Diplomarbeit

L4/Valgrind

Aaron Pohle

2009

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter: Björn Döbel, Michael Roitzsch

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten:	**Aaron Pohle**
Studiengang:	Informatik
Immatrikulationsnummer:	3013906

Thema: **Dynamic Binary Analysis on L4**

Zielstellung:

Virtuelle Maschinen ermöglichen die Ausführung von Betriebssystem-Diensten auf virtueller Hardware. Dies eröffnet neue Möglichkeiten für System-Analyse und Debugging, indem der Virtualisierungs-Layer zu diesem Zweck instrumentiert wird. Im Rahmen der Diplomarbeit gilt es zu untersuchen, inwiefern sich existierende Frameworks zur dynamischen binären Analyse dazu eignen, solche Instrumentierung vorzunehmen. Hierzu soll das DBA-Framework Valgrind auf den L4.Fiasco-Mikrokern und das L4 Environment (L4Env) portiert werden, so dass schließlich beliebige L4-Anwendungen in Valgrind ausgeführt werden können.

Hierzu sind folgende Schritte durchzuführen:

- Analyse von Valgrind hinsichtlich Anforderungen an die Laufzeitumgebung
- Analyse der durch das L4Env bereitgestellten Mittel, um damit Valgrind-Mechanismen (bspw. Shadow Values, Threading, usw.) umzusetzen
- Implementierung der L4Env-basierten Valgrind-Version unter möglichst weitgehender Wiederverwendung existierenden Quellcodes
- Performance-Vergleich und -Analyse der L4-spezifischen Mechanismen im Vergleich zur existierenden Linux-Implementierung

Sollte der Rahmen der Diplomarbeit dies zulassen, ist darüber hinaus die Ausführung eines para-virtualisierten Betriebssystems (L4Linux) in L4/Valgrind zu untersuchen.

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Döbel, Dipl.-Inf. Roitzsch
Institut:	Systemarchitektur, Betriebssysteme
Beginn:	01. 10. 2008
Einzureichen:	31. 03. 2009

i.V.

Dresden, 15. 9. 2008 *Unterschrift des verantwortlichen Hochschullehrers*

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 28. März 2009

Aaron Pohle

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Motivation

> „The computer programmer is a creator of universes for which he alone is responsible. Universes of virtually unlimited complexity can be created in the form of computer programs.“ (Joseph Weizenbaum)

Bereits seit dem ersten Computer-Programm von Ada Lovelace aus dem Jahr 1842 nimmt die Komplexität von Software zu. Die damit einhergehenden Fehler sind ein impliziertes Folgeproblem, das gleichermaßen qualitativ als auch quantitativ mit dem Grad der Entwicklung wächst. Daher sind automatisierte Fehlersuche sowie Testläufe unabdingbare Voraussetzung, um einerseits hochkomplizierte Prozesse zu prüfen bzw. zu korrigieren und andererseits die Programmierkosten wesentlich zu senken.

Systemnahe Programmiersprachen wie C oder C++ verfügen naturbedingt über keinen Schutz vor typischen Programmierfehlern [19]. Dies sind beispielsweise Fehler der Speicherverwaltung – also Stack-Überläufe, Memory-Leaks oder die Benutzung von nicht alloziertem Speicher. In Hochsprachen wie Java oder Python können solche Fehler konstruktionsbedingt nicht auftreten: Es ist unmöglich, Zeiger zu benutzen oder selbst Speicher zu verwalten. Dies führt zu erheblichen Einschränkungen für das Implementieren von Betriebssystemen, Treibern oder anderer systemnaher Software. Dieser Sachzwang erfordert den Einsatz systemnaher Programmiersprachen in Verbindung mit entsprechenden Analyse-Tools für automatische Tests sowie das Validieren von C- und C++- Programmen. Generell kann zwischen zwei möglichen Analyseverfahren unterschieden werden:

1. Analyse vor der Ausführung, die die Prüfung von Semantik, Syntax, Speicherallokationen, logischer Ausdrücke, Schnittstellen und Sicherheitsaspekte [13] ermöglicht.

2. Analyse während der Ausführung; diese ermöglicht tiefe Einblicke in das ausgeführte Programm zum Debugging und Testen. Durch Instrumentierung zur Laufzeit können Programme auf Fehler jeglicher Art geprüft werden.

Mikrokerne der zweiten Generation wurden mit dem Ziel entworfen, Geschwindigkeit und Minimalismus zu vereinen sowie gleichzeitig Modularität, Flexibilität

und Sicherheit zu verbessern. Fiasco ist ein solcher Mikrokern; er wurde im Rahmen des Dresden Realtime Operating System (DROPS) an der Technischen Universität Dresden entwickelt. DROPS ist ein mikrokern-basiertes Betriebssystem, das aus dem L4-Mikrokern Fiasco und mehreren Servern – welche Dienste wie Dateisysteme, Netzwerk oder Grafik zur Verfügung stellen – sowie den Nutzerprogrammen besteht.

Ziel dieser Arbeit ist die Portierung von Valgrind auf das mikrokern-basierte Betriebssystem DROPS. Valgrind ist ein Framework zur dynamischen binären Analyse von Programmen. Mit Valgrind können Tools zum automatischen Testen und Validieren von Programmen entwickelt werden. Durch die Portierung kann Valgrind für die Entwicklung robuster, fehlerfreier und stabiler Programme für L4 genutzt werden.

Gliederung

Im ersten Kapitel „Einführung“ meiner Arbeit diskutiere ich unterschiedliche Analyse-Arten und stelle Werkzeuge zur automatischen Analyse sowie die Zielumgebung meiner Portierung vor. Im zweiten Kapitel beschäftige ich mich mit Valgrind, einem Werkzeug, das der automatischen Fehlersuche während der Laufzeit dient. Im dritten Kapitel „Die Portierung“ erläutere ich notwendige Änderungen und getroffene Entscheidungen bei der Portierung von Valgrind auf L4. Die Qualität und Geschwindigkeit meiner Portierung bewerte ich in Kapitel „Auswertung“. Abschließend gebe ich einen kurzen Ausblick und fasse die Ergebnisse meiner Arbeit zusammen.

Danksagung

An dieser Stelle danke ich Prof. Dr. Hermann Härtig für die Möglichkeit, in der Betriebssystemgruppe arbeiten zu dürfen. Mein besonderer Dank gilt auch meinen Betreuern Michael Roitzsch und Björn Döbel. Sie haben mich in allen Phasen meiner Arbeit sehr engagiert mit fachlicher Kompetenz und Kritik betreut. Meinen Eltern danke ich an dieser Stelle dafür, dass Sie mir dieses Studium ermöglichten. Meiner Freundin Michaela danke ich für die Geduld und Hilfe während der Zeit. Dank an die geduldigen Korrekturleser Lucas, Borchert und Martin für ihre Zeit und Mühe. Ebenfalls danke ich Maria, Michael, Steffen, Julian, Dirk und Joe für die angenehme Arbeitsatmosphäre und die konstruktiven Gespräche im Studentenlabor.

1 Einführung

Im ersten Teil möchte ich verschiedene Werkzeuge zum automatischen Testen und Debuggen von Programmen vorstellen. Dazu sollen zuerst die verschiedenen Analyse-Arten und deren Unterschiede erörtert werden. Daran anschließend stelle ich die Ziel-Plattform meiner Portierung – L4 – vor.

1.1 Die Analyse

Wann erfolgt die Analyse?

Analysen können vor oder während der Ausführung des zu testenden Programmes erfolgen. Dabei wird die Analyse vor der Ausführung als **statisch** und während der Laufzeit als **dynamisch** bezeichnet [20]. (In dieser Arbeit bezeichne ich Programme, die in Valgrind zur Analyse ausgeführt werden als Gäste.)

Die **statische** Analyse erfolgt vor der Laufzeit des Programmes, d.h. das Gast-Programm wird nicht ausgeführt. Als Folge dessen gibt es keine Testdaten bzw. Testfälle. Die statische Analyse kann daher auch als formale Prüfung des Programmes bezeichnet werden. Statische Tester prüfen zum Beispiel auf Korrektheit, suchen Optimierungsmöglichkeiten, visualisieren den Code oder suchen nach typischen Programmierfehlern. Vorteil dieser Analyse ist, dass alle Ausführungspfade des Gastes analysiert und geprüft werden können.

Im Gegensatz dazu steht die **dynamische** Analyse. Bei dieser wird das Gast-Programm, während es ausgeführt wird, untersucht. Zur Untersuchung ist Analysecode notwendig, der, ohne die Programm-Ausführung des Gastes zu beeinflussen, zusätzlich ausgeführt wird. Ziel des Analysecode ist allgemein das Sammeln von Daten, Messen der Geschwindigkeit und das Suchen von Fehlern. Dabei wird das Generieren und Hinzufügen des Analysecode während der Laufzeit als Instrumentierung bezeichnet. Der Vorteil dieser Analyse ist, dass sie während der Laufzeit stattfindet. Es wird also das Verhalten des Gastes unter realen Bedingungen getestet. Resultierend daraus gibt es Testdaten und Testfälle. Von Nachteil ist, dass immer nur ein Ausführungs-Pfad des Gastes analysiert werden kann.

Welcher Code wird analysiert?

Je nach Ziel und Vorlage kann **Quellcode** oder **Maschinencode** analysiert werden [20].

Bei der **Quellcode**-Analyse können High-Level Informationen, wie z.B. Variablen, Funktionen, Ausdrücke usw. berücksichtigt werden. Beachtet werden sollte, dass die Analyse von Quellcode immer abhängig von der Programmiersprache ist. Vorteil bei der Analyse von Quellcode ist die Unabhängigkeit von der Plattform (Architektur und Betriebssystem).

Die **Maschinencode**-Analyse hingegen ist sprachunabhängig, benötigt keinen Quellcode, kann dadurch aber auch nur Low-Level (systemnahe) Informationen, wie zum Beispiel Speicher, Register, Sprünge berücksichtigen. Oft wird bei der Analyse von Maschinencode dieser nicht direkt, sondern eine Zwischensprache (Bytecode) analysiert. Vorteil der Maschinencode-Analyse ist, dass auch Binaries und Bibliotheken fremder Anbieter (ohne Vorlage von Quellcode) analysiert werden können. Demgegenüber wirkt sich die Abhängigkeit von der Plattform nachteilig aus.

Arten der Analyse

In den vorangegangen Abschnitten beschrieb ich, dass die Analyse statisch oder dynamisch stattfinden und dabei Quell- oder Maschinencode untersucht werden kann. Zusammen ergeben sich vier Arten der Analyse:

- Statische Quellcode-Analyse
- Statische Maschinencode-Analyse
- Dynamische Quellcode-Analyse
- Dynamische Maschinencode-Analyse

In meiner Arbeit steht die **Dynamische Maschinencode-Analyse** (DBA – *Dynamic Binary Analysis*) im Mittelpunkt. Dazu möchte ich im Folgenden die Instrumentierung von Code – eine Grundlage von dynamischen Analysen – diskutieren. Beim Instrumentieren wird dem zu untersuchenden Code zusätzlicher Analysecode hinzugefügt. Der zu untersuchende Code kann dabei Quellcode oder Maschinencode sein. In meiner Arbeit will ich mich auf die Instrumentierung von Maschinencode beschränken.

Wie erfolgt die Instrumentierung?

Beim Instrumentieren werden dem Code des Gastes zusätzliche Anweisungen hinzugefügt, welche die eigentliche Analyse durchführen bzw. Daten sammeln. Die einfachste Variante ist die direkte Instrumentierung des Maschinencodes. Dabei verlängert sich die Ausführungszeit des Gastes nur unwesentlich. Eine aufwändigere Variante ist die Übersetzung des Maschinencodes in eine Zwischensprache („intermediate representation“ IR), die dann instrumentiert wird. Bei der Zwischensprache handelt es sich im Allgemeinen um RISC-Bytecode. Dieser lässt sich wegen des einfachen Aufbaus leicht instrumentieren. Bei einer solchen Analyse des Gastes ergeben sich damit mehr Möglichkeiten als bei der direkten Instrumentierung des Maschinencodes. In den nächsten Absätzen erläutere ich die direkte und indirekte Instrumentierung von Maschinencode.

Bei **Copy-and-Annotate** wird der Maschinencode direkt instrumentiert und danach ausgeführt (siehe Abbildung 1.1). Dazu wird im ersten Schritt jede Instruktion annotiert (kommentiert). Beim Annotieren werden die Effekte der Instruktion beschrieben. Im zweiten Schritt nutzen Tools diese Beschreibungen, um Analysecode zu generieren. Dieser wird im dritten Schritt in den originalen Maschinencode eingefügt [21]. Vorteil dieser Methode ist, dass der Gast-Code mit sehr wenigen Änderungen direkt ausgeführt werden kann.

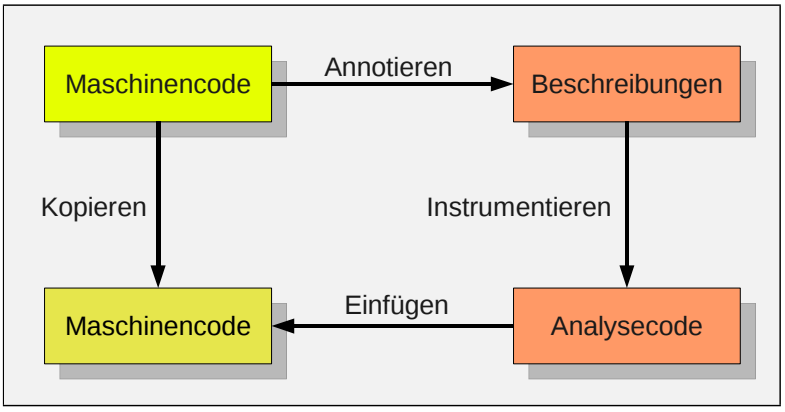

Abbildung 1.1: Copy & Annotate

Im Gegensatz dazu wird bei **Disassemble & Resynthesize** [21] im ersten Schritt der Maschinencode in eine Zwischensprache übersetzt. Bei der Zwischensprache handelt es sich in der Regel um Bytecode. Bei dieser Übersetzung werden aus jeder Maschinencode-Instruktion ein oder mehrere Instruktionen der IR. Im zweiten Schritt wird die IR instrumentiert, also Analysecode hinzugefügt. Im drit-

ten Schritt wird der IR wieder zurück in Maschinencode übersetzt. Abbildung 1.2 zeigt die Struktur von Disassemble & Resynthesize, instrumentiert wird nur die Zwischensprache des Maschinencodes.

Verglichen mit Copy-and-Annotate ist das Design und die Implementation von Disassemble & Resynthesize komplexer und aufwändiger. Von Vorteil ist Disassemble & Resynthesize, wenn komplexer Analysecode hinzugefügt werden soll. Da der Analysecode und der Code des Gastes den gleichen IR nutzen, ist sichergestellt, dass der Analysecode genauso ausdrucksstark ist wie der Code des Gastes.

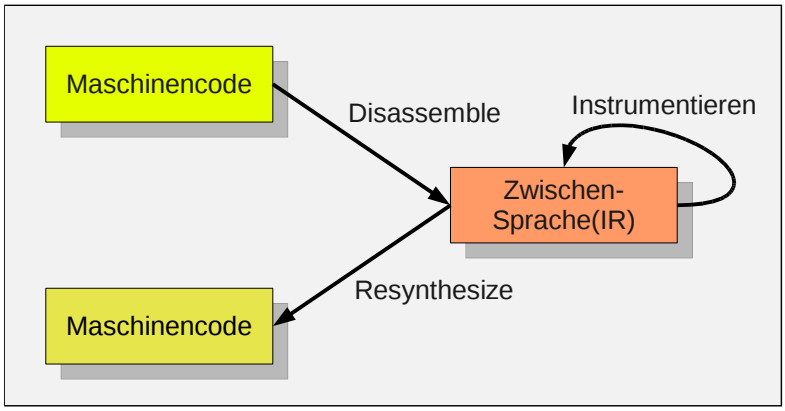

Abbildung 1.2: Disassemble & Resynthesize

Statische und Dynamische Instrumentierung von Maschinencode

Die Idee dynamischer Maschinencode-Analysen ist es, den Gast mit Analysecode zu instrumentieren. Dies kann vor oder während der Ausführung stattfinden. Je nachdem wann die Instrumentierung stattfindet, ergeben sich unterschiedliche Vor- und Nachteile bezüglich der Geschwindigkeit und des Umfangs der Instrumentierung.

Wenn die Instrumentierung vor der Laufzeit des Programmes stattfindet, wird diese als **Static Binary Instrumentation** (SBI) bezeichnet. Dabei wird vor der Ausführung des Programmes der Code des Programmes modifiziert. Da diese Modifikationen vor der Laufzeit erfolgen, ergeben sich nur sehr geringe Geschwindigkeitsverluste bei der Ausführung. Die Möglichkeiten bei der Instrumentierung sind jedoch eingeschränkt. Zusätzliche Bibliotheken oder Code, der zur Laufzeit nachgeladen oder generiert wird, kann nicht instrumentiert werden.

Instrumentierung während der Laufzeit wird als **Dynamic Binary Instrumentation** (DBI) bezeichnet. Dabei wird zur Laufzeit der Code des Gastes modifiziert. Es ist nicht notwendig, das Gast-Binary neu zu übersetzen; jeglicher Gast-Code, die verwendeten Bibliotheken und der generierte Code können instrumentiert werden. Nachteilig wirken sich bei DBI die Geschwindigkeitsverluste aus, die entstehen, weil die Instrumentierung während der Laufzeit stattfindet. Meist erfolgt die Ausführung des Gastes in DBI-Frameworks in Form von Blöcken (*Basic Blocks*).

1.2 Debugging-Werkzeuge

In den folgenden Abschnitten möchte ich einige ausgewählte Werkzeuge zum Analysieren und Testen von Programmen vorstellen. Abschließend werde ich die vorgestellten Werkzeuge vergleichen.

Pin

Ziel von Pin [14] ist es, eine Plattform zum Erstellen von Analyse-Tools für unterschiedliche Architekturen zur Verfügung zu stellen. Dazu bietet es die Möglichkeit, Gast-Programme mittels dynamisch Instrumentierung von Maschinencode zu untersuchen. Pin unterstützt die Analyse von Linux- (ARM, x86, ia64, x86/64), Windows- (x86, x86/64) oder MacOSX- (x86) Programmen.

Neben der Möglichkeit, während der Laufzeit den Code des Gastes zu instrumentieren, können mit Pin auch bereits laufende Prozesse analysiert werden. Um schnell Tools auf Basis von Pin bauen zu können, stellt Pin eine API zur Verfügung. Diese abstrahiert den darunterliegenden Befehlssatz und ermöglicht Zugriff auf Kontext- (z.B. Register-Inhalte) und Debug-Informationen.

Zu dem zu analysierenden Code kann mit Pin durch Einfügen von Funktionsaufrufen Analysecode hinzufügt werden. Einfache Analyse-Routinen fügt Pin direkt ein. Mit dem Analysecode können die Inhalte von Register und Speicher zur Laufzeit analysiert und modifiziert werden.

Die Pin-Distribution beinhaltet viele architekturunabhängige Beispiel-Tools, z.B. Profiler, Cachesimulatoren und Speicherdebugger. Mit Hilfe dieser können schnell neue Pintools entwickelt werden.

Pin steht unter der Intel Open Source License, der Quellcode ist jedoch nicht frei erhältlich. Die mitgelieferten Beispiel-Pintools sind Open Source und stehen unter einer BSD-artigen Lizenz.

DynamoRIO

DynamoRIO (RIO - run time introspection and optimisation) ist ein System zur Modifikation von Maschinencode [7], entwickelt von Hewlett-Packard und dem Massachusetts Institute of Technology (MIT) im Jahr 2001. Es ermöglicht die Instrumentierung und Optimierung von Code während der Laufzeit. Unterstützt werden Windows- (x86) und Linux- (x86) Programme.

Für die Modifikation und für das Hinzufügen von Code bietet DynamoRIO eine gut dokumentierte API. Mit Hilfe dieser kann direkt in den ursprünglichen Code der Analysecode eingefügt werden. Weiterhin bietet die API Unterstützung für den Aufruf von C-Funktionen aus dem instrumentieren Code heraus. Außerdem stellt die API Funktionalität zum Sichern und Wiederherstellen von Registerinhalten bereit.

DynamoRIO ist praktisch einsetzbar, da Programme wie Microsoft Office oder Mozilla analysiert werden können. Es können Tools zur dynamischen Optimierung, zum Profiling oder zur dynamischen Instrumentierung entwickelt werden. DynamoRIO ist unter der DynamoRIO License lizenziert und für den internen Gebrauch kostenlos nutzbar.

DIOTA

DIOTA (Dynamic Instrumentation, Optimization and Transformation of Applications) ist ein Framework zur dynamischen Instrumentierung von Maschinencode [12]. Zu DIOTA gehören viele Tools zum Debuggen von Speicher, zum Finden von Deadlocks, zum Verfolgen von Speicherzugriffen.

Mit DIOTA können Speicheroperationen instrumentiert, Systemaufruf-Parameter modifiziert oder Aufrufe dynamisch gelinkter Funktionen umgeleitet werden. Zusätzlich können am Ende jedes ausgeführten Codeblocks oder beim Zugriff auf bestimmte Speicherbereiche Funktionen aufgerufen werden. Außerdem kann DIOTA mit Threads, positionsunabhängigem Code, Signalen und selbstmodifizierendem Code umgehen. Um selbstmodifizierenden Code zu unterstützen, werden bereits übersetzte Codeblöcke durch DIOTA als schreibgeschützt markiert. Sobald schreibende Zugriffe auf die geschützten Codeblöcke stattfinden, kann DIOTA diese neu übersetzen.

Mit DIOTA können Linux- (x86) Programme wie beispielsweise die Webbrowser Mozilla und Konqueror analysiert werden. Es steht unter der GNU General Public License (GPL) und ist kostenlos erhältlich.

Dyninst

Dyninst, entwickelt an der University of Wisconsin-Madison und der University of Maryland, ermöglicht das Einfügen von Code in ein laufendes Programm [8]. Dabei ist es nicht notwendig, das Programm neu zu kompilieren, neu zu linken oder neu auszuführen. Dyninst basiert auf der Idee der dynamischen Instrumentierung von Maschinencode. Mit Dyninst können Profiling-Werkzeuge, Debugger und Anwendungen zur interaktiven Steuerung und Visualisierung einer laufenden Computersimulation entwickelt werden.

Realisiert als Bibliothek, können mit Dyninst Programme zur Laufzeit instrumentiert werden. Umgesetzt wird die Instrumentierung durch einen „Mutator-Prozess“. Mit diesem kann während Laufzeit Code eingefügt oder entfernt werden. Dyninst bietet eine API, um architekturunabhängige Analyse-Tools zu entwickeln. Mit der API können beispielsweise Argumente und Rückgabewerte bei Funktionsaufrufen analysiert oder der Funktionsaufruf ersetzt werden. Das Einfügen von Code geschieht durch das Einfügen von Sprungbefehlen in den originalen Code. Durch die Sprungbefehle wird der einzufügende Code angesprungen und ausgeführt.

Mit Dyninst können große Programme wie Microsoft Office oder MySQL untersucht werden. Dabei werden die Plattformen Linux (x86 und ia64), Windows (x86), Solaris (SPARC), AIX (PowerPC), IRIX (MIPS) und Tru64 Unix (Alpha) unterstützt. Dyninst ist frei für Forschungszwecke erhältlich, die Weitergabe bedarf jedoch der Zustimmung von Paradyn.

Purify

Purify ist ein Debugger zum Finden von Speicherzugriffs-Fehlern in C- oder C++- Programmen [9]. Ursprünglich wurde es von Reed Hastings bei Pure Software entwickelt, zur Zeit wird es bei IBM weiterentwickelt. Basierend auf dynamischer Instrumentierung von Maschinencode protokolliert und kontrolliert Purify jeden Speicherzugriff. Dadurch können Zugriffe auf nicht allozierten oder nicht initialisierten Speicher gefunden werden. Weiterhin können Memory Leaks (allozierte Speicherbereiche, auf die kein Zeiger verweist) erkannt werden. Für die Analyse mit Purify wird kein Quellcode benötigt; somit können auch Bibliotheken von Fremdanbietern getestet werden.

Mit Purify können Programme für Windows (x86,x86/64), Solaris (SPARC, x86, x86/64), HP-Unix (ia64, PA-RISC) und Linux (x86, x86/64) untersucht werden. Purify ist proprietäre Software, die bei IBM käuflich erworben werden kann [25].

Insure++

Insure++ ist ein Analyse- und Test-Werkzeug für C und C++ Programme [1]. Mit Insure++ können automatisch Fehler bei Speicherzugriffen, Fehler bei Zugriffen auf Arrays, Memory Leaks und invalide Zeiger gefunden werden. Während der Analyse überprüft Insure++ alle Typen von Speicherreferenzen, speziell Referenzen auf Stack und gemeinsam genutzten Speicher. Dabei wird aller Code des Programmes inklusive Fremd-Bibliotheken analysiert.

Mit Insure++ kann Quellcode oder Maschinencode instrumentiert werden. Dazu bietet Insure drei verschiedene Arten der Analyse:

- Vollständige Instrumentierung des Quellcodes: Dazu ist der Quellcode notwendig. Insure++ generiert daraus instrumentierten Code und übersetzt diesen.
- Statisches Linken: Insure++ wird Teil des zu untersuchenden Programms. Analysefunktionen werden direkt zum Maschinencode hinzugefügt.
- Chaperon: Nur für dynamisch gelinkte Linux-Programme. Insure++ wird in Form einer dynamisch ladbaren Bibliothek zum Maschinencode des zu untersuchenden Programms hinzugefügt.

Mit Insure++ können Windows- (x86, x86/64), Linux- (x86, x86/64), Solaris- (UltraSPARC), AIX- (PowerPC) und HP-Unix- (PA-RISC) Programme analysiert werden. Insure++ ist proprietäre Software und kann von Parasoft [5] käuflich erworben werden.

Valgrind

Valgrind ist ein Framework zum Entwickeln von Analyse-Tools. Es basiert auf der dynamischen Maschinencode-Analyse. Gast-Programme können während der Laufzeit instrumentiert und damit untersucht werden. Da Valgrind Maschinencode analysiert, ist die Untersuchung von der Programmiersprache unabhängig. Der gesamte Code des Gastes sowie die verwendeten Bibliotheken können untersucht werden. Die Instrumentierung findet zur Laufzeit statt, der Gast kann also unter realen Bedingungen getestet und analysiert werden.

Zur Instrumentierung wird der Maschinencode des Gastes in eine Zwischensprache übersetzt. Diese kann durch die Tools instrumentiert werden; dabei wird Code hinzugefügt, modifiziert oder ausgetauscht. Weiterhin unterstützt Valgrind *Shadow Values*. In diesen können für Register und Speicher Metainformationen gespeichert werden. Shadow Values werden beispielsweise genutzt, um fehlerhafte Speicherzugriffe zu erkennen. Dazu vermerkt Valgrind in den Shadow Values

für jedes einzelne genutzte Byte Speicher, ob dieses adressierbar und gültig ist. Nach dem Instrumentieren der Zwischensprache wird diese in Maschinencode übersetzt und ausgeführt.

Mit Memcheck – dem ersten Valgrind-Tool – können Fehler wie beispielsweise Puffer-Überläufe, Memory-Leaks oder Lese- und Schreibzugriffe auf nicht allozierten Speicher erkannt werden.

Valgrind ist zur Zeit für Linux (x86, x86/64, PPC32/64) und MacOS (x86, x86/64) verfügbar. Lizenziert ist Valgrind unter der GNU General Public License und frei erhältlich.

Vergleich

Ziel meiner Arbeit ist die Untersuchung und Portierung eines Werkzeuges zur dynamischen Maschinencode-Analyse von Programmen. Voraussetzung für eine Portierung ist die Verfügbarkeit des Quellcodes und eine freie Lizenz des zu portierenden Programmes. Frameworks zum Erstellen von Analyse-Werkzeugen haben im Unterschied zu einzelnen monolitischen Analyse-Werkzeugen den Vorteil, dass weitere Werkzeuge entwickelt werden können (die auf den Frameworks basieren).

In Tabelle 1.1 habe ich die in den vorangegangenen Abschnitten vorgestellten Tools gegenübergestellt. Nur von Valgrind, DIOTA und Dyninst ist der Quelltext frei verfügbar. Dyninst ist für sehr viele Plattformen verfügbar, steht jedoch unter keiner freien Lizenz. Valgrind ist, verglichen mit DIOTA, für mehr Plattformen verfügbar und hat damit die besseren Voraussetzungen für eine Portierung.

Mit auf Valgrind basierenden Analyse-Tools können Programme während der Laufzeit untersucht werden. Neben der Möglichkeit, neue Tools auf Basis von Valgrind zu entwickeln, bringt Valgrind bereits Standard-Tools mit. Diese eignen sich zur Fehlersuche und zum Profiling von Anwendungen. Da Valgrind frei verfügbar ist und gute Voraussetzungen für automatisches Testen und Debugging bietet, entschied ich mich, Valgrind auf L4 zu portieren.

1.3 L4: Mikrokern und Multi-Server-OS

L4 ist eine Familie von Mikrokernen der zweiten Generation, ursprünglich entwickelt und implementiert von Jochen Liedtke [17]. Liedtkes Implementierung war vor allem auf Leistung optimiert und zu diesem Zweck direkt in Assembler geschrieben. Bei anderen Implementationen der L4 ABI (Applikation Binary Interface), zum Beispiel bei L4Ka::Pistachio (Universität Karlsruhe), L4/MIPS (University of New South Wales) und Fiasco (Technische Universität Dresden)

Name	Art	Plattform Host	Plattform OS	Ausführung [20]	Quelltext verfügbar
Pin	F	x86,ia64 x86/64,Xscale	Linux, Windows MacOS	C	nein
DynamoRIO	F	x86	Linux, Windows	C, O	nein
DIOTA	F	x86	Linux	C, O	ja
Dyninst	F	x86, ia64 SPARC, PPC MIPS, Alpha	Linux, Windows Solaris, AIX IRIX, Tru64 UX	N	ja
Purify	M	x86, x86/64 SPARC, ia64 PA-RISC	Linux, Windows, Solaris, HP-UNIX	N	nein
Insure++	M	x86, x86/64 UltraSPARC PPC, PA-RISC	Linux, Windows, Solaris, AIX, HP-Unix	N, C	nein
Valgrind	F	x86,x86/64 PPC32/64	Linux, AIX MacOSX, FreeBDS	C	ja

Art:
F Framework
M monolitisches Analyse-Werkzeug

Ausführung:
C dynamische Übersetzung von Maschinencode mit Caching
O dynamische Optimierung von Maschinencode
N normale Ausführung, mit Sprüngen in Analysecode

Tabelle 1.1: Vergleich der vorgestellten Analyse-Werkzeuge

liegt der Fokus auf Sicherheit, Isolation und Robustheit. Mikrokerne zweiter Generation sind nach folgendem Prinzip von Jochen Liedtke entworfen:

> „. . . a concept is tolerated inside the micro-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.“ [17]

Das heißt, dass nur die primitivsten Mechanismen und Abstraktionen im Kern implementiert werden. Unabdingbare Abstraktionen im Kern sind Threads und Adressräume. Notwendige Mechanismen sind Kommunikation, Scheduling und Mappings.

Programme werden, wie bei monolithischen Kernen, im Nutzermodus ausgeführt. Während bei monolithischen Systemen die meisten Ressourcen und Dienste (z.B. Speicher, Dateien, Netzwerk) direkt vom Kern verwaltet werden, findet in mikrokern-basierten Systemen die Verwaltung von Ressourcen im Nutzermodus statt.

Die Verwaltung von Ressourcen auf mikrokern-basierten Systemen kann auf zwei verschiedene Arten erfolgen: Entweder erfolgt die Realisierung als monolithisches Programm, dabei wird es als *Single-Server-OS* bezeichnet. Alle Ressourcen des Systems werden von einem Server zur Verfügung gestellt. Demgegenüber wird die Verwaltung von Ressourcen durch ein verteiltes System als *Multi-Server-OS* bezeichnet. Dabei bieten verschiedene Server Dienste an oder stellen Ressourcen zur Verfügung.

1.4 L4Env

Das *L4Env* (*L4 Environment*) ist eine Umgebung zur Entwicklung von Programmen, die auf L4-Mikrokernen basieren [23]. Es wurde als Teil des Dresden Real-Time Operating System (DROPS) entwickelt. Die Idee des L4Env ist die Bereitstellung einer minimalen Umgebung zur Entwicklung von L4-basierten Programmen. In Abbildung 1.3 sind der Aufbau und die Komponenten des L4Env dargestellt.

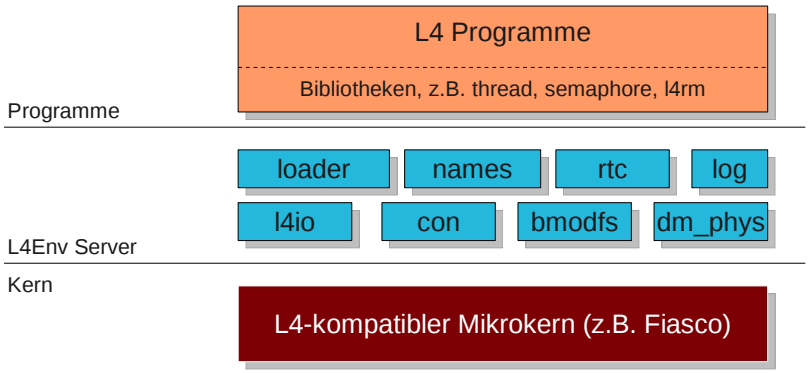

Abbildung 1.3: Aufbau des L4Env

Dazu stellt das L4Env Server und Bibliotheken zur Verfügung. Die Server dienen zur Verwaltung von System-Ressourcen, wie Speicher, Tasks und I/O. Die Bibliotheken ermöglichen den Zugriff auf die von den Servern verwalteten Ressourcen und stellen darüber hinaus zusätzliche Funktionalität bereit.

In Tabelle 1.2 erläutere ich einige ausgewählte Server und Bibliotheken.

Server

- **Roottask** stellt Ressourcen wie physischen Speicher, Interrupts und Tasks bereit.
- **L4IO** ist ein einfacher I/O Manager, der den Zugriff auf Interrupts, I/O-Ports und PCI-Geräte bietet.
- **Loader + simple_ts** ermöglicht das Starten und Beenden von Programmen zur Laufzeit.
- **Con**, **Log** sind Konsolen, die zur Ein- und Ausgabe während der Laufzeit im L4Env dienen.
- **DMphys** stellt physischen Speicher zur Verfügung.
- **Names** ist ein Namensdienst, der Threads das Registrieren und Abfragen von IDs und Namen ermöglicht.
- **Simple File Server** ist ein einfacher Dateiserver des L4Env, der den Zugriff auf Dateien ermöglicht.

Bibliotheken

- **L4rm** Der L4-Region-Mapper, verwaltet den virtuellen Adressraum einer Task und ist gleichzeitig Pager.
- **Semaphore + Lock** stellt Semaphoren und Locks zur Synchronisation zur Verfügung.
- **Thread** ist eine Thread-Bibliothek, die das Starten und Beenden von Threads ermöglicht.

Tabelle 1.2: Server und Bibliotheken des L4Env

2 Valgrind

In diesem Kapitel stelle ich das Framework Valgrind vor. Dieses führt zur Analyse und zum Testen auf einer virtuellen CPU Gast-Programme aus. Während der Ausführung des Gastes in Valgrind kann das Gast-Programm untersucht werden.

Mit seinem Aufbau beginnend stelle ich wichtige Teile und Mechanismen von Valgrind vor. Ziel dieses Kapitels ist es, die Funktionsweise von Valgrind zu erläutern und damit Grundlagen zu legen, die für das Verständnis des nächsten Kapitels – die Portierung – wichtig sind.

2.1 Der Aufbau Valgrinds

Valgrind ist eine Programmier-Distribution, die zur Entwicklung von Debugging-Werkzeugen (Tools) [22] dient. Es besteht aus Tools, LibVEX und Valgrinds Core (Coregrind), letztere stellen das Valgrind-Framework dar. Um vollständige Kontrolle über die Ausführung des Gastes zu erhalten, verwendet Valgrind die Methode der dynamischen Übersetzung von Maschinencode [21]. Dieser Ansatz besitzt zwei Vorteile:

- Umfang: Jeglicher Code des Gastes sowie die genutzten Bibliotheken können ohne Vorlage des Quellcodes analysiert werden.
- Vereinfachung: Weder der Gast noch die Bibliotheken müssen neu übersetzt, neu gelinkt oder anderweitig modifiziert werden.

Aufgabe der LibVEX ist die Übersetzung des Gast-Codes. Dessen Ausführung wiederum wird durch Coregrind gesteuert. Die Instrumentierung des Gastes, das Sammeln von Laufzeitinformationen sowie deren Auswertung wird von den Tools durchgeführt. (In dieser Arbeit bezeichne ich die Ausführung eines Gastes in Valgrind als interpretierte Ausführung.)

Zur modularen Gestaltung Valgrinds dient der Valgrind-Loader. Dieser entscheidet je nach Kommandozeilen-Argument, welches Tool gestartet werden soll. Das Tool selbst besteht aus der LibVEX, Coregrind und dem eigentlichen Tool-Code.

2.2 Geschichte und aktueller Status

Valgrind wurde erstmals 2002 als monolithischer „Speicherchecker“ für C- und C++- Programme vorgestellt [21]. Durch das Entwickeln weiterer Werkzeuge auf der Basis von Valgrind wurde aus dem „Speicherchecker“ ein Framework zum Erstellen von Analyse-Tools. Valgrinds Core unterstützt systemnahe Instrumentierung von Programmen. Dazu beinhaltet es einen Just-In-Time (JIT) Compiler, eine einfache C-Bibliothek, Unterstützung für Signale und einen Scheduler für die Ausführung von Threads.

Valgrind ist freie Software, lizenziert unter der General Public Licence (GPL). Momentan läuft es auf Linux, AIX und MacOSX. Es ermöglicht die Analyse von Programmen wie Mozilla und OpenOffice.

Die folgende Liste enthält Tools, die auf dem Framework Valgrind basieren. Sie gibt einen Überblick über die Möglichkeiten der Analyse mit Valgrind:

- **Memcheck** ist ein „Speicherchecker“, zum automatisierten Suchen von Speicherverwaltungsproblemen.
- **Cachegrind** ist ein Cache-Profiler.
- **Callgrind** ist eine Erweiterung von Cachegrind, die die Darstellung von Funktionsaufrufgraphen ermöglicht.
  KCachegrind stellt die Ergebnisse von Cachegrind und Callgrind visuell dar.
- **Massif** ist ein Heap-Profiler.
- **Helgrind + DRD** dienen zum Debuggen von Threads.
- **iogrind** ist ein I/O-Profiling Tool.
- **Redux** erstellt dynamische Datenflussgraphen für die gesamte Programmausführung.
- **Annelid** ist ein *Bounds-Checker*.
- **Crocus** prüft die Benutzung von Signalen.
- **Interactive** bringt eine GDB-artige Schnittstelle zum Debugging mit.
- **VGprof** ist ein *gprof*-artiger Profiler.
- **Cachetool** generiert *Memory-Traces*.
- **Valgrind/Wine** dient zum Debugging von Windows-Programmen.

2.3 Das Valgrind-Framework LibVEX & Coregrind

Die LibVEX [4] und Valgrinds Core bilden zusammen das Valgrind-Framework. Mit dessen Hilfe können Tools zur dynamischen Maschinencode-Analyse von Programmen erstellt werden.

Die LibVEX ist ein Just-In-Time Compiler für die Architekturen x86, x86/64, ARM und PPC. Sie arbeitet mit statisch oder dynamisch gelinkten ELF-Binaries. Diese müssen weder neu kompiliert, neu gelinkt noch modifiziert werden. Der Gast-Code wird in Bytecode, genauer in *UCode*, übersetzt. Dabei handelt es sich um einen RISC-artigen Zwei-Adress-Bytecode. Jedes Register der simulierten CPU wird in den Speicher abgebildet.

Die Übersetzung findet während der Laufzeit statt. Der Gast-Code wird blockweise übersetzt und ausgeführt. Die Blöcke – Basic Blocks – sind Maschinencode-Sequenzen, die an Funktions-Einsprungpunkten beginnen und mit einem den Kontrollfluss beeinflussenden Befehl (`jump`, `call`, `return`, `int`) enden.

Die blockweise Übersetzung von Maschinencode in Bytecode und umgekehrt verläuft in 5 Phasen:

(i) *Disassemblieren:* Jede Maschinen-Instruktion wird in eine oder mehrere UCode-Instruktionen übersetzt. Dabei werden virtuelle Register benutzt.

(ii) *Optimierung:* Redundanter UCode, der durch den einfachen Disassembler entsteht, wird entfernt.

(iii) *Instrumentierung:* Das Tool fügt zusätzliche Instruktionen (Analysecode) hinzu.

(iv) *Registerallokation:* Die virtuellen Register werden auf reale Register abgebildet.

(v) *Code-Generierung:* Jede UCode-Instruktion wird in eine oder mehrere Maschinen-Instruktionen übersetzt. Manche Instruktionen werden durch Assembler- und C-Funktionsaufrufe ersetzt.

Übersetzte Blöcke werden zwischengespeichert, um so die Geschwindigkeitsverluste bei der Ausführung gering zu halten. Am Ende jedes Blockes wird der nächste Block im Cache gesucht, gegebenenfalls neu übersetzt und ausgeführt.

Um selbst modifizierenden Code zu unterstützen, wird bei der Übersetzung eines Code-Blockes zusätzlich dessen Hash-Summe gespeichert. Vor der Ausführung eines Blocks überprüft Valgrind die Hash-Summe und übersetzt den Block

gegebenenfalls neu. Zudem kann der Gast Valgrind explizit auffordern, bestimmte zwischengespeicherte Blöcke zu verwerfen und neu zu übersetzten. Dies ist notwendig, um zum Beispiel JIT-Compiler in Valgrind ausführen zu können[22].

Die LibVEX wird von OpenWorks LLP entwickelt. Sie ist ein grundsätzlicher Bestandteil von Valgrind. Zur Nutzung der LibVEX in Projekten, die unter der GPL stehen, bietet OpenWorks LLP ein duales Lizenzierungsschema. Projekte, die unter der GPL lizenziert sind, dürfen die LibVEX nutzen (diese steht dann auch unter der GPL). Für die Nutzung der LibVEX in anderen Projekten muss eine Lizenz von OpenWorks LLP erworben werden.

Coregrind ist für die Ausführung des Gastes zuständig. Dazu verfügt es über einen Scheduler, der die Ausführung der Code-Blöcke des Gastes steuert. Außerdem ist Valgrinds Core für die Behandlung von Gast-Anfragen, Systemaufrufen, für die Speicherverwaltung und das Scheduling von Threads verantwortlich. Die Tools können mit Hilfe von Valgrinds Core die Ausführung des Gastes beeinflussen.

Details zu den Komponenten von Valgrinds Core erläutere ich in den folgenden Abschnitten.

2.4 Der Start von Valgrind unter Linux

Um später die Unterschiede beim Start von Valgrind auf L4 darstellen zu können, möchte ich im Folgenden wichtige Punkte beim Start von Valgrind auf Linux vorstellen.

Das Ziel ist es, die Ausführung von Valgrinds Core, dem Tool und dem Gast in einem Adressraum vorzubereiten. Die Tools sind statisch gelinkte Binaries. Sie bestehen aus dem Code von Valgrinds Core und den eigentlichen Tool-Code. Das Starten der Tools erfolgt durch den Valgrind-Loader. Unter Linux startet der Aufruf `valgrind` den Valgrind-Loader. Dieser analysiert die Kommandozeilen-Argumente und startet das gewünschte Valgrind-Tool mit Hilfe des Systemaufrufes |execve|. Der Start des jeweiligen Tools erfolgt dabei in folgenden Schritten:

1. Valgrind wechselt auf seinen eigenen Stack.
2. Umgebung und Kommandozeilen-Argumente werden analysiert.
3. Beim Start der Speicherverwaltung wird der initiale Aufbaus des Adressraums (siehe Abschnitt 2.7) ermittelt.
4. Valgrind lädt das Gast-Binary (Daten- und Text-Segment).
5. Das Stack- und Daten-Segment des Gastes wird vorbereitet.

6. Initialisierung des Tool.
7. Initialisierung der LibVEX.
8. Initialisierung des Schedulers.
9. Laden der Debug-Informationen des Gastes.
10. Beginn der interpretierten Ausführung des Gastes.

2.5 Ressourcen-Konflikte

Bei der Ausführung von Gast und Valgrind in einem Adressraum entstehen aufgrund der gemeinsamen Nutzung von Ressourcen Probleme [20]. Beispielsweise führt die gemeinsame Nutzung von Ressourcen wie Speicher oder Threads zu Konflikten. Je nach Konflikt verwendet Valgrind unterschiedliche Methoden um ihn zu lösen:

- **Partitionierung** (*space multiplexing*) Die gemeinsam genutzte Ressource wird in separate Teile eingeteilt. Valgrind unterbricht alle speicherrelevanten Systemaufrufe des Gastes, aktualisiert seine eigene Speicherverwaltung und modifiziert die Argumente des Systemaufrufs. Dadurch wird die Partitionierung von Speicher erreicht. Andere Ressourcen, die sich durch Partitionierung gemeinsam nutzen lassen, sind zum Beispiel Datei-Deskriptoren und Speicher.

- **Zeit-Multiplexing** Die Zeit, für die eine Ressource zur Verfügung steht, wird aufgeteilt. Abwechselnd können Gast und Valgrind eine Ressource nutzen. Beispielsweise werden die Register von CPU und FPU gemeinsam benutzt. Um dies zu ermöglichen, werden die Inhalte in den Speicher gesichert, nachdem Gast oder Valgrind die Register benutzt haben. Bevor Gast oder Valgrind die Register wieder benutzen, werden die jeweiligen Inhalte aus dem Speicher wiederhergestellt.

- **Virtualisierung** Hierbei wird die Ressource teilweise oder vollständig in Software emuliert. Beispielsweise werden die „Local Descriptor Tables“ in Verbindung mit den Segment-Registern virtualisiert.

- **Sharing** Einige Ressourcen können gemeinsam genutzt werden, ohne dass dabei Konflikte entstehen. Dies sind beispielsweise Ressourcen wie die Prozess ID, die Prozess-Struktur oder das aktuelle Verzeichnis.

2.6 C-Bibliothek

Valgrind ist weder von der GNU Standard C-Bibliothek noch von anderen Bibliotheken abhängig. Fehler in der C-Bibliothek können somit die Ausführung des Gastes auf der virtuellen CPU sowie die Ausführung von Core und Tool auf der realen CPU nicht beeinflussen.

Damit Valgrind und das Tool trotzdem Funktionen aus der C-Bibliothek nutzen können, stellt Valgrind Implementationen einiger wichtiger Standard-Funktionen, wie `printf()`, `malloc()` und `open()`, zur Verfügung. Da Valgrind und Gast in einem Adressraum ausgeführt werden, beginnt jede dieser Funktionen mit einem Präfix. Dadurch soll gewährleistet werden, dass die Namen von Valgrinds Funktionen sich von denen des Gastes unterscheiden. Allgemein beginnen alle Funktionen von Valgrind und Tool mit einem Präfix, um so die Ressource „Funktionsnamen “zu partitionieren.

2.7 Speicherverwaltung in Valgrind

Da Valgrind und der Gast in einem Adressraum ausgeführt werden, ist es notwendig, diesen zu partitionieren und die Nutzung zu überwachen. Das Speicherverwaltungssystem von Valgrind verfolgt alle Änderungen im virtuellen Adressraum. Zusätzlich steuert es alle Mappings von Valgrind und Gast, um so den Aufbau des Adressraumes direkt zu beeinflussen.

Bei der Initialisierung des Speicherverwaltungssystemes wird der Aufbau des Adressraums aus der virtuellen Datei `/proc/self/maps` gelesen. Alle folgenden Mappings von Valgrind bzw. Gast werden vom Speicherverwaltungssystem gesteuert und beobachtet. Das Steuern der Mappings erfolgt durch Beobachtung und Modifikation aller speicherrelevanten Systemaufrufe des Gastes. Beispiele für solche Systemaufrufe sind `mmap()`, `munmap()`, `brk()`, `mprotect()`, `shmat()`, `shmdt()`.

2.8 Systemaufrufe des Gastes

Systemaufrufe werden auf der realen CPU durchgeführt. Dazu fängt die LibVEX Systemaufrufe des Gastes ab und ermöglicht Valgrind, die Systemaufrufe zu modifizieren oder zu simulieren. Um die korrekte Ausführung der Systemaufrufe zu gewährleisten, werden diese so ausgeführt, als wäre es der Gast selbst. Dabei muss Valgrind sicherstellen, dass es nicht die Kontrolle über die gesamte Programmausführung verliert.

Um dies zu garantieren, werden Systemaufrufe in folgenden Schritten ausgeführt:

(i) Sichern des Stackpointers von Valgrind.

(ii) Kopieren der Inhalte der virtuellen Register in die echten mit Ausnahme des Befehlszählers, damit Valgrind die Kontrolle bei der Ausführung des Systemaufrufes nicht verliert.

(iii) Ausführen des Systemaufrufes.

(iv) Zurückkopieren der Inhalte der echten Register in die virtuellen.

(v) Wiederherstellen des Stackpointers von Valgrind.

Valgrind und Tool können indirekt die Ausführung der Systemaufrufe des Gastes beeinflussen. Dazu inspizieren und verändern sie Argumente und Rückgabewerte der Systemaufrufe oder führen den Systemaufruf gar nicht aus. Realisiert wird dies mit Pre- und Post-Wrappern, die vor und nach dem Systemaufruf ausgeführt werden.

Systemaufrufe dienen dem Zugriff auf Ressourcen. Um die in Abschnitt 2.5 beschriebenen Methoden zur Behandlung von Konflikten umzusetzen, können mit den eben genannten Pre- und Post-Wrappern vor und nach dem Systemaufruf entsprechende Aktionen durchgeführt werden.

Besondere Rücksicht ist bei blockierenden Systemaufrufen nötig. Im nächsten Abschnitt erläutere ich, wie Valgrind mit den vom Gast erstellten Threads umgeht. Dabei muss Valgrind die Konsistenz seiner Daten sicherstellen.

2.9 Tasks und Threads in Valgrind

Mit Valgrind können Gast-Programme mit Threads analysiert werden. Dabei führt jeweils ein Valgrind-Thread einen Gast-Thread interpretiert aus. Außerdem verwaltet Valgrind zu jedem vom Gast genutzten Register oder Speicherbereich zusätzliche Daten - *Shadow Values*. Da die Shadow Values immer aktualisiert werden müssen, sind Modifikationen in Registern oder im Speicher nicht mehr atomar. Wenn also mehrere Threads des Gastes ausgeführt werden, muss die Kohärenz der Zugriffe auf die Shadow Values gewährleistet sein.

Dazu führt Valgrind Threads nacheinander aus. Die Serialisierung wird mit Hilfe eines Locks realisiert – nur der Thread, der das Lock besitzt, wird ausgeführt. Spätestens nach 100.000 ausgeführten Code-Blöcken wird das Lock abgegeben, damit andere Threads ausgeführt werden können. Probleme ergeben

sich bei blockierenden Systemaufrufen. Sobald ein Thread in einem Systemaufruf blockiert, kann kein anderer Thread ausgeführt werden. Der blockierende Thread besitzt das Lock und verhindert so, dass andere Threads ausgeführt werden können. Um trotzdem blockierende Systemaufrufe zu ermöglichen, wird vor deren Ausführung das Lock abgegeben. Sobald der Thread aus dem Systemaufruf zurückkehrt, muss er das Lock wieder anfordern.

Damit Valgrind nach der Beendigung des Gastes letzte Aufgaben durchführen kann, wird der Systemaufruf `exit()` durch Valgrind abgefangen. Valgrind stoppt dann die simulierte CPU, führt letzte Aktionen aus (beispielsweise das Ausgeben der Analyse-Ergebnisse) und beendet sich danach selbst mit dem Systemaufruf `exit()`.

Die serielle Ausführung der Threads führt zu Geschwindigkeitsverlusten. Insbesondere auf Multi-Prozessor-Maschinen wird dadurch der Vorteil der echten parallelen Ausführung von Threads bei Multi-Thread-Anwendungen nicht genutzt.

2.10 Funktionsersetzung, Funktionswrapping und Anfragen des Gastes

Valgrind bietet unterschiedliche Möglichkeiten, die Ausführung des Gastes zu beeinflussen. Umgekehrt kann aber auch der Gast Anfragen an Valgrind bzw. an das Tool senden. Mit Hilfe von *Funktionsersetzung* ist es möglich, Funktionen des Gastes vollständig zu ersetzen. Beispielsweise können Aufrufe kritischer Funktionen umgeleitet werden. Außerdem kann durch *Funktionswrapping* vor und nach der eigentlichen Zielfunktion Code ausgeführt werden. Dies ist sinnvoll, um Argumente und Rückgabewerte von Funktionen zu inspizieren und gegebenenfalls zu modifizieren.

In Abbildung 2.1 stelle ich ein Beispiel für den bei Funktionsersetzung und Funktionswrapping notwendigen Code vor. Dieser muss in Form einer dynamisch ladbaren Bibliothek oder durch statisches Linken zum Gast hinzugefügt werden. Das Hinzufügen des Codes hat keinen Einfluss auf die Ausführung des Gastes. Valgrinds Core erkennt beim Lesen des Gast-Binarys diesen Code und richtet entsprechende Funktionsweiterleitungen ein.

Außerdem haben auch Gast-Programme die Möglichkeit, Anfragen an Valgrind oder das Tool zu senden. Mit dem Makro `VALGRIND_NON_SIMD_CALL` können Funktionen des Gastes auf der realen statt auf der virtuellen CPU, also nicht instrumentiert, ausgeführt werden.

```
#include "valgrind.h"

int
VG_REPLACE_FUNCTION_ZU(NONE , foo) ( int arg0, char *arg1 )
{
/* do something */
}

#include "valgrind.h"
int I_WRAP_SONAME_FNNAME_ZU(NONE, bar)( int arg0, char *arg1 )
{
   int    result;

   OrigFn fn;
   VALGRIND_GET_ORIG_FN(fn);

   /* before calling wrapped function */

   CALL_FN_W_WW(result, fn, arg0, arg1);

   /* after calling wrapped function */

   return result;
}
```

Abbildung 2.1: Code für Funktionsersetzung und Funktionswrapping

2.11 Tools

Valgrind ist ein Framework zum Erstellen von Test- und Analyse-Tools. Während Valgrind den Gast ausführt, können die Tools den Code des Gastes instrumentieren und somit die Ausführung beobachten und kontrollieren.

Ein Tool muss vier Funktionen zur Initialisierung, Instrumentierung und für die Beendigung bereitstellen. Die Funktionen `<toolname>_pre_clo_init()` und `<toolname>_post_clo_init()` dienen zur Initialisierung und werden vor bzw. nach der Analyse der Kommandozeilen-Argumente ausgeführt. Bei der Initialisierung können die Tools Dienste von Valgrinds Core anfordern, Funktionen für bestimmte Ereignisse registrieren und eigene Initialisierungen durchführen. Valgrind bietet beispielsweise Dienste für Speicherung und Ausgabe von Fehlern.

Die Funktion `<toolname>_instrument()` dient der Instrumentierung des Gast-Codes. Sie wird von Valgrinds Core aufgerufen, sobald das Tool den UCode des Gastes instrumentieren soll. Beim Instrumentieren können dem Gast-Code Anweisungen oder Sprünge in C-Funktionen hinzugefügt werden. Das Tool kann

dabei jeden einzelnen Maschinenbefehl instrumentieren. Der Anfang und das Ende jedes Maschinenbefehls ist im UCode markiert.

Am Ende der Ausführung wird die Funktion `<toolname>_fini()` des Tools aufgerufen. Mit ihr können die Ergebnisse der Analyse ausgegeben werden. Abbildung 2.2 zeigt Ausschnitte aus dem Quell-Code von Valgrinds Tool Nulgrind.

```
#include "pub_tool_basics.h"
#include "pub_tool_tooliface.h"

static void nl_post_clo_init(void)
{
}
static IRSB* nl_instrument ( VgCallbackClosure* closure,
                             IRSB* bb,
                             VexGuestLayout* layout,
                             VexGuestExtents* vge,
                             IRType gWordTy, IRType hWordTy ) {
    return bb;
}
static void nl_fini(Int exitcode) {
    return;
}
static void nl_pre_clo_init(void) {
   VG_(details_name)            ("Nulgrind");
   VG_(details_version)         (NULL);
   VG_(details_description)     ("a binary JIT-compiler");
   VG_(details_copyright_author)(
      "Copyright (C) 2002-2008, and GNU GPL'd, \
       by Nicholas Nethercote.");
   VG_(details_bug_reports_to)  (VG_BUGS_TO);

   VG_(basic_tool_funcs)        (nl_post_clo_init,
                                 nl_instrument,
                                 nl_fini);

   /* No needs, no core events to track */
}
```

Abbildung 2.2: Quell-Code des einfachsten Valgrind-Tools – Nulgrind

2.12 Valgrind in Aktion - interpretierte Ausführung des Gastes

Während der der Ausführung eines Gastes in Valgrind interagieren die LibVEX, Valgrinds Core und das Tool. Ziel ist es, den Gast während der Laufzeit zu untersuchen. Valgrinds Core, die LibVEX und das Tool laufen auf der realen CPU; der Gast wird auf der in Software simulierten CPU ausgeführt. In Abbildung 2.3 ist die Ausführung des Gastes und die Interaktion von Valgrind, der LibVEX und des Tools dargestellt.

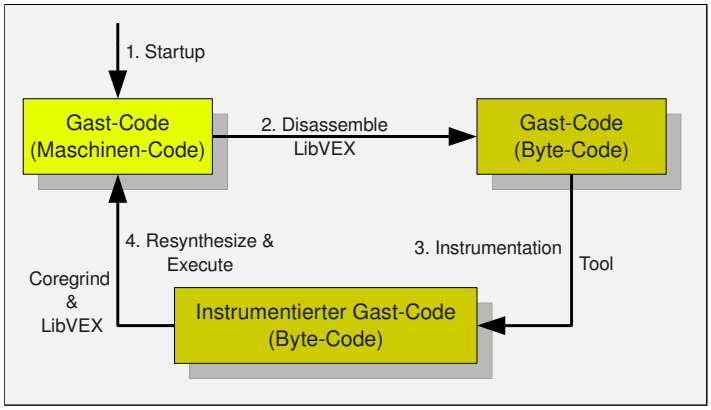

Abbildung 2.3: Interpretierte Ausführung des Gastes

Ausführungskontexte des Gastes

Der Code des Gastes wird in drei verschiedenen Kontexten ausgeführt. Je nach Kontext hat das Tool verschiedene Möglichkeiten der Kontrolle und Beobachtung:

1. Nutzer: Der gesamte Code, den der JIT-Compiler produziert, wird im Nutzermodus ausgeführt. Das Tool kann den ausgeführten Code beobachten und instrumentieren. Dabei hat das Tool die vollständige Kontrolle. Im Nutzermodus werden der Programm-Code des Gastes, fast die gesamte C-Bibliothek und alle anderen Bibliotheken ausgeführt.

2. Valgrind: Dieser Kontext umfasst alle Aktionen, die in Valgrind für den Gast ausgeführt werden. Dazu gehören das Ausführen von Systemaufrufen, das Signal-Handling, das Thread-Handling und das Scheduling. Die Tools können diese Aktionen nicht instrumentieren. Valgrinds Core stellt für das

Tool Kontroll- und Beobachtungsmöglichkeiten bereit, um auch die in Valgrinds Core ausgeführten Aktionen analysieren zu können. Beispielsweise können vor und nach Systemaufrufen die Argumente und Rückgabewerte modifiziert werden.

3. Kernel: Dieser Kontext umfasst alle Operationen, die im Betriebssystemkern ausgeführt werden. Systemaufrufe können weder vom Tool noch von Valgrinds Core direkt beobachtet oder kontrolliert werden. Indirekt können Systemaufrufe durch die jeweiligen Pre- und Post-Wrapper (siehe Abschnitt 2.8) untersucht werden.

Gast und Valgrind

Valgrinds Ziel ist es, den Gast wie unter normalen Bedingungen (ohne Valgrind) auszuführen. Da Valgrind und Gast einen Adressraum und andere Ressourcen während der Laufzeit zusammen nutzen, muss sichergestellt werden, dass keine Konflikte auftreten, die die Ausführung des Gastes beeinflussen könnten. Dies geschieht durch die in Abschnitt 2.5 genannten Methoden Partitionierung, Zeit-Multiplexing, Virtualisierung und Sharing von Ressourcen.

Ausführung des Gastes

Die Ausführung des Gastes erfolgt blockweise. Die Blöcke (*Basic Blocks*) werden während der Ausführung von der LibVEX disassembliert, durch das Tool instrumentiert und von Valgrinds Core ausgeführt. Bereits übersetzte Blöcke werden in einem Cache zwischengespeichert. Dadurch können die Geschwindigkeitsverluste, die aufgrund der interpretierten Ausführung entstehen, gemindert werden.

Valgrinds Scheduler steuert die Ausführung des Gastes und prüft periodisch, ob Signale ausgeliefert werden müssen oder es notwendig ist, zu einem anderen Thread umzuschalten. Zusätzlich wird er aktiv, sobald der Gast Systemaufrufe durchführen oder Anfragen an Valgrind senden will.

Abbildung 2.4 zeigt die Ausführung eines Programmes ohne und mit Valgrind. Dabei kann das Programm aufgrund der nativen Ausführung direkt mit dem System kommunizieren. Wird es in Valgrind ausgeführt, werden alle Zugriffe des Gastes beobachtet und kontrolliert.

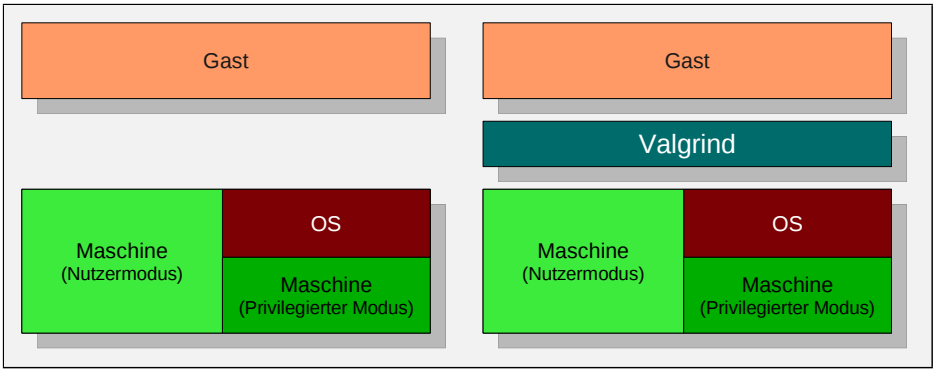

Abbildung 2.4: Ausführung eines Programmes auf einem monolithischen System: nativ (links) und in Valgrind (rechts).

3 Die Portierung

In diesem Kapitel diskutiere ich die für die Portierung von Valgrind auf L4 notwendigen Änderungen und die dabei getroffenen Entscheidungen. Von dem Start Valgrinds bis hin zur interpretierten Ausführung des Gastes sind Modifikationen aufgrund der Unterschiede zwischen Linux und L4 notwendig.

Grundlage meiner Portierung ist Valgrind mit der Version 3.4.0.SVN. Ziel ist die Portierung auf den L4-Mikrokern Fiasco für x86.

3.1 Valgrind & L4Env

Ziel dieser Arbeit ist die Portierung von Valgrind auf L4. Das L4Env dient dabei als Entwicklungsumgebung im Nutzermodus. In den folgenden Abschnitten gebe ich einen Überblick über die Änderungen, die für die Portierung von Valgrind auf L4 notwendig sind.

3.1.1 Programme im L4Env

L4Env-Programme müssen speziellen Code ausführen, bevor die `main()`-Funktion ausgeführt werden kann. Dieser Code initialisiert Bibliotheken und startet die initialen Threads der Task. Das ist nötig, um Subsysteme zu starten, die bei monolithischen Systemen wie Linux im Kern laufen:

1. Parsen der Kommandozeile und initialisieren von `argc`, `argv`.
2. Initialisierung der Log-Bibliothek.
3. Initialisierung des Region-Mappers.
4. Initialisierung der Thread-Bibliothek.
5. Initialisierung der Semaphore-Bibliothek und Start des Semaphore-Threads.
6. Start des `main()`-Threads.

7. Start des Region-Mappers.

L4Env-Programme müssen beispielsweise Speicher und Semaphoren selbst verwalten beziehungsweise bereitstellen. In Linux sind Speicher und Semaphoren Aufgaben des Betriebssystemkerns.

3.1.2 Valgrind im L4Env

Unter Linux dient der Valgrind-Loader zum Starten von Valgrind. Dieser startet entsprechend der Argumente das jeweilige Tool mit `exec()`. Da L4 über keine Implementation von `exec()` verfügt, verwende ich den Valgrind-Loader nicht. Stattdessen wird das jeweilige Tool in L4 direkt gestartet.

Damit Valgrind auf L4 als L4Env-Programm gestartet werden kann, muss der im vorangegangenen Abschnitt beschriebene Start-Code des L4Env in Valgrind integriert werden. Dies erfordert minimale Änderungen in Valgrind, einzig das Linker-Skript von Valgrind muss angepasst werden. Bei der Initialisierung des Region-Mappers – zuständig für Speicherverwaltung und Pagefaults – muss diesem der Start und das Ende der Text- und Daten-Sektionen übergeben werden. Der Start und das Ende der Sektionen wird im Binary mit den Symbolen `_prog_img_start` und `_prog_img_end` markiert.

Zum Starten von Valgrind auf L4 wird als erstes der Start-Code des L4Env ausgeführt. Anschließend verläuft der Start von Valgrind ähnlich dem unter Linux (siehe Abschnitt 2.4):

0. Initialisierung des L4Envs (Ausführung des L4Env-Start-Codes).
1. Valgrind wechselt auf einen eigenen Stack.
2. Analyse der Umgebung und der Kommandozeilen-Argumente.
3. Start der Speicherverwaltung, dabei wird der initiale Aufbaus des Adressraumes (siehe Abschnitt 2.7) ermittelt.
4. Laden des Gast-Binarys (Daten- und Text-Segment).
5. Vorbereitung von Stack- und Daten-Segment des Gastes.
6. Initialisierung des Tools.
7. Initialisierung der LibVEX.
8. Initialisierung des Schedulers.

9. Laden der Debug-Informationen des Gastes.

10. Beginn der interpretierten Ausführung des Gastes.

Nach der Ausführung des L4Env-Start-Codes initialisiert Valgrind sich selbst, lädt das Gast-Programm und bereitet die Ausführung des Gastes vor. In Abbildung 3.1 ist die Ausführung eines Programmes auf L4 dargestellt. Die linke Grafik zeigt die native Ausführung eines Programmes auf L4. Das Programm hat direkten Zugriff auf Teile der Maschine (z.B. Register), auf andere Teile kann nur mit Hilfe des Betriebssystems zugegriffen werden. Die rechte Grafik zeigt die Ausführung in Valgrind. Bei dieser werden alle Zugriffe des Gastes durch Valgrind beobachtet und kontrolliert.

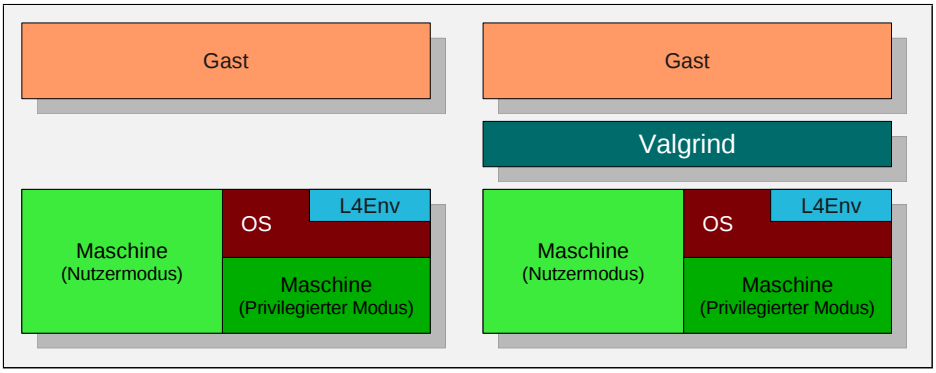

Abbildung 3.1: Ausführung eines Programmes auf einem mikrokern-basierten System: nativ (links) und in Valgrind (rechts).

Als nächstes gebe ich einen Überblick über zur Portierung notwendiger Änderungen in Valgrind und verweise für Details auf die entsprechenden Abschnitte.

Speicherverwaltung Valgrind ermittelt beim Start den initialen Aufbau seines Adressraumes. In Linux kommuniziert Valgrind dazu mit dem Kern, da dieser den Adressraumes verwaltet. In L4 müssen solche Informationen, bedingt durch den Aufbau als Multi-Server-OS, beim jeweiligen Server abgefragt werden. Den Aufbau des Adressraumes verwaltet unter L4 der Region-Mapper. Weitere Probleme und Details der Portierung des Speicherverwaltungssystems von Valgrind erläutere ich in Abschnitt 3.2.3.

Kommunikation mit dem Kern Die LibVEX ist weitgehend unabhängig vom Betriebssystem, daher sind nur kleine Änderungen notwendig. Um die L4-Systemaufrufe des Gastes in Valgrind behandeln zu können, muss die LibVEX modifiziert werden. Valgrinds Core führt die Systemaufrufe des Gastes aus. In Abschnitt 3.3.2 erläutere ich dazu notwendige Modifikationen und dabei getroffene Entscheidungen.

Tasks und Threads Unter L4 werden Tasks und Threads mit den Systemaufrufen *task_new()* und *lthread_ex_regs()* erzeugt. In Abschnitt 3.4.1 beschäftige ich mich mit der Integration von L4-Tasks und -Threads in Valgrind.

3.1.3 C-Bibliothek, POSIX & L4

Das L4Env nutzt als C-Bibliothek die *uClibc* [6]. Die uClibc ist für eingebettete Linux-Systeme konzipiert. Im Gegensatz zur Glibc ist sie wesentlich kleiner, da nicht benötige Features deaktiviert werden können.

Um den gesamten Umfang der uClibc nutzen zu können, sind sogenannte *Backends* notwendig. Das sind beispielsweise Funktionen zum Arbeiten mit Dateien wie `open()`, `read()`, `write()` und `close()`. Diese stellen Funktionalität zur Verfügung, die bei monolithischen Systemen wie Linux oder Windows im Kern zu finden ist. Im L4Env werden diese Backends durch das *L4VFS* (*L4 virtual file system layer*) bereitgestellt.

Das L4VFS besteht aus Bibliotheken und Servern. Diese stellen beispielsweise einen UNIX-artigen Namensraum, `mmap()` und Dateideskriptoren zur Verfügung. Hauptziel bei der Entwicklung des L4VFS war es, die Portierung von POSIX Programmen auf L4 zu erleichtern.

Außerdem gehört gehört zum L4VFS auch ein Server, der POSIX-Signale zur Verfügung stellt. Da die Implementation der POSIX-Signale jedoch unvollständig ist, unterstützt L4/Valgrind diese nicht. Dies stellt keine Einschränkung dar, da die meisten L4-Anwendungen keine Signale benutzen.

Valgrind ist unter Linux, wie im Abschnitt 2.6 beschrieben, unabhängig von der C-Bibliothek. Dies ist möglich, da alle von Valgrind benötigten Ressourcen in monolithischen Systemen vom Kern bereitgestellt werden. In L4 werden aufgrund der Mikrokern-Architektur Ressourcen mit Servern im Nutzermodus verwaltet und durch Bibliotheken bereitgestellt. Da L4/Valgrind Ressourcen wie den Zugriff auf