Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Invasives Rust

Bachelorarbeit von

Hermann Krumrey

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	Dipl.-Inform. Andreas Zwinkau

Bearbeitungszeit: 1. Januar 1990 – 31. Dezember 2000

Zusammenfassung

Die Parallelisierung von Rechnern liegt immer mehr im Fokus der derzeitigen technologischen Entwicklung. Dies erfordert die Entwicklung und Nutzung von anderen Programmierparadigmen, welche effektiv diese neuen Architekturen ausnutzen können. Ein möglicher Ansatz ist hierbei das invasive Computing. Dieses ermöglicht es dem Programmierer, die Ressourcennutzung eines Programms feiner zu kontrollieren.

Das IRTSS Betriebssystem, welches eine beispielhafte Implementierung für ein solches invasives System bietet, unterstützt derzeit die Verwendung der Programmiersprachen C, C++ als auch X10. Hiermit wird die Einführung der Programmiersprache Rust als weitere unterstützte Sprache vorgeschlagen. Diese Sprache hat einige wünschenswerte Merkmale, welche interessant für den Gebrauch im invasiven Computing sind.

Inhaltsverzeichnis

1 Einführung

Das berühmte Mooresche Gesetz besagt, dass sich die “Komplexität integrierter Schaltkreise mit minimalen Komponentenkosten regelmäßig verdoppelt”[1]. Diese Beobachtung wurde im Jahre 1965 von Gordon Moore formuliert und erwies sich seither größtenteils als korrekt. Dieser Trend wird sich mit den derzeit verwendeten Fertigungsmethoden jedoch nicht unendlich fortsetzen können und eventuell an physische Grenzen stoßen. Um zukünftig trotzdem eine verbesserte Rechenleistung zu erzielen, wird unter anderem auf Parallelrechner gesetzt. Der Grundgedanke dahinter ist es, mehrere Recheneinheiten zu verwenden, welche gemeinsam ein Problem abarbeiten und somit nicht eine Verbesserung der einzelnen Recheneinheiten benötigen, um eine verbesserte Leistung aufzuweisen.

Der Trend zum parallelen Rechnen kann bereits seit einiger Zeit beobachtet werden; so sind moderne PCs oder Smartphones generell alle mit Mehrkernsystemen ausgestattet. Bei Grafikprozessoren kommen mittlerweile bereits Tausende einzelne Kerne zum Einsatz, so hat beispielsweise die Nvidia GTX 1080 GPU laut Spezifikation [2] 2560 Kerne verbaut.

Durch den Einsatz von parallelem Rechnen entstehen jedoch auch Kosten für den Programmierer, denn das Rechnersystem wird hierdurch komplexer. Der Programmierer muss sicherstellen dass durch die gleichzeitige Verarbeitung durch die einzelnen Recheneinheiten keine Fehler entstehen, es muss also die Kommunikation zwischen den Prozessorelementen bestehen, um solche Fehler zu vermeiden. Außerdem muss der Programmierer auch die vorliegenden Hardwareressourcen

Um die zusätzliche Komplexität des parallelen Rechnens für den Programmierer zu verringern, müssen neue Techniken oder Programmierparadigmen entwickelt werden. Eine solche Idee ist das Invasive Computing, welches es einem Programmierer erlaubt, die Ressourcen in Parallelen Systemen besser zu nutzen. Vor allem bei Systemen mit vielen Kernen ist dieses Paradigma eine interessanter Lösungsansatz. So kann man beispielsweise auf einer Nvidia GTX 1080 GPU mit insgesamt 2560 Kernen ein Programm ausführen, welches dann ein Problem bearbeitet welches auf genau 1000 dieser Kerne ausgeführt wird.

Das Invasive Computing unterstützt derzeit nur die Programmiersprachen C, C++ und X10. Eine interessante Addition hierzu wäre die Sprache Rust, welche einen

Fokus auf die Vermeidung von Fehlern legt.

1.1 Verwandte Arbeiten

Eine verwandte Arbeit ist “Invasive Computing—An Overview” von Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat und Gregor Snelting. Diese illustriert die Grundkonzepte hinter dem Invasiven Computing.

Abgrenzung zu meiner Arbeit

2 Grundlagen und Verwandte Arbeiten

Im folgenden werden die Grundlagen der Programmiersprache Rust, des invasiven Computing und der Rechnerarchitektur SPARC behandelt.

2.1 Rust

Rust ist eine Programmiersprache, welche von Mozilla Research entwickelt wird [**?**]rustWikDe). Die Entwicklung der Sprache begann als persönliches Projekt des Mozilla-Mitarbeiters Graydon Hoare und wird seit 2009 von Mozilla unterstützt [**?**]rustWikDe). 2010 erschien die erste öffentliche Version der Sprache, im Jahr 2015 wurde die erste stabile Version veröffentlicht [**?**]rustWikDe). Sie erfreut sich in der jüngsten Vergangenheit wachsende Beliebtheit bei Programmierern aller Art. So wurde bei einer Umfrage der Webseite stackoverflow.com im Jahre 2016 als beliebteste Programmiersprache bei Entwicklern ermittelt. [3]

Das wohl derzeit prominenteste Projekt, welche Rust verwendet ist der Servo Layout-Engine, welcher gemeinsam von Mozilla und Samsung[4] entwickelt wird. Dieser Layout-Engine soll schrittweise den bisherigen Engine ‘Gecko‘, welcher in C++ geschrieben wurde, im Mozilla Firefox Webbrowser ersetzen und dabei deutlich bessere Leistung vorweisen.

