunks.back());
        auto i = c->bpo;
        const auto end = i + c->count;
        bool defaultsOnly = true;
        while (i != end) {
                if(defaultsOnly && data[i] == api::box((int64_t)0L))
                defaultsOnly = false;

                result += type->offset(data[i++]);
        }

        // We have only defaults, no serialization
        if(defaultsOnly)
                result = 0;

        return result;
}
```

Listing A.20 Lesen eines Chunks (C++)

```
// Check if there was no data serialized to trigger default value filling
if(dc->begin == dc->end) {
        if(auto c = dynamic_cast<const ::skill::internal::SimpleChunk*>(dc)) {
                int i = c->bpo + 1;
                f->rscd(i, i + c->count);
        } else {
                auto bc = dynamic_cast<const ::skill::internal::BulkChunk *>(dc);
                f->rbcd(bc);
        }

        continue;
}

MappedInStream *part = dataList[blockIndex].get();
// Map out chunk
skill::streams::MappedInStream in(part, dc->begin, dc->end);

streams::MappedInStream* pIn = &in;
std::unique_ptr<uint8_t[]> buffer = nullptr;
// If encoding was omitted
bool omit = false;
if(f->getEncoding() != nullptr) {

        if(dynamic_cast<encodings::PrepassEncoding*>(f->getEncoding())) {
                omit = in.boolean();
        }

        if(!omit) {

                auto decoded = f->getEncoding()->decode(pIn, dc->count);
                pIn = new MappedInStream(decoded.first.get(), decoded.first.get(),
                decoded.first.get() + decoded.second);
                // Keep decoded data valid
                buffer = std::move(decoded.first);
        }
}

try {
        if (auto c = dynamic_cast<const ::skill::internal::SimpleChunk *>(dc)) {
                int i = c->bpo + 1;
                f->rsc(i, i + c->count, pIn);
        } else {
                auto bc = dynamic_cast<const ::skill::internal::BulkChunk *>(dc);
                f->rbc(pIn, bc);
        }
} catch (...) {
        ...
}
if(f->getEncoding() != nullptr && !omit) {
        delete pIn;
}
```

Listing A.21 Schreiben eines Chunks (C++)

```
auto c = f->dataChunks.back();
// Early exit if we have nothing to write b/c of only default values
if(c->begin == c->end) {
        continue;
}
// Map chunk
const auto map = source->clone(c->begin, c->end);
streams::MappedOutStream* out = map;
uint8_t* buffer = nullptr;
bool omit = false;
if(f->getEncoding() != nullptr) {
        if(auto* penc = dynamic_cast<encodings::PrepassEncoding*>(f->getEncoding())) {
                // Write omit byte
                omit = penc->shouldOmit();
                map->boolean(omit);
                if(!omit) {
                        uint8_t* cache = penc->getCache();
                        if(cache != nullptr) {
                                void* position = (uint8_t*)map->getBase() + map->getPosition();
                                // We need to copy offset amount of bytes
                                // b/c cache may be greater than the data we need to copy
                                memcpy(position, cache, f->awaitOffset-1);
                                penc->reset();
                                delete map;

                                // We're done
                                continue;
                        }
                        buffer = new uint8_t[c->rawSize];
                        out = new streams::MappedOutStream(buffer, buffer + c->rawSize);
                }
        } else {
                buffer = new uint8_t[c->rawSize];
                out = new streams::MappedOutStream(buffer, buffer + c->rawSize);
        }

}

f->wsc(out);

if(f->getEncoding() != nullptr) {
        if(buffer != nullptr && !omit) {
                streams::MappedInStream in(buffer, buffer, buffer + c->rawSize);
                f->getEncoding()->encode(&in, map);
                delete[] buffer;
                delete out;
                out = nullptr;
        }
        f->getEncoding()->reset();
}

delete map;
```

Listing A.22 Erzeugung der Befehle, die eine entsprechende Restriction in das Set hinzufügen (Java)

```
s"""...
${
        def restrictionMethod(r : String) : String = s"addRestriction($r);"

        (for(r <- f.getRestrictions.asScala) yield
                makeRestriction(f.getType, r)).flatten.map(restrictionMethod).mkString("\n")
} ..."""

...

private final def makeRestriction(t : Type, r : Restriction) : Option[String] = Option(r match {

        ...

        case r : CodingRestriction => r.getValue match {
                case "bitvector" => t.getSkillName match {
                        case "bool" => "new CodingRestriction(BitvectorEncoding.getInstance())"
                        case s => System.err.println(
                                "Bitvector encoding doesn't support type: " + s); null
                }
                case "lz4" => "new CodingRestriction(new LZ4FastEncoding(false))"
                case "lz4hc" => "new CodingRestriction(new LZ4HCEncoding(false))"
                case "lz4cached" => "new CodingRestriction(new LZ4FastEncoding(true))"
                case "lz4hccached" => "new CodingRestriction(new LZ4HCEncoding(true))"

                case s => System.err.println("Unsupported coding: " + s); null
        }

        ...
})
```

Listing A.23 Erzeugung der Befehle, die eine entsprechende Restriction in das Set hinzufügen (C++)

```
s"""...
${
        def restrictionMethod(r : String) : String = s"addRestriction($r);"

        (for(r <- f.getRestrictions) yield
                makeRestriction(f.getType, r)).flatten.map(restrictionMethod).mkString("\n")

}
 ..."""

...

private final def makeRestriction(t : Type, r : Restriction) : Option[String] = Option(r match {
        ...
        case r : CodingRestriction => r.getValue match {
                case "bitvector" => t.getSkillName match {
                        case "bool" => "new ::skill::restrictions::Coding(
                                new ::skill::encodings::BitvectorEncoding())"
                        case s => System.err.println(
                                "Bitvector encoding doesn't support type: " + s); null
                }

                case "lz4" => "new ::skill::restrictions::Coding(
                        new ::skill::encodings::LZ4FastEncoding())"
                case "lz4cached" => "new ::skill::restrictions::Coding(
                        new ::skill::encodings::LZ4FastEncoding(true))"
                case "lz4hc" => "new ::skill::restrictions::Coding(
                        new ::skill::encodings::LZ4HCEncoding())"
                case "lz4hccached" => "new ::skill::restrictions::Coding(
                        new ::skill::encodings::LZ4HCEncoding(true))"

                case s => System.err.println("Unsupported coding: " + s); null
        }

        case s => System.err.println("Unsupported field restriction type: " + s.getName); null

})
```

Abbildungsverzeichnis

Verzeichnis der Listings

Literaturverzeichnis

[Ami16] K. I. L. Amit Jain. *Comparative Study of Dictionary based Compression Algorithms on Text Data*. 2016. URL: http://paper.ijcsns.org/07_book/201602/20160215.pdf (besucht am 03. 12. 2018) (zitiert auf S. 8).

[Col18] Y. Collet. 2018. URL: https://lz4.github.io/lz4/ (zitiert auf S. 8).

[Fel17] T. Felden. *The SKilL Language V1.0*. Techn. Ber. 2017. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2017-01/TR-2017-01.pdf (zitiert auf S. 7, 9, 11–14, 19, 26).

[Fel18] T. Felden. „Änderungstolerante Serialisierung großer Datensätze für mehrsprachige Programmanalysen“. 2018. DOI: http://dx.doi.org/10.18419/opus-9661 (zitiert auf S. 17, 34, 39, 41, 42).

[GJHV11] E. Gamma, R. Johnson, R. Helm, J. Vlissides. *Entwurfsmuster*. 2011. ISBN: 3827330432. URL: https://www.amazon.com/Entwurfsmuster/dp/3827330432?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=3827330432 (zitiert auf S. 18).

[GJS+18] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith. *The Java Language Specification*. Techn. Ber. 2018. URL: https://docs.oracle.com/javase/specs/jls/se11/html/index.html (zitiert auf S. 7).

[ISO13] ISO/IEC. *Programming Languages - C++*. Techn. Ber. 2013. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf (zitiert auf S. 7).

[LZ4] LZ4. *LZ4 Block Format Description*. URL: https://lz4.github.io/lz4/lz4_Block_format.html (besucht am 30. 10. 2018) (zitiert auf S. 8).

[Prz17] D. Przytarski. „SKilLed Bauhaus“. 2017. DOI: http://dx.doi.org/10.18419/opus-9270 (zitiert auf S. 39).

Alle URLs wurden zuletzt am 03. 12. 2018 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