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 neuen 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[1]. 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 gesteuert werden, um mögliche Fehler zu vermeiden. Außerdem muss der Programmierer auch die vorliegenden Hardwareressourcen beachten.

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. Gleichzeitig kann dann anderes Problem auf einer weiteren Untermenge der Kerne ausgeführt werden.

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 Sicherheit vor schwerwiegenden Programmierfehlern, die vor allem durch fehlerhafte Speicherzugriffe ausgelößt werden, legt.

1.1 Verwandte Arbeiten

Eine verwandte Arbeit ist „Invasive Computing—An Overview“[3] von Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat und Gregor Snelting. Diese illustriert die Grundkonzepte hinter dem invasiven Computing, befasst sich jedoch im Gegensatz zu dieser Arbeit nicht mit der Programmiersprache Rust.

2 Grundlagen und Verwandte Arbeiten

Im Folgenden Kapitel werden die Grundlagen der Programmiersprache Rust, der Rechnerarchitektur SPARC-V8 und des invasiven Computing behandelt, welche zum Verständnis dieser Arbeit beitragen.

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 offiziell unterstützt [?]rustWikDe). 2010 erschien die erste öffentliche Version der Sprache und 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 Rust im Jahre 2016 als beliebteste Programmiersprache bei Entwicklern ermittelt[4].

Das wohl derzeit prominenteste Projekt, welches Rust verwendet, ist der Servo Layout-Engine, welcher gemeinsam von Mozilla und Samsung entwickelt wird[5]. Dieser Layout-Engine soll nach und nach im Mozilla Firefox Webbrowser integriert werden und hierbei eine bessere Leistung als vorhergehende Technologien vorweisen[5].

2.1.1 Motivation

Eine der Kernziele der Rust Programmiersprache ist es, sichere Speicherzugriffe zu gewährleisten[6]. Dies wird mithilfe des sogenannten Ownership-Systems und einem besonderen, typsicheren Typsystem, wie es bei funktionalen Programmiersprachen oft der Fall ist, [6] erreicht. Das Ownership-System ermöglicht es, fehlerhafte Speicherzugriffe, welche ein ernsthaftes Sicherheitsrisiko darstellen, zu vermeiden. Fehlerhafte Speicherzugriffe können zu undefiniertem Verhalten führen, welches wiederum auch

gezielt als Angriffsvektor genutzt werden kann. Diese Speichersicherheit wird in Rust erreicht, anders als bei den meisten anderen Sprachen mit automatischer Speicherverwaltung, ohne dabei von einem Garbage Collector Gebrauch zu machen[6]. 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 wiederum andere Nachteile vorweisen.

Ein weiteres Ziel der Sprache ist die Leistung. Rust soll vergleichbare Leistungen bezüglich des Laufzeitverhaltens wie andere Systemsprachen wie C oder C++ erreichen. Dies kann in Benchmarks zumeist auch bestätigt werden[7]. Benchmarks, welche Rust mit höheren Programmiersprachen wie Java[8] oder Python[9] vergleichen, weisen meist eine bessere Leistung von 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“ (FFI) Funktionen, welche in anderen Programmiersprachen, beispielsweise C, 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 beeinflussen. 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-System

Das signifikanteste Alleinstellungsmerkmal der Programmiersprache Rust ist das Ownership-System[10]. Erst hierdurch wird die Speichersicherheit ohne Garbage

Collector möglich.

Die Grundlage des Ownership-Systems ist die Bindung von Speicher an eine Variable[10]. Sobald eine Variable initialisiert wird, wird Speicher für diese alloziert. Sobald diese Variable sich allerdings nicht mehr im Geltungsbereich befindet, wird dieser Speicher automatisch wieder freigegeben[10]. Dies wird im folgenden Beispiel veranschaulicht[11]:

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

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

Außerdem erlaubt es Rust keinen zwei Variablen auf denselben Speicher zu zeigen[10]. 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[11]:

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

fn main() {

    let v = vec![1, 2, 3];
    let w = v;  // v ist ab jetzt nicht mehr benutzbar!
    foo(w);     // w 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“ oder das „Clone-Trait“ implementieren[11][12]. So können beispielsweise Variablen vom Typ i32, welches das „Copy-Trait“ implementiert, beliebig oft wiederverwendet werden[11]. In diesem Fall existieren trotzdem nicht mehrere Zeiger auf dieselbe Speicherregion, der Speicherinhalt wird stattdessen jedes Mal kopiert.

Ein weiteres interessantes „Trait“ welches ein Typ in Rust implementieren kann ist das „Drop-Trait“[13]. Implementiert ein Typ dieses, so kann zusätzlicher Code ausgeführt werden, sobald eine Variable von dem Typ den Geltungsbereich verlässt[13]. Dies kann hilfreich sein, um Abhängigkeiten zu behandeln, welche in einem nicht-trivialen Zusammenhang von der nun ungültigen Variable abhängen. Beispielsweise kann eine Netzwerkverbindung korrekt geschlossen werden sobald die dazugehörige Variable den Geltungsbereich verlässt.

Das Ownership-System wird in der Praxis von Compiler durchgesetzt. So werden die Garantien, welche das Ownership-System verspricht, bereits zur Compile-Zeit sichergestellt. Zusätzlich fällt die Notwendigkeit für einen Garbage Collector weg, welches zu einer besseren Laufzeitseffizienz und Determinismus im Vergleich zu Sprachen die einen Garbage Collector verwenden beitragen. Durch das Ownership-System entstehen keine zusätzlichen Laufzeitkosten[10].

Typsystem

Rust verfügt über ein statisches, typsicheres Typsystem, welches sich an Typsystemen aus funktionalen Sprachen, beispielsweise Haskell, anlehnt[6]. 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[6].

Es gibt wie in den meisten statischen Typsystemen eine Handvoll von primitiven Typen. In Rust gibt es die folgenden[14]:

Vorzeichenlose numerische Typen u8, u16, u32, u64

Vorzeichenbehaftete numerische Typen im Zweier-Komplement i8, i16, i32, i64

Platformabhängige numerische Typen usize, isize

Textuelle Typen char, str

Zudem gibt es zusätzlich noch generische Konstrukte, nämlich Tupel, Arrays, Slices, Funktionszeiger, Referenzen und Zeiger, welche auf beliebige Typen anwendbar

sind[14]. Funktionen in Rust erlauben den Gebrauch für generische Typen, welches es ermöglicht, dass eine Implementierung einer solche Funktion auf mehrere Typen anwendbar ist.

Rust borgt sich Ideen von substrukturellen Typsystemen[12]. Standardmäßig verhalten sich Variablen in Rust wie affine Typen, welches sicherstellt, dass die Variablen höchstens ein mal verwendet werden[15]. Dies wird in der Funktionsweise der „Move“-Semantik verdeutlicht[12]. Es ist jedoch auch möglich, durch den Gebrauch der „Copy“- oder „Clone“-Traits Variablen beliebig oft zu verwenden[12]. Außerdem ist es möglich, Variablen sich wie lineare Typen verhalten zu lassen, also erzwingen dass diese Variablen mindestens ein mal verwendet werden[12].

Rusts Objektmodell basiert auf Strukturen, Implementierungen und Traits[16]. Strukturen (Schlüsselwort struct) ermöglichen die Deklaration von Feldern[16], Traits (Schlüsselwort trait) bieten Polymorphie und Vererbung [16] und Implementierungen (Schlüsselwort impl) erfüllen eine ähnliche Rolle wie Klassen in anderen Programmiersprachen[16].

Paradigmen

Unterschiedliche Programmierparadigmen nahmen einen Einfluss auf das Sprachdesign von Rust[6]. Unter anderem sind Elemente der funktionalen, objektorientierten und nebenläufigen Programmierung anzutreffen[6]. Dies ermöglicht es unterschiedlichsten Programmierern Rust zu benutzen als auch ein hohes Abstraktionsniveau zu bieten[6].

Fehler- und Ausnahmebehandlung

In der Programmiersprache Rust werden Ausnahmen behandelt, in dem die „Option“ oder „Result“ Aufzählung (enum) als Rückgabeparameter einer Funktion verwendet werden. Anhand dieser kann man dann prüfen, ob eine Ausnahme beim Funktionsaufruf aufgetreten ist. Bei unbehandelten Ausnahmen wird für den aktiven Faden (Thread) die „panic_fmt“-Funktion aufgerufen. Bei Fehlern beendet Rust die Ausführung des Programms.

2.1.3 Architektur/Compiler

Der offizielle Rust Compiler heißt rustc und kann eigenständige Rust-Quelldateien kompilieren. Rust-Quelldateien haben konventionsgemäß die Dateiendung „.rs“. Als Lösung zum Abhängigkeitsmanagement und der Distribution wurde das Werkzeug cargo entwickelt. Es ermöglicht es dem Programmierer verwendete Bibliotheken in ein Projekt einzugliedern, indem diese in einer „Cargo.toml“ Datei im Hauptverzeichniss des Projekts angegeben werden. Außerdem unterstützt „cargo“ mehrere weiter hilfreiche Funktionen zum Testen oder Distribuieren der entwickelten Software. Kompilierte Software kann als ein sogenanntes „Crate“ bei der von den Rust entwickelten Platform „crates.io“ mithilfe von cargo hochgeladen werden.

Rustc und cargo ermöglichen das Kompilieren für unterschiedliche Zielarchitekturen mithilfe der „--target“-Option.

Um den Gebrauch von unterschiedlichen rustc und cargo Versionen zu vereinfachen, wurde das Werkzeug rustup entwickelt. Dieses Hilfsprogramm ermöglicht es unterschiedliche Varianten des Rust-Compilers zu installieren und bei Belieben zu wechseln. Außerdem erleichtert es die Kompilierung für anderer Zielarchitekturen, indem es für unterstützte Architekturen eine vorkompilierte Standardbibliothek herunterladen kann.

Als Compiler-Backend wird LLVM verwendet. Durch diese wird Rust vor der Übersetzung zu Maschinenbefehlen in die Zwischensprache LLVM-IR übersetzt. Dadurch ist es möglich, Rust-Code auf allen von LLVM unterstützen Rechnerarchitekturen laufen zu lassen.

2.2 SPARC-V8

SPARC (**S**calable **P**rocessor **ARC**hitecture) ist eine Mikroprozessorarchitektur, welche von Sun Microsystems und seit 2010[17], nach der Übernahme von Sun Microsystems, von Oracle entwickelt wird[18]. Sie kommt kommerziell derzeit größtenteils in Produkten von Oracle zum Einsatz[18].

Die aktuelle 32-bit Variante der SPARC-Architektur wird als SPARC-V8 bezeichnet[18]. Die Byte-Reihenfolge von SPARC-V8 ist komplett Big-Endian[18].

2.2.1 LEON

Die ursprünglich von der European Space Agency (ESA) und anschließend von Aeroflex Gaisler entwickelten LEON-Prozessoren basieren auf der SPARC-V8 Architektur[19] Der erste LEON Prozessor war das erste vollständige Mikroprozessor-Design welches unter einer Open-Source Lizenz, in diesem Fall der GNU Lesser General Public License, veröffentlicht wurde[19]. Die beiden anschließenden Iterationen der Prozessorfamilie, LEON2 und LEON3 wurden ebenfalls unter Open-Source Lizenzen verfügbar gestellt[19]. Deren Nachfolger, LEON4, ist jedoch nicht mehr unter einer freien Lizenz verfügbar[19].

Da die Designs der LEON, LEON2 und LEON3 Prozessoren als frei verfügbare VHDL-Designs[19] zur Verfügung stehen, eignen sich diese gut zur Verwendung in angepassten FPGAs oder ASICs[19].

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 [20].

In der „Invade" Phase werden zunächst Ressourcen für das laufende Programm reserviert. Welche Ressourcen genau reserviert werden, werden durch sogenannte „Constraints" bestimmt[20]. Anschließend wird in der „Infect" Phase die Funktion auf den reservierten Prozessorelementen ausgeführt[20]. Werden die reservierten Ressourcen nicht mehr benötigt, so kann man diese in der „Retreat' Phase wieder freigeben[20].

Das Konzept des invasiven Computings wurde von einer Kollaboration von Wissenschaftlern der Friedrich-Alexander-Universität Erlangen-Nürnberg, dem Karlsruher Institut für Technologie und der Technischen Universität München entwickelt[21]. Im Rahmen des „Transregional Collaborative Research Center Invasive Computing"[21] wird die Entwicklung dieses Programmiermodells vorangetrieben. Fördermittel erhält diese von der Deutschen Forschungsgemeinschaft[21].

Für den praktischen Einsatz des invasiven Computings wurde ein Betriebssystem namens IRTSS entwickelt. Dieses ist generell für den Einsatz auf FPGAs mit LEON Architektur konzipiert, bietet jedoch auch Versionen für die x86-Architektur. Derzeit ist es möglich, in den Programmiersprachen C, C++ und X10 Programme für den

Einsatz mit IRTSS zu schreiben.

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 allerdings nicht trivial auf andere Architekturen zu portieren. Um diese Hürde zu umgehen bietet Rust 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[22]:

```
{
    "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. Außerdem 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 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 mit der Anweisung „#![no_std]“ am Anfang des Programms kenntlich machen, dass das Programm ohne die Standardbibliothek kompiliert werden soll. 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 nach einer unbehandelten Ausnahme kontrolliert abstürzt. Der Anfang eines zu kompilierenden Programms muss also beispielsweise wie folgt aussehen:

```
#![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 build“ Befehl für Cargo Projekte. Beiden Befehlen muss dann wie auch beim Kompilieren von libcore die Spezifikations-JSON-Datei als Argument für die „--target“-Option übergeben werden.

3.2 Erstellung des octorust Hilfsprogramms

Um das relativ komplizierte und fehleranfällige Kompilieren für Rust-Programme auf der SPARC-V8 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 Pythons Standardbibliothek viele nützliche Funktionen zur Manipulation von Dateien bietet, welches sich für diesen Zweck als hilfreich erwiesen hat. Außerdem bietet Python mit „argparse“ ein praktisches Modul zum Verarbeiten von Kommandozeilen-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 Verzeichnis namens .octorust im Heimverzeichnis 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 unter anderem erstellte 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-Quelldateien und Rust-Quelldateien, 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-Projekten muss das „crate-type“ Attribut immer als „staticlib“ angegeben werden, damit das Projekt als statische Bibliothek kompiliert wird, 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 dieser 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 minimal 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 aufgerufen um die Ausführung des Programms zu beenden.

Diese generierte C Datei wird mithilfe eines passenden gcc Compilers als Objekt-Datei kompiliert und anschließend mit dem IRTSS Betriebssystem und der zuvor kompilierten statischen Rust-Bibliothek verlinkt. Nun sollte eine ausführbare Datei existieren, welche auf einem x86 Rechner 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 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, welche 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. Hierfür wird mithilfe des „toml“ Python Moduls der derzeitige Inhalt der Cargo.toml Datei eingelesen, der Pfad zur octolib Bibliothek in diese Daten injiziert und anschließend wird die Datei mit den neuen Daten überschrieben. Eine Kopie der ursprünglichen Daten werden vorerst noch im Hauptspeicher behalten und nachdem die Kompilierung beendet wurde, wobei hier egal ist ob diese erfolgreich war oder nicht, werden die alten Daten wieder in die Cargo.toml Datei geschrieben, um zu vermeiden dass die Cargo.toml Datei eine Konfiguration enthält, die auf lokale Gegebenheiten basieren. 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, die nicht direkt mit dem IRTSS in Verbindung stehen, aber trotzdem bei der Pogrammierung nützlich sein können

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.

Das Helper-Modul

Das helper-Modul enthält Adapter für C-Funktionen, für die es in Rust ohne Zugriff zur Standardbibliothek keine Alternative existiert. So werden beispielsweise mehrere Adapter um die „printf“-Funktion geboten, welche je eine andere Anzahl an zusätzlichen Parametern angeben, um diese Variablen ausdrucken zu können.

Beachtet werden muss bei diesen print-Funktionen, dass die überreichten Strings manuell Null-terminiert werden müssen. Um „Hello World“ zu drucken, muss man also wie folgt vorgehen:

```
print("Hello World\n\0");
```

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, spezifiziert 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 profitieren vor allem die Methoden, welche boolesche 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. Dies wird in den Methoden dieser Struktur jedoch abstrahiert, so dass man in Rust den „bool“-Typ verwenden kann.

Außerdem unterstützt die Methode „merge_constraints“ anstelle einer „constraints_t“ Struktur aus der C-Schnittstelle eine in Rust definierte „Constraints“ Struktur. Möchte man hierbei trotzdem lieber eine „constraints_t“-Struktur verwenden, so kann stattdessen die „merge_constraints_t“ Methode verwendet werden.

Um Zugriff auf die interne „constraints_t“ Struktur zu erhalten, kann die „to_constraints_t“ Methode aufgerufen werden. Hiernach ist die Constraints-Struktur selbst nicht mehr verwendbar, da die interne „constraints_t“-Variable nicht mehr im Besitz des zugehörigen Speichers ist.

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[23] bestehen, ist es nicht möglich die Closure direkt zu einer normalen Rust Funktion zu konvertieren. Stattdessen erstellt man einen Zeiger auf die Datenregion, welche die Closure repräsentiert und übergibt diesen dann einer „extern "C"“ Funktion, welche aus diesem Zeiger die Closure zurückgewinnt. Diese „extern "C"“-Funktion kann dann der C-Schnittstelle zusammen mit dem Zeiger auf die Closure-Daten übergeben werden.

Die Erstellung des Closure-Zeigers wird direkt in der später genauer erläuterten „infect“-Methode der „AgentClaim“-Struktur erledigt, während 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. Es wird also bereits im Konstruktor implizit die „Invade“-Phase des invasiven Computing ausgeführt.

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

Die „reinvade“-Methode elaubt es, eine „Reinvade“-Operation auf dem Claim auszuführen. Diese erlaubt es, die von der internen „agentclaim_t“-Struktur verwendeten Constraints zu aktualisieren. Diese Methode nimmt einen optionalen Parameter an, mit dem man komplett neue Constraints setzen kann. Werden neue Constraints gesetzt, werden die alten Constraints mit der „agent_constr_delete“-Funktion aus dem Speicher gelöscht.

Die wohl wichtigste Methode der „AgentClaim“-Struktur ist die „infect“-Methode. Mit dieser kann eine invasive „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. Diese Parameterdaten werden als ein Array von Void-Zeiger Daten übergeben. Die Anzahl der Elemente des Arrays müssen mit den vom Claim reservierten Rechenelementen übereinstimmen, ansonsten kann es zu Fehlern oder sogar zum Absturz des Programms kommen.

In der „infect“-Methode werden Signal-Strukturen aus der C-Schnittstelle verwendet, um die Kommunikation zwischen verschiedenen Rechenelementen zu verwalten. Um dies zu vereinfachen, wird die übergebene Funktion oder Closure nochmals von einer Closure umgeben, welche eine Referenz auf eine in der „infect“-Methode erstellten Signal-Struktur besitzt und über dieses Signal signalisiert, wann diese Closure die Ausführung ihrer Aufgabe beendet hat. Hierdurch ist es möglich, nach der Initialisierung der Ilets auf die vollständige Abarbeitung dieser zu warten oder alternativ das Signal, mit dem die einzelnen Ilets kommunizieren, als Rückgabeparameter zu verwenden. Um beide dieser Varianten zu bieten, gibt es zusätzlich zur „infect“-Methode, welche auf die vollständige Ausführung der Ilets wartet, die „infect_async“-Methode,

welche die „simple_signal“-Struktur, mit der die Ilets das Ende ihrer Ausführung signalisieren, als Rückgabewert verwendet. Anschließend kann der Nutzer mit der „simple_signal_wait“-Funktion aus der C-Schnittstelle auf die vollständige Ausführung aller Ilets warten. Die interne Closure, welche die Kommunikation über Signale implementiert, wird anschließend zu einem Void-Zeiger konvertiert, welche den Ilets dann als Datenparameter übergeben wird.

Für jedes Rechenelement wird iteriert und ein Ilet initialisiert. Diese Ilets erhalten die „execute_closure“-Funktion als ihren Funktionsparameter und den Closure-Zeiger als ihren ersten Datenparameter. Wurden der „infect“-Methode zusätzliche Datenparameter übergeben, so werden diese als zweite Datenparameter den Ilets übergeben. Nachdem ein Ilet initialisiert wurde, wird die eigentliche „Infect“-Operation durchgeführt.

Das bereits in Kapitel 2.1.2 erwähnte Drop-Trait für Rust-Strukturen wird für die „AgentClaim“-Struktur implementiert. Verlässt eine Instanz der „AgentClaim“-Struktur den Geltungsbereich, wird eine „Retreat“-Operation auf der internen „agentclaim_t“-Struktur ausgeführt und zusätzlich die Constraints 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
Hauptspeicher	8GB
Betriebssystem	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 31183335a89917f489046da746c5181174a7bdb3

Abbildung 4.1: Dies ist die Hardware/Software Konfiguration des Rechners, auf der die nachfolgenden Programme ausgeführt wurden.

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

Anhand der Werte in Tabelle 4.2 kann man prinzipiell erkennen, dass die Anlaufzeiten von C und Rust sich sehr nahe sind, während X10 hierfür merklich 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 4.3: 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 4.4: 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, sind die Werte bei optimiertem Rust nicht signifikant schlechter, 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 kann dies allerdings auch auf Kosten der Laufzeiteffizienz und Verfügbarkeit durch Garbage Collector Pausen geschehen.

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

Die erstellten Objekte sind in diesem Fall Arrays von Integer-Zahlenwerten. Das Äquivalent zu Arrays in X10 sind Rails, daher wurden diese in diesem Benchmark verwendet.

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

Abbildung 4.5: 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. So kann es dann nicht aus Versehen zu Speicherlecks kommen.

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 auf jeden Fall unerwünscht ist. Versucht man dies in Rust, so bricht das Programm sofort ab, indem die „panic_fmt“-Funktion aufgerufen wird und 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 mit einem Aufruf der „panic_fmt“-Funktion die Ausführung beendet.

4.2.3 Nicht Initialisierte Variablen

In C ist es möglich, uninitialiserte Variablen zu verwenden, dies führt allerdings ebenfalls 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.

5 Fazit und Ausblick

Im folgenden werden die Ergebnisse der Evaluation bewertet und ein Ausblick auf die Zukunft der Programmiersprache Rust im Bezug zum invasiven Computing geboten.

5.1 Fazit

Betrachtet man die Ergebnisse der Evaluation, so erkennt man, dass Rust in gewissen Aspekten C oder X10 vorzuziehen ist.

Zum einen eliminiert Rust einige Fehlerquellen, welche in C zu undefiniertem Verhalten führen können. In Rust muss der Programmierer nicht selbst auf solche Fehler achten und wird teils bereits vom Compiler auf Fehler hingewiesen. Dies erlaubt es, sicherere und weniger fehleranfällige Programme zu schreiben.

Des Weiteren weist Rust meist ein besseres Laufzeitverhalten auf als X10 es tut, vor allem in Situationen in denen der Garbage-Collector notwendig wird. Ein Garbage Collector ist zudem für Situationen, in denen Pausen nicht erwünscht sind, beispielsweise bei Echtzeitsystemen, ein Problem, denn diese verletzt die Verfügbarkeitsbedingung des Systems. Außerdem weist das Kompilieren mit octorust weitaus kürzere Kompilierungsdauern als der x10i-Compiler auf, welches bei der Entwicklung von Programmen unter Umständen ein nicht insignifikantes Zeitersparnis mit sich bringen kann. Zum Gebrauch auf Geräten mit sehr limitiertem Speicherplatz wäre Rust ebenfalls X10 vorzuziehen, da kompilierte Rust-Programme weniger Speicherplatz in Anspruch nehmen. Allerdings führt C in dieser Hinsicht weiterhin.

Negativ zu betrachten ist die starke Abhängigkeit von der C-Schnittstelle, welche die Sicherheit von Rust teils aufgibt. Die Funktionen der C-Schnittstelle bieten nämlich keine Garantien über die Sicherheit und die häufige Nutzung von Void-Zeigern zum Übertragen von Daten ist ebenfalls ein Problem, welches zu Fehlern führen kann.

5.2 Ausblick

Da Rust noch eine relativ junge Programmiersprache ist, existieren noch viele Bibliotheken für diese nicht. Zwar können mithilfe der Foreign Function Interface Bibliotheken, die in anderen Sprachen geschrieben worden sind, eingebunden werden, diese bieten jedoch nicht die Sicherheitsgarantien wie Rust es tut. In dieser Hinsicht sollte sich die Lage jedoch in der Zukunft verbessern, wenn Rust weiterhin gerne von Entwicklern genutzt wird und eventuell großflächiger zum Einsatz kommt.

Eine weitere Möglichkeit in der Zukunft ist die Portierung der Standarbibliothek auf die SPARC-V8 Architektur. Die Standardbibliothek bietet einige hilfreiche Konstrukte und Funktionen, die das Programmieren erleichtern und sicherer machen. Möglich wäre es, dass die Entwickler der Rust Programmiersprache selbst die SPARC-V8 Architektur in der Zukunft unterstützen und so die Standarbibliothek portiert wird, alternativ wäre es möglich dass die Standarbibliothek im Rahmen des invasiven Computing portiert wird, sollte genug Interesse daran bestehen.

Um die starke Abhängigkeit von IRTSS' C-Schittstelle zu verringern, können weitere Abstraktionen erstellt werden, welche von den besonderen Eigenschaften der Rust-Programmiersprache Gebrauch machen. Dass dies möglich ist, hat die Implementierung des X10-Compilers x10i bewiesen, denn dieser spricht die exakt selbe C-Schnittstelle an, bietet jedoch robuste Abstraktionen über diese und erlaubt so das Entwickeln von X10-Programmen ohne dass der Programmierer sich mit dieser befassen muss.

Literaturverzeichnis

[1] wikipedia.org, "Mooresches Gesetz," 2017.

[2] Nvidia, "GeForce GTX 1080 Specifications," 2017.

[3] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," in *Multi-processor System-on-Chip – Hardware Design and Tool Integration* (M. Hübner and J. Becker, eds.), pp. 241–268, Springer, Berlin, Heidelberg, 2011.

[4] stackoverflow.com, "Developer survey results 2016," 2016.

[5] wikipedia.org, "Servo (Software)," 2017.

[6] wikipedia.org, "Rust (Programmiersprache)," 2017.

[7] debian.org, "Rust programs versus C gcc," 2017.

[8] debian.org, "Rust programs versus Java," 2017.

[9] debian.org, "Rust programs versus Python 3," 2017.

[10] rust lang.org, "What Is Ownership?," 2017.

[11] rust lang.org, "Ownership," 2017.

[12] A. Beingessner, "The pain of real linear types in rust," 2017.

[13] rust lang.org, "Drop," 2017.

[14] rust lang.org, "Types," 2017.

[15] wikipedia.org, "Substructural type system," 2017.

[16] wikipedia.org, "Rust (programming language)," 2017.

[17] wikipedia.org, “Sun Microsystems,” 2017.

[18] wikipedia.org, “SPARC-Architektur,” 2017.

[19] wikipedia.org, “Leon,” 2017.

[20] T. C. R. C. I. Computing, “Overview of invasive Computation,” 2017.

[21] T. C. R. C. I. Computing, “Welcome to the pages of the TCRC 89 İnvasive Computing"(InvasIC),” 2017.

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

[23] 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 Janine und Heinz als auch meinem Bruder Michael, die mich meine gesamtes Leben lang durch dick und dünn begleitet und unterstützt haben. 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 mein Studium am KIT nicht halb so schön wäre. Ich danke meiner „Ersatzfamilie“, der Familie Eherler, die immer einen Platz in ihrer Mitte für mich hat. Ich danke meinem Betreuer Andreas Zwinkau, welcher mich freundlich und hilfreich durch die Erstellung dieser Arbeit begleitet hat. Und zu guter Letzt danke ich dem Karlsruher Institut für Technologie, welches es mir erst ermöglichte, dieses Studium zu absolvieren.