2.1.1 Motivation

Eine der Kernziele der Rust Programmiersprache ist es, sichere Speicherzugriffe zu gewährleisten. Dies wird mithilfe des sogenannten “Ownership”-modells erreicht. Dieses Modell ermöglicht es, fehlerhafte Speicherzugriffe zu vermeiden, welche ein ernsthaftes Sicherheitsrisiko darstellen, denn diese können zu undefiniertem Verhalten führen, welches wiederum auch gezielt als Angriffsvektor genutzt werden kann. Diese Speichersicherheit wird erreicht, ohne dabei von einem Garbage Collector Gebrauch

zu machen. Dies macht Rust zu einer interessanten Alternative zu Systemsprachen wie C, welche nicht sicher bezüglich der Speicherverwaltung sind, als auch zu Sprachen mit Garbage Collector wie X10, welche durch den Garbage Collector in gewissen Situationen und Anwendungsbereichen, beispielsweise bei Echtzeitsystemen, nicht ideal sind.

Eine weitere Angriffsfläche als auch Fehlerquelle in herkömmlichen Systemprogrammiersprachen wie C und C++ ist das unsichere Typsystem. In Rust wird dies vermieden, indem ein typsicheres Typsystem verwendet wird.

Ein weiteres Ziel der Sprache ist die Geschwindigkeit. Rust soll vergleichbare Geschwindigkeiten zu anderen Systemsprachen wie C oder C++ erreichen. Dies kann in Benchmarks zumeist auch bestätigt werden[5]. Benchmarks, welche Rust mit höheren Programmiersprachen wie Java[6] oder Python[7] vergleichen, weisen oft einen signfikanten Vorspruch für Rust auf.

Das effiziente Einbinden mit anderen Sprachen ist eine zusätzliche Motivation hinter der Programmiersprache. So kann Rust beispielsweise mithilfe der sogenannten "Foreign Function Interface" Funktionen welche in anderen Programmiersprachen geschrieben sind benutzen. Außerdem ist es möglich, Rust Code aus anderen Programmiersprachen, inklusive höheren Programmiersprachen wie Python oder Ruby, aufzurufen.

Nebenläufigkeit ist ein weiteres Ziel der Sprache, mithilfe des Architektur der Sprache soll es zwischen einzelnen Threads nicht zu kritischen Wettläufen kommen, welches eines der Hauptfehlerquellen bei parallelem Programmieren beseitigt.

Rust soll dem Programmierer zudem höhere Abstraktionen bieten, welche nicht oder nur gering die Effizienz des Programm beinflussen. Diese sogenannten "Zero-cost Abstractions" erleichtern es Programmierern, welche nur wenig Erfahrung mit Systemsprachen vorweisen können, die Programmiersprache zu verwenden, ohne dabei auf Leistung verzichten zu müssen..

2.1.2 Grundlegende Eigenschaften der Sprache

Im folgenden werden die grundlegenden Eigenschaften der Programmiersprache Rust erläutert.

Das Ownership-Modell

Das signifikanteste Alleinstellungsmerkmal der Programmiersprache Rust ist das Ownership Modell. Erst hierdurch wird die Speichersicherheit ohne Garbage Collector möglich.

Die Grundlage des Ownership-Modells ist die Bindung von Speicher an eine Variable. Sobald eine Variable initialisiert wird, wird Speicher für diese alloziert. Sobald diese Variable sich allerdings nicht mehr im gültigen Anwendungsbereich befindet, wird dieser Speicher automatisch wieder freigegeben. Dies wird im folgenden Beispiel veranschaulicht:

```
fn foo() {
    let v = vec![1, 2, 3]; // Alloziert Speicher
}

fn main() {
    foo();
    // Speicher ist hier wieder freigegeben
}
```

Außerdem erlaubt es Rust keinen zwei Variablen auf denselben Speicher zu zeigen. So wird beispielsweise bei einer Zuweisung einer Variable zu einer anderen der “Besitz” des Speichers auf die neue Variable übertragen und die alte Variable kann nun nicht mehr verwendet werden. Dies geschieht auch wenn die Variable als Parameter in einem Funktionsaufrauf verwendet wird. Dieses Verhalten ist als “Move”-Semantik bekannt. Eine Veranschaulichung dessen folgt:

```
fn foo(v: Vec<i32>) {
    // Details unwichtig
}

fn main() {

    let v = vec![1, 2, 3];
    let v2 = v;  // v ist ab jetzt nicht mehr benutzbar!
    foo(v2);     // v2 ist ab jetzt nicht mehr benutzbar!

}
```

Durch dieses Verhalten wird sichergestellt, dass eine Variable exklusiven Zugriff auf die allozierten Speicherbereiche besitzt, wodurch fehlerhafte Speicherzugriffe vermieden werden.

Eine Ausnahme hierzu bilden Type, welche das “Copy-Trait” implementieren. So können beispielsweise variablen vom Typ i32, welches das “Copy-Trait” implementiert, beliebig oft wiederverwendet werden. In diesem Fall existieren trotzdem nicht mehrere Zeiger auf dieselbe Speicherregion, der Speicherinhalt wird stattdessen jedesmal kopiert.

Ein weiteres interessantes “Trait” welches ein Typ in Rust implementieren kann ist das “Drop-Trait”. Imlementiert ein Typ dieses, so kann zusätzlicher Code ausgeführt werden, sobald eine Variable von dem Typ den gültigen Anwendungsbereich verlässt. Dies kann hilfreich sein, um andere Speicherbereiche, welche in einem nicht-trivialen Zusammenhang von der nun ungültigen Variable abhängen.

Das Ownership-Modell wird in der Praxis von Compiler durchgesetzt. So werden die Garantien, welche das Ownership-Modell verspricht, bereits zur Compile-Zeit sichergestellt. Zusätzlich fällt die Notwendigkeit für einen Garbage Collector weg, welches zu einer besseren Laufzeitseffizienz und Determinismus beitragen.

Typsystem

Rust verfügt über ein statisches, typsicheres Typsystem. Dies bedeutet dass jede Variable einen eindeutigen Typ besitzt und keinen anderen Typ annehmen kann.

Rust erlaubt es zudem, per Typinferenz die explizite Angabe eines Typen bei der Variablendeklaration zu vermeiden.

Typsystem - Hindley-Milner

Typsicher

Lineare Typen / -> Affine Typen <-

Generics

Paradigmen

Mehrere Pardigmen (OOP, Funktional)

Die Programmiersprache Rust lehnt sich syntaktisch an andere C-artige Sprachen an, unterscheidet sich aber hierbei in einigen Aspekten. So sind in ‘if‘/‘else‘ Blöcken beispielsweise die runden Klammern beispielsweise Optional, so wie es auch in Sprachen wie Python der Fall ist.

Fehlerbehandlung

Keine Exceptions -> Result<T,U>

Keine Race Conditions - Sync/Send

2.1.3 Architektur/Compiler

Der offizielle Rust Compiler rustc

Als Lösung zum Abhängigkeitsmanagement und der Distribution wurde das Tool ‘cargo‘ entwickelt. Es ermöglicht es dem Programmierer verwendete Bibliotheken in ein Projekt einzugliedern, indem diese in einer ‘.toml‘ Datei im Hauptverzeichniss des Projekts angegeben werden. Außerdem unterstützt ‘cargo‘ mehrere weiter hilfreiche Funktionen zum testen oder distributieren der entwickelten Software. Kompilierte Software kann als ein sogenanntes ‘Crate‘ bei der von den Rust entwickelten Platform[Citation Needed] ‘crates.io‘ mithilfe von ‘cargo‘ hochgeladen werden.

Etwas zum Cross Compiling

LLVM

2.2 SPARC

Sparc

2.3 Invasives Computing

Invasives Computing ist ein paralleles Programmiermodell, welches es ermöglicht, temporär Ressourcen auf einem Parallelrechner zu beanspruchen und anschließend

wieder freizugeben.Hierbei gibt es drei wesentliche Phasen, die ‘invade‘, ‘infect‘ und ‘retreat‘ Phasen.

In der ‘invade‘ Phase werden zunächst Ressourcen für das laufende Programm reserviert. Welche Ressourcen genau reserviert werden, wird durch ‘constraints‘ bestimmt.

Anschließend wird in der ‘infect‘ Phase die Funktion auf den reservierten Prozessorelementen ausgeführt.

Werden die reservierten Ressourcen nicht mehr benötigt, so kann man diese in der ‘retreat‘ Phase wieder freigeben.

Von WEM? https://invasic.informatik.uni-erlangen.de/en/index.php

3 Entwurf und Implementierung

Im folgenden werden Programme und Bibliotheken entwickelt, welche es ermöglichen, Rust auf dem IRTSS Betriebssystem verwenden zu können. Dies ermöglicht es dann, Programme welche vom invasiven Computing Gebrauch machen, in der Programmiersprache Rust zu schreiben.

3.1 Rust auf der SPARC LEON Architektur

Rust wird derzeit nicht offiziell auf der SPARC-V8 Architektur unterstützt. Rust verwendet jedoch als Backend LLVM, welches diese Architektur unterstützt und daher ist es prinzipiell möglich, Rust-Programme für diese Architektur zu kompilieren. Die Rust-Standardbibliothek ist jedoch nicht trivial auf andere Architekturen zu portieren, Rust bietet allerdings die Funktion, Programme mit einer minimalen, platformunabhängigen Untermenge der Standardbibliothek zu kompilieren. Diese Funktion wird hier ausgenutzt, um eine minimale Implementierung der Programmiersprache auf die SPARC-V8 Architektur zu portieren.

Um ein Rust Programm für eine nicht offiziell unterstütze Architektur zu kompilieren, muss man zuerst die libcore Bibliothek für die Zielarchitektur kompilieren. Um dies zu erreichen, benötigt man eine JSON-Datei, welche dem Compiler die nötigen Informationen zur Ziel-Architektur zur Verfügung stellt. Eine solche JSON Datei für die SPARC-V8 Architektur sieht beispielsweise wie folgt aus[8]:

```
{
    "arch": "sparc",
    "data-layout": "E-m:e-p:32:32-i64:64-f128:64-n32-S64",
    "executables": true,
    "llvm-target": "sparc",
    "os": "none",
    "panic-strategy": "abort",
    "target-endian": "big",
    "target-pointer-width": "32",
```

```
    "linker-flavor": "ld",
    "linker": "path_to_sparc_gcc",
    "link-args": [
        "-nostartfiles"
    ]
}
```

Außerdem wird ein C-Linker, beispielsweise gcc, für die SPARC-V8 Architektur benötigt. Der Pfad zu diesem Linker muss in der `linker`-Option der JSON-Datei angegeben werden.

Sobald man diese JSON Datei erstellt hat, kann man die libcore Bibliothek mit dem "cargo build" Befehl kompilieren. Um dies für SPARC-V8 zu tun, gibt man als Parameter für die "--target"-Option den Pfad zur Spezifikations-JSON-Datei an. Derzeit ist es nicht möglich libcore mit dem stabilen Rust-Compiler zu kompilieren, da diese Bibliothek Funktionalitäten verwendet, welche auf dem stabilen Compiler deaktiviert sind. Daher muss ein nightly-Compiler installiert werden, welches dank rustup jedoch relativ nutzerfreundlich gestaltet ist. Außerderm sollte die Version der libcore Bibliothek mit der Version des nightly-Compilers übereinstimmen um versionsabhängige Konflikte bei der Kompilierung zu vermeiden.

Nachdem die libcore Bibliothek erfolgreich kompiliert wurde kann man, wenn rustup verwendet wurde um rustc und cargo zu installieren, die resultierende .rlib Datei anderen Rust-Programmen beim Kompilieren zur Verfügung zu stellen, indem man diese in das korrekte "lib" Unterverzeichniss im lokalen ".rustup" Verzeichniss kopiert. Alternativ kann man das libcore Projekt direkt durch die Abhängigkeiten in der Cargo.toml Datei eines Cargo Projekts einbinden.

Damit ein Rust-Programm libcore anstelle der Standardbibliothek verwendet, muss man die Zeile

```
#![no_std]
```

zum Anfang des Programms hinzufügen. Außerdem müssen die Funktionen "eh_personality", "eh_unwind_resume" und "panic_fmt" in diesem Programm manuell implementiert werden. Nennenswert ist vor allem letztere, denn diese Funktion wird aufgerufen, sobald ein Programm kontrolliert abstürzt. Der Anfang eines zu kompilierenden Programm muss also beispielsweise wie folgt ausssehen:

```
#![feature(lang_items, libc)]
#![no_std]
```

```
#![no_main]

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "eh_unwind_resume"] extern fn eh_unwind_resume() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

Um dann das Programm zu kompilieren, verwendet man entweder den “rustc” Befehl bei eigenständigen .rs Dateien oder den “cargo” Befehl für Cargo Projekte. Beiden Befehlen muss dann wie auch beim Kompilieren von libcore die Speizifikations-JSON-Datei als Argument für die “--target”-Option angegeben werden.

3.2 Erstellung des octorust Hilfsprogramms

Um das relativ komplizierte und fehleranfällige Kompilieren für Rust-Programme auf der SPARC LEON Architektur zu vereinfachen, als auch besagte Rust-Programme mit dem IRTSS Betriebssytem zu verwenden, wurde ein Python-Programm geschrieben, welche diese Schritte vereinfacht. Das Programm wurde in Python geschrieben, da Python’s Standardbibliothek viele nützliche Funktionen zur Manipulation von Dateien bietet, welches sich für diesen Zweck als hilfreich erweisen. Außerdem bietet Python mit argparse ein praktisches Modul zum Verarbeiten von von Kommandozeile-Argumenten.

Das Programm verwendet python-setuptools und eine dafür konfigurierte setup.py Datei, um das Programm lokal zu installieren. Während dem Installationsprozesses wird zum Einen das octorust-Programm selbst lokal als Python Modul installiert und das “octorust” Skript als ausführbare Datei dem Nutzer zur Verfügung gestellt. Gleichzeitig wird ein Verzeichniss names .octorust im Heimverzeichniss des derzeitigen Nutzers erstellt, in dem IRTSS-builds, die octolib Rust-Bibliothek, welche später genauer erläutert wird, als auch ein SPARC-V8 gcc, insofern dass ein solcher nicht bereits installiert ist und im Pfad gefunden werden kann. Zudem werden für alle unterstützten Architekturen die libcore, libc und liballoc Bibliotheken kompiliert und in den Installationspfad des derzeit verwendeten Rustup-Toolchains kopiert.

Octorust bietet die folgenden Optionen:

- -h, --help

  Diese Option druckt einen Nutzungshinweis inklusive aller möglichen Kommandozeilenoptionen.

- -a. --architecture

Diese Option ermöglicht es, eine IRTSS Zielarchitektur zu wählen. Zur Wahl stehen “x86guest”, “x64native” als auch “leon”. Sollte diese Option nicht angegeben werden, wird standardmäßig für die x86guest Architektur kompiliert.

- -v, --variant

  Diese Option ermöglicht es, eine IRTSS-Variante zu wählen, beispielsweise “generic” oder “4t5c-nores-chipit-w-iotile”. Sollte diese Option nicht angegeben werden, wird je nach gewählter Architektur eine passende Standardvariante verwendet. Im Falle von “x86guest” und “x64native” wird die Variante “generic” gewählt und für “leon” die Variante “4t5c-nores-chipit-w-iotile”.

- -o, --output

  Mit dieser Option kann der Pfad zur resultierenden Ausgabedatei explizit angegeben werden. Ansonsten ermittelt octorust automatisch einen sinnvollen Ausgabenamen, beispielsweise “foo.out” für ein Cargo-Project mit dem Namen “foo”.

- -k, --keep

  Wird diese Option verwendet, werden jegliche temporäre Dateien, die während dem Kompilieren und dem Linken erstellt werden im Anschluss nicht gelöscht. Dies beinhaltet beispielsweise statische Bibliotheken oder Objekt-Dateien.

- -r, --run

  Wird diese Option verwendet, wird das Programm nach dem Kompilieren sofort ausgeführt. Für andere Architekturen als “x86guest” wird dabei vorausgesetzt dass das Virtualisierungsprogramm “qemu” installiert ist.

- -i, --irtss-build-version

  Mithilfe dieser Option kann man eine spezifische IRTSS Version verwenden.

- --release

  Wird diese Option verwendet, werden Compiler-Optimierungen für Rust-Programme aktiviert.

- --fetch-irtss

  Mithilfe dieser Option können IRTSS-builds von https://www4.cs.fau.de/invasic/octopos/

heruntergeladen werden. Hierfür ist jedoch eine gültige Nutzername/Passwort-Kombination in einer .netrc Datei im folgenden Format vonnöten:

```
machine www4.cs.fau.de
login NUTZERNAME
password PASSWORT
```

Unterstützt wird das Kompilieren von eigenständigen .c C-Programmen und .rs Rust-Programmen, als auch das Kompilieren von Cargo-Projekten. Cargo-Projekte bieten durch das Abhängigkeitsmanagement zusätzlich den Gebrauch von Rust-Bibliotheken, vor allem der octolib-Bibliothek, welche eigens für die Interaktion zwischen Rust und IRTSS entwickelt wurde. Daher werden sich die folgenden Prozesse primär mit dem Kompilierungsvorgang der Cargo-Projekte beschäftigen.

3.2.1 Struktur eines invasiven Cargo-Projekts

Wie normale Cargo-Projekte besteht ein invasives Cargo-Projekt aus mindestens einem “src” Verzeichnis, einer Haupt-Bibliotheksdatei und einer “Cargo.toml” Datei. Bei invasiven Cargo-Projekte muss das “crate-type” Attribut immer als “staticlib” angegeben werden, damit das Projekt als statische Bibliothek kompiliert werden kann, welche dann mit dem IRTSS Betriebssystem verlinkt werden kann.

Zusätzlich muss die Haupt-Quelldatei des Projekts mit “#![no_std]” beginnen und anstelle der main() Funktion muss eine “pub extern "C"” Funktion namens “rust_main_ilet” als Startpunkt des Programms verwendet werden. Diese Funktion nimmt genau einen Parameter entgegen, welcher vom Typ u8 ist. Über diese Funktion muss außerdem “#[no_mangle]” stehen, damit die Funktion aus einem C Kontext heraus aufrufbar ist. Außerdem sollte die octolib Bibliothek mit einem “extern crate octolib;” eingebunden werden.

Die Haupt-Quelldatei eines invasiven Cargo-Projekts sähe demnach wie folgt aus:

```
#![no_std]

extern crate octolib;

#[no_mangle]
pub extern "C" fn rust_main_ilet(claim: u8) {

}
```

3.2.2 Kompilieren

Vor der Kompilierung wird im Falle, dass “leon’ als Zielarchitektur ausgewählt wurde, zuerst eine wie in Kapitel 3.1 erläuterte JSON-Spezifikationsdatei generiert, in der der Pfad zum SPARc-v8 gcc Compiler automatisch eingetragen wird. Für die anderen beiden Zielarchitekturen ist ein solcher Schritt nicht nötig, da diese offiziell unterstützt werden. Nachdem diese Spezifikationsdatei generiert wurde, wird das Projekt mithilfe des “cargo”-Befehls als statische Bibliothek kompiliert. Wurde das Projekt erfolgreich kompiliert, wird anschließend eine minimale C Datei generiert, welche die vom IRTSS benötigte main_ilet Funktion implementiert und innerhalb dieser die rust_main_ilet Funktion aufruft. Im Anschluss daran wird dann noch die “shutdown” Funktion verwendet um die Ausführung des Programms zu beenden. Diese generierte C Datei wird mithilfe eines passenden gcc Compilers als object-Datei kompiliert und anschließend mit dem IRTSS und der zuvor kompilierten statischen Rust-Bibliothek verlinkt. Nun sollte eine ausführbare Datei existieren, welche im Falle der “x86guest” Architektur nativ ausführbar ist oder im Falle von “x64native” und “leon” mithilfe eines Emulators wie “qemu”.

Im Anschluss an die Kompilierung werden, vorausgesetzt die “–keep”-Option wurde nicht verwendet, jegliche temporäre Dateien gelöscht, beispielsweise die generierte Cargo.toml Datei, die statische Rust-Bibliothek, die minimale C Datei oder auch die C Objektdatei.

3.3 octolib

Zusätzlich zum octorust Programm wurde ebenfalls eine Rust-Bibliothek names octolib geschrieben. Diese Bibliothek soll die Interaktion zwischen Rust und dem IRTSS Betriebssystem ermöglichen und and die neue Programmiersprache anpassen. IRTSS bietet eine C-Schnittstelle an, welche man mithilfe der Foreign Function Interface und der libc Bibliothek aus Rust heraus ansprechen kann. Die libc Bibliothek bietet die Möglichkeit, Komponenten die die Standardbibliothek benötigen auszulassen, daher ist sie auch auf der SPARC-v8 Architektur benutzbar.

Die octolib Bibliothek wird vor dem Kompilieren als Abhängigkeit in die Cargo.toml Datei eingefügt und so dem Projekt hinzugefügt. Damit die Bibliothek für jedes zu kompilierende Programm auffindbar ist, wird eine lokale Kopie während der Installation des “octorust”-Programms im “ /.octorust” Verzeichnis erstellt.

3.3.1 Struktur

Octolib besteht aus einer Hauptbibliotheksdatei namens “lib.rs”, welche alle anderen Module innerhalb des Projekts einbindet. Diese Datei implementiert außerdem die in Kapitel 3.1 erwähnten Funktionen “eh_personality”, “eh_unwind_resume” und “panic_fmt”, so dass nicht jedes einzelne Projekt diese aufs Neue implementieren muss. Sobald die octolib Bibliothek mit der Anweisung “extern crate octolib” ins Projekt eingebunden wurde, sind diese Funktionen ebenfalls vorhanden.

Die Bibliothek bietet die folgenden Module:

- bindings

  Dieses Modul enthält die direkten C-Rust Bindings zur C-Schnittstelle des IRTSS .

- helper

  Dieses Modul enthält eine Sammlung an Hilfsfunktionen, beispielsweise eine ohne “unsafe”-Block verwendbare print-Funktion, welche von der C printf Funktion Gebrauch macht. Dies ist notwendig da Rust ohne die Standardbibliothek keine äquivalente Funktion besitzt.

- improvements

  Rust-spezifische Abstraktionen, die es dem Programmierer erleichtern, Programme mit Rust für das IRTSS zu schreiben.

- octo_structs

  Rust-äquivalente Structs zu denen, die in der C-Schnittstelle verwendet werden.

- octo_types

  Rust-äquivalente Typen zu denen, die in der C-Schnittstelle verwendet werden.

3.3.2 Direkte C-Rust Bindings

Im ersten Schritt wurden die Funktionen der C Schnittstelle direkt Eins-zu-Eins als Rust-Funktionen eingebunden. Dank der Foreign Function Interface ist dies ein relativ simples Unterfangen, es müssen lediglich die Parameter- und Rückgabetypen

an die neue Programmiersprache angepasst werden. Für die meisten Typen gibt es direkte Äquivalente in Rust, jegliche anderen Typen stellt die libc Bibliothek zur Verfügung. Ein nennenswertes Beispiel eines Typs der nicht in Rust enthalten ist, ist der “void” Typ. Dieser Typ kann beispielsweise durch den c_void Typen aus der libc Bibliothek emuliert werden. Void-Pointer (void*), welche an einigen Stellen der C-Schnittstelle verwendet werden, können so als “*mut c_void” dargestellt werden.

Um die Funktionen erfolgreich in die Rust-Bibliothek einbinden zu können, muss man einen “extern”-Block erstellen und dort die Funktionsdefinitionen vornehmen. Damit diese auch von anderen Modulen oder Projekten aufgerufen werden können, müssen diese als “pub fn“ deklariert werden. Da diese Funktionen lediglich importierte C-Funktionen ohne jegliche Sicherheitsgarantien sind, muss man bei jedem Gebrauch dieser Funktionen einen “unsafe”-Block verwenden.

Structs und Typen

Zusätzlich zu den Funktionsdefinitionen mussten die in IRTSS definierten structs und Typen ebenfalls portiert werden. Dank der #[repr(C)] Anweisung kann man in Rust Structs erstellen, welche kompatibel mit den C-Äquivalenten sind. Diese Structs sind teils von Platform-spezifischen Konstanten abhängig, welche man in Rust mit der #[cfg(target_arch = ärch")] Anweisung für unterschiedliche Architekturen definieren kann. benutzerdefinierte Typen lassen sich in Rust durch Schlüsselwort “type” erstellen.

3.3.3 Rust-spezifische Verbesserungen

Im folgenden werden verschiedene Abstraktionen über die direkten C-Rust Bindings erstellt, welche das Programmieren vereinfachen und die Stärken von Rust ausnutzen sollen.

Constraints

Die “Constraints” Struktur ist eine leichte, objektorientierte Abstraktion für die Constraints, mit denen die Anzahl Ressourcen, die einem Claim zur Verfügung stehen, limitiert werden. Es bietet einen Konstruktor, welcher eine neue “constraints_t” Struktur mithile der “agent_constr_create” Funktion aus der C-Schnittstelle erstellt und anschließend einige Standardwerte setzt. Dem Konstruktor werden zudem 2 Parameter übergeben, welche die minimale und maximale Anzahl an Prozessorelementen

spezifizieren.

Alle Parameter der Constraints lassen sich nachträglich durch Methoden der Struktur ändern. Hierbei profitiern vor allem die Methoden die “bool”-Werte als Parametertypen besitzen, denn diese werden in den direkten C-Rust Bindings statt mit “bool”-Werten mit u8-Werten aufgerufen. Dies liegt daran, dass es in C keinen “bool” oder ähnlichen Typen gibt, dort werden Zahlenwerte auf den Wert 0 geprüft.

Außerdem unterstützt die Methode “merge_constraints” anstelle einer “constraints_t” Struktur aus der C-Schnittstelle eine in Rust definierte “Constraints” Struktur.

Um Zugriff auf die “constraints_t” Struktur zu erhalten, muss die “to_constraints_t” Methode aufgerufen werden.

Closures

Rust Closures können nicht ohne Weiteres einer C-Schnittstelle als Parameter übergeben werden. Somit ist es mit den direkten C-Rust bindings nicht möglich, Closures zu verwenden, um Code auf den Rechenelementen auszuführen. Dies wäre jedoch eine wünschenswerte Funktionalität. Da Closures aus der Funktion selbst als auch ihrem Erstellungskontext[9] bestehen, ist es nicht möglich die Closure direkt zu einer normalen Rust Funktion zu konvertieren. Stattdessen erstellt man einen Pointer auf die Datenregion, welche die Closure repräsentiert und übergibt diesen dann einer “extern "C"” Funktion, welche aus diesem Pointer die Closure zurückgewinnt. Diese “extern "C"”-Funktion kann dann der C-Schnittstelle zusammen mit dem Pointer auf die Closure-Daten übergeben werden.

Die Erstellung des Closure-Pointers wird direkt in der später genauer erläuterten “infect”-Methode der “AgentClaim”-Struktur erledigt, während es zur Rückkonvertierung eine “extern "C"”-Funktion namens execute_closure erstellt wurde.

AgentClaim

Die größte Abstraktion in der octolib Bibliothek ist die “AgentClaim” Struktur. Diese bietet zunächst einmal eine Abstraktionsschicht über die ‘agentclaim_t‘ Struktur aus der C-Schnittstelle und verschiedene Methoden um diese zu verwenden.

Der Konstruktor nimmt als einzigen Parameter eine “Constraints”-Struktur entgegen, welche verwendet wird um die Ressourcen des Claims zu definieren. Mit diesen

Constraints wird dann eine interne “agentclaim_t”-Struktur initialisiert, welche für alle folgenden Methodenaufrufe verwendet wird.

Eine praktische Methode fürs Debugging ist “set_verbose”. Diese Methode erlaubt es, die Ausführlichkeit der auf die Kommandozeile gedruckten Informationen zu beeinflussen.

Die “reinvade”-Methode elaubt es, eine “Reinvade”-Operation auf dem Claim auszuführen. Diese Methode nimmt einen optionalen Parameter an, mit dem man neue Constraints setzen kann. Werden neue Constraints gesetzt, werden die alten Constraints aus dem Speicher gelöscht.

Die wohl wichtigste Methode der “AgentClaim”-Struktur ist die “infect”-Methode. Mit dieser kann eine “Infect”-Operation auf dem Claim ausgeführt werden. Dieser Methode wird eine Funktion oder eine Closure als Parameter übergeben, welche dann auf den Rechenelementen des Claims ausgeführt werden. Zusätzlich können ebenfalls Parameterdaten für die einzelnen Recheneinheiten übergeben werden. In der “infect”-Methode werden Signale verwendet, um die Kommunikation zwischen verschiedenen Rechenelementen zu verwalten. Für jedes Tile und Rechenelement wird iteriert und ein ilet aus der übergegebenen Funktion generiert, welches dann der C-Schnittstelle übergeben wird und so dann ausgeführt wird. Anschließend wartet die Methode dann auf den erfolgreichen Abschluss aller Ilets.

Zusätzlich zur normalen “infect”-Methode gibt es auch eine “infect_async”-Methode, welche nicht auf das Ende der Ausführung aller Rechenelemente wartet, sondern stattdessen wird das Signal, mit dem die Rechenelemente signalisieren dass sie fertig ausgeführt wurden, als Rückgabewert verwendet. Anschließend kann der Nutzer dann mithilfe dieses Signals manuell auf die Ausführung der Rechenelemente warten.

Das bereits in Kapitel 2 erwähnte Drop-Trait für Rust Strukturen wird für die “AgentClaim”-Struktur implementiert. Verlässt eine Instanz der “AgentClaim”-Struktur den gültigen Anwendungsbereich, wird eine “Retreat”-Operation auf der internen “agentclaim_t”-Struktur ausgeführt und zusätzlich die Contraints des Claims gelöscht. Dieses implizite Retreat stellt sicher, dass die vom Claim verwendeten Ressourcen nach dem Gebrauch wieder freigegeben werden und somit anderen Programmteilen zur Verfügung stehen.

4 Evaluation

Im folgenden wird der Gebrauch von Rust im Zusammenhang mit dem invasiven Computing evaluiert. Hierbei wird vor allem mit den bereits unterstützten Sprachen C und X10 verglichen.

4.1 Laufzeitverhalten, Kompilierungsdauer und Dateigröße

Zu Beginn werden das Laufzeitverhalten, die Kompilierungsdauer und die Dateigröße kompilierter Programme zwischen den drei Programmiersprachen verglichen.

Es wurde ein Python-Script geschrieben, welches es ermöglicht, die unterschiedlichen Benchmark-Programme nacheinander abzuarbeiten und währenddessen die Compilezeit, Dateigröße und Laufzeitdauer jedes Programms zu messen.

Alle Programme wurden auf der folgenden Hardware/Software Konfiguration ausgeführt:

- CPU: Intel Core i5-5200U @ 2.2GHZ x 2
- RAM: 8GB
- OS: Antergos Linux, Kernel 4.12.13-1-ARCH
- gcc: 6.3.0
- IRTSS: 2017-06-07-nightly
- rustc: 1.19.0-nightly
- jdk: openjdk 1.8.0_131

- octorust: Version 1.0.0
- x10i: Commit

Im Falle der Programmiersprache Rust wurden immer zwei Szenarien betrachtet: Einmal wenn mit Compiler-Optimierungen kompiliert wurde und einmal ohne. Die Variante ohne Optimierungen wird als "Rust (Debug)" kenntlich gemacht, wohingegen die Variante mit Optimierungen als "Rust (Release)" bezeichnet wird.

4.1.1 Vergleich der Anlaufzeit

Es wird überprüft, wie lange ein Programm, welches in einer der respektiven Programmiersprachen geschrieben wurde, benötigt, um die Ausführung zu starten und anschließend wieder aufzuhören.

Sprache	Laufzeit	Compilezeit	Dateigröße
C	0	0	0
Rust (Debug)	0	0	0
Rust (Release)	0	0	0
X10	0	0	0

Abbildung 4.1: Diese Tabelle stellt die Messergebnisse des startup-Benchmarks dar. Dieser Benchmark simuliert das Starten und sofortige Schließen eines Programms.

Anhand dieser Werte kann man prinzipiell erkennen, dass die Anlaufzeiten von C und Rust sich sehr nahe sind, während X10 hierfür ca. 2 Sekunden länger benötigt. Außerdem benötigt das X10 Programm weitaus länger um die Kompilierung auszuführen, in diesem Fall dauert die Kompilierung eines X10 Programms X-mal so lange wie die eines Rust-Programms. Zudem sind die Dateigrößen der kompilierten X10-Programme in diesem Beispiel x-mal größer als im Falle eines Rust-Programms. C-Programme weisen hierbei die insgesamt kürzesten Kompilierungszeiten und Dateigrößen auf.

4.1.2 Berechnen von Primzahlen

Um die Rechenleistung der verschiedenen Programmiersprachen bei einem intensiveren Problem zu vergleichen, wurden Programme geschrieben, welche Primzahlen berechnen. Hierbei wurden zwei unterschiedliche Ansätze verwendet: Zum einen

eine naive Berechnung, welche jede Zahl individuell auf Teilbarkeit mit kleineren Zahlen prüft und andererseits das Sieb von Eratosthenes, eine effiziente Methode zum Berechnen von Primzahlen.

Im Falle des Siebs von Eratosthenes kann Rust keine Quadratwurzeloperation durchführen, da diese die Standardbibliothek benötigen. Daher wurden in allen Sprachen anstelle eine Quadratwurzelfunktion feste Zahlen verwendet, um den Vergleich zwischen den Sprachen gerecht zu gestalten. Da jedoch nur eine Quadratwurzeloperation für das Sieb des Eratosthenes benötigt wird, wäre dies ohnehin aller Wahrscheinlichkeit nach kein entscheidender Faktor bei der Laufzeit.

Sprache	Laufzeit	Compilezeit	Dateigröße
C	0	0	0
Rust (Debug)	0	0	0
Rust (Release)	0	0	0
X10	0	0	0

Abbildung 4.2: Diese Tabelle stellt die Ergebnisse des naive-primes-Benchmarks dar. Dieser Benchmark berechnet die ersten 50000 Primzahlen indem er für jede Zahl individuell alle kleinere Zahlen auf Teilbarkeit prüft.

Sprache	Laufzeit	Compilezeit	Dateigröße
C	0	0	0
Rust (Debug)	0	0	0
Rust (Release)	0	0	0
X10	0	0	0

Abbildung 4.3: Diese Tabelle stellt die Ergebnisse des eratosthenes-primes-Benchmarks dar. Dieser Benchmark berechnet die ersten 1690000 Primzahlen mithilfe des Siebs von Eratosthenes.

Hier erkennt man bereits einige Leistungsunterschiede zwischen den Programmiersprachen. Während C eindeutig das beste Laufzeitverhalten aufweist, hinkt Rust nicht signifikant hinterher, während das Laufzeitverhalten des X10-Programms auch nach Abzug der größeren Anlaufzeit schlechter als die der beiden Systemsprachen abschneidet.

4.1.3 Müll-Ersteller

X10 verwaltet den Speicher mithilfe eines Garbage Collectors, wobei C und Rust ohne einen solchen auskommen. Während Garbage Collector ein sehr hilfreiches Werkzeug sind, um den Programmieraufwand zu verringern, so kommt dies allerdings auch auf Kosten der Laufzeiteffizienz. Vor allem die Garbage-Collector-Pausen sind hierbei ein nicht zu unterschätzender Faktor.

Um diese Leistungsdifferenz zu veranschaulichen, wurde ein Benchmark-Programm geschrieben, welche kontinuierlich Objekte auf dem Heap erstellt, welche anschließend wieder aus dem gültigen Anwendungsbereich verschwindet. In C muss der Speicher, in dem diese Objekte gespeichert werden, manuell befreit werden, in Rust werden diese automatisch ungültig sobald sie den gültigen Anwendugsbereich verlassen und bei X10 kümmert sich der Garbage Collector darum.

Die erstellten Objekte sind in diesem Fall Arrays von Integer-Zahlenwerten. In Rust sind das Äquivalent zu Arrays Slices und in X10 heißen diese Rails.

Sprache	Laufzeit	Compilezeit	Dateigröße
C	0	0	0
Rust (Debug)	0	0	0
Rust (Release)	0	0	0
X10	0	0	0

Abbildung 4.4: Diese Tabelle stellt die Ergebnisse des garbageonly-Benchmarks dar. Dieser Benchmark erstellt insgesamt 1000000 Arrays mit einer Größe von 5000.

Wie man an den Ergebnissen des Benchmarks erkennen kann, benötigt X10 deutlich länger um die selbe Anzahl an gleich großen Objekten zu erstellen als C oder Rust es tun. Dies weist darauf hin, dass der Garbage Collector unter Umständen einen signifikanten Einfluss auf das Laufzeitverhalten haben kann. C und Rust haben hier also einen Vorteil, vor allem Rust, denn bei dieser Sprache muss der Speicher nicht manuell wieder freigegeben werden, wie es in C der Fall ist.

4.2 Sicherheit

Im folgenden wird die Sicherheit bezüglich undefiniertem Verhalten als Folge von fehlerhaften Speicherzugriffen analysiert. Hierbei ist vor allem der Vergleich zwischen

Rust und C interessant, da einige von Rusts primären Eigenschaften die Risiken der C-Programmierung beseitigen sollen.

4.2.1 Division durch 0

Dividiert man in C einen Wert durch die Zahl 0, kann es zu undefiniertem Verhalten führen, welches generell unerwünschtes Verhalten darstellt. Versucht man dies in Rust, so stürzt das Programm sofort ab, es kann nicht zu undefiniertem Verhalten kommen.

4.2.2 Pufferüberlauf

Ein weiterer Fall, der in C zu undefiniertem Verhalten führt, sind Pufferüberläufe. Diese geschehen, wenn man beispielsweise in einem Array der Größe n versucht, auf das n+1te Element zuzugreifen. Wie bereits im letzten Vergleich kann dies in Rust nicht geschehen, da das Programm abstürzt.

4.2.3 Nicht Initialisierte Variablen

In C ist es möglich, uninitialiserte Variablen zu verwenden, dies führt allerdings zu undefiniertem Verhalten. In Rust hingegen wird dies bereits vom Compiler verhindert, da er den Gebrauch von uninitialisierten Variablen verbietet. Ein Rust-Programm welches also uninitialisierte Variablen verwendet kompiliert also gar nicht und kann so natürlich auch nicht zu undefiniertem Verhalten führen.

4.3 Abstraktionen

Es werden nun die implementierten Abstraktionen der octolib Bibliothek mit den Implementierungen in C und X10 verglichen.

4.3.1 Minimales Infect

4.3.2 Cleanup

4.3.3 Closures

Closures bieten eine praktische Art und Weise, anonyme Funktionen zu nutzen

5 Fazit und Ausblick

Zusammenfassend ist

5.1 Fazit

5.2 Ausblick

Standardbibliothek

Echter Support

Crates.io

Literaturverzeichnis

[1] wikipedia.org, “Mooresches Gesetz,” 2017.

[2] Nvidia, “GeForce GTX 1080 Specifications,” 2017.

[3] stackoverflow.com, “Developer survey results 2016,” 2016.

[4] wikipedia.org, “Servo (Software),” 2017.

[5] debian.org, “Rust programs versus C gcc,” 2017.

[6] debian.org, “Rust programs versus Java,” 2017.

[7] debian.org, “Rust programs versus Python 3,” 2017.

[8] J. Aparicio(japaric), “initial SPARC support,” 2017.

[9] wikipedia.org, “Closure (Funktion),” 2017.

Erklärung

Hiermit erkläre ich, Hermann Krumrey, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Danke

Ich danke meinen Eltern, die mich meine gesamtes Leben lang durch dick und dünn begleitet und unterstützt haben. Ich danke meinem Bruder, in dem ich einen ewigen Kumpanen im Leben gefunden habe. Ich danke meinem Betreuer Andreas Zwinkau, welcher mich freundlich und hilfreich durch die Erstellung dieser Arbeit begleitet hat. Außerdem danke ich meinen guten Freunden Simon Eherler und Frederick Horn, ohne die ich nicht der Mensch wäre der ich heute bin. Ich danke meinen Kommilitonen Marius Take, Johannes Bucher, Thomas Schmidt und Daniel Mockenhaupt, ohne die das Studium nicht halb so schön wäre. Ich danke meiner "Ersatzfamilie", der Familie Eherler, die immer einen Platz in ihrer Mitte für mich haben. Und zu guter Letzt danke ich dem Karlsruher Institut für Technologie, welches es mir erst ermöglichte, dieses Studium zu absolvieren.