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 **7**
1.1 Verwandte Arbeiten 8

2 Grundlagen und Verwandte Arbeiten **9**
2.1 Rust 9
2.1.1 Motivation 9
2.1.2 Grundlegende Eigenschaften der Sprache 10
2.1.3 Architektur/Compiler 13
2.2 SPARC 13
2.3 Invasives Computing 13

3 Entwurf und Implementierung **15**
3.1 Rust auf der SPARC LEON Architektur 15
3.2 Erstellung des octorust Hilfsprogramms 16
3.2.1 Struktur eines Cargo-Projekts 18
3.2.2 Kompilieren 19
3.3 octolib 20
3.4 Rust-spezifische Verbesserungen 20

4 Evaluation **21**
4.1 Laufzeitverhalten und Dateigröße 21
4.1.1 Vergleich der Anlaufzeit 21
4.1.2 Berechnen von Primzahlen 21
4.1.3 Müll-Ersteller 22
4.2 Sicherheit 22
4.2.1 Undefiniertes Verhalten 22
4.3 Abstraktionen 22
4.4 Minimales Infect 23
4.5 Cleanup 23
4.6 Closures 23

5 Fazit und Ausblick **25**
5.1 Fazit 25
5.2 Ausblick 25

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.

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

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

3.1 Rust auf der SPARC LEON Architektur

Rust wird derzeit nicht offiziell auf der SPARC LEON/SPARCv8 Architektur unterstützt. Rust verwendet jedoch als Backend LLVM, welches diese Architektur unterstützt, daher ist es 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 minimalem, platformunabhängigen Untermenge der Standardbibliothek zu kompilieren. Diese Funktion wird hier ausgenutzt, um eine minimale Implementierung der Programmiersprache auf die SPARC LEON Architektur zu portieren.

Um ein Rust Programm für eine nicht offiziell unterstütze Architektur zu kompilieren, muss man eine JSON-Datei erstellen, welche dem Compiler die nötigen Informationen zur Ziel-Architektur zur Verfügung stellt. Eine solche JSON Datei für die SPARC LEON Architektur sieht 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 LEON Architektur benötigt. Der Pfad zu diesem Linker muss in der linker -Option der JSON-Datei angegeben werden.

Da die Standardbibliothek für die SPARC LEON Architektur nicht verfügbar ist, muss man stattdessen das Core-Crate/libcore verwenden. 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. Am Anfang eines zu kompilierenden Programm muss also beispielsweise die folgenden Zeilen angegeben werden:

```
#![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 Core-Crate einzubinden, verwendet man idealerweise ein Cargo-Projekt und gibt den Pfad zum Core-Crate in der Cargo.toml Datei als Abhängigkeit an. Anschließend kann man das Programm mithilfe des Befehls “cargo rustc --target leon.json” kompilieren, wobei leon.json der Name der zuvor erstellten JSON Datei ist. Dies kompiliert nun zuerst das Core-Crate für die SPARC LEON Architektur und anschließend das Programm selbst.

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 in diesem Fall als hilfreich erweist. Außerdem bietet Python eine simple Schnittstelle zum Parsen von Kommandozeile-Argumenten.

Das Programm verwendet setuptools und eine dafür konfigurierte setup.py Datei um das Programm lokal zu installieren. Während dem Installationsprozesses wird gleichzeitig 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 gcc, insofern dass ein solcher nicht bereits installiert und im Pfad gefunden werden kann.

Octorust soll die folgenden Optionen bieten:

- -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 automtisch einen sinvollen 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.

- -r, --run

  Wird diese Option verwendet, wird das Programm nach dem Kompilieren ausgeführt direkt ausgeführt. Für andere Architekturen wird dabei vorausgesetzt dass qemu installiert ist.

- -i, --irtss-build-version

  Mithilfe dieser Option kann man eine spezifische IRTSS Version verwenden.

- --release

  Wird diese Option verwendet, werden 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 Cargo-Projekts

Cargo-Projekte die mit octorust kompiliert werden sollen müssen einige Eigenheiten im Vergleich zu normalen Cargo-Projekten aufweisen. Zum einen werden die Projektdaten nicht in eine Cargo.toml, sondern in ein OctoCargo.toml Datei eingegeben. In dieser OctoCargo.toml Datei muss außerdem der Abschnitt “[dependencies]” als letztes vorkommen. Dies ist der Fall, da octorust aus dieser OctoCargo.toml

die eigentlich Cargo.toml Datei generiert und dabei benötigte Abhängigkeiten einfügt. Außerdem muss in dieser OctoCargo.toml Datei das crate-type Attribut als `["staticlib"]` angegeben werden. Eine minimale OctoCargo.toml Datei sieht demnach wie folgt aus:

```
[package]
name = "project_name"
version = "project_version"
authors = ["project_authors"]

[lib]
crate-type = ["staticlib"]

[dependencies]
```

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. Ein solches minimales Rust-Programm sähe also wie folgt aus:

```
#![no_std]

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

}
```

3.2.2 Kompilieren

Vor der Kompilierung des eigentlichen Cargo-Projekts werden zunächst alle Abhängigkeiten für die gewählte Architektur kompiliert. Für x86guest und x64native wird hierbei nur das korrekte Target-Tripel beim Kompilieren angegeben, x86_64-unknown-linux-gnu für x64native und i686-unknown-linux-gnu für x86guest. Soll jedoch für die leon Architektur kompiliert werden, generiert octorust zuerst eine wie in Kapitel 3.1 erläuterte JSON Datei, die dann als Target verwendet wird. Die Abhängigkeiten, welche in jedem von octorust kompilierten Projekt verwendet werden, sind libcore, libc und octolib. Diese Abhängigkeiten befinden sich nach einer erfolgreichen Installation alle im /.octorust Verzeichnis.

Nachdem die Abhängigkeiten erfolgreich kompiliert wurden, wird das Cargo-Projekt selbst als statische Bibliothek kompiliert. Anschließend wird 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 erwendet 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. Nun sollte die Kompilierung erfolgreich beendet sein, vorausgesetzt dass keine Compilerfehler aufgetreten sind.

Im Anschluss an die Kompilierung wird, vorausgesetzt die "–keep"-Option wurde nicht verwendet, jegliche temporäre Dateien gelöscht, welches die generierte Cargo.toml Datei, die minimale C Datei als auch die C object-Datei.

3.3 octolib

Im folgenden wurde eine Rust Bibliothek geschrieben, welche

3.4 Rust-spezifische Verbesserungen

AgentClaim / Constraints Structs

infect

Implizites retreat

Closures

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 und Dateigröße

Zu Beginn werden

4.1.1 Vergleich der Anlaufzeit

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

Startup

Auswertung

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.

Naiv

Eratosthenes

Auswertung

4.1.3 Müll-Ersteller

X10 verwaltet den Speicher mithilfe eines Garbage Collectors, wobei C und Rust ohne einen solchen auskommen. In C muss jedoch

GarbageOnly

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

4.2.1 Undefiniertes Verhalten

Undefiniertes Verhalten (Wird in rust vom Kompiler verhindert)

4.3 Abstraktionen

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

4.4 Minimales Infect

4.5 Cleanup

4.6 Closures

5 Fazit und Ausblick

Zusammenfassend ist

5.1 Fazit

5.2 Ausblick

Standardbibliothek

Echter Support

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.

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 Ërsatzfamilie", 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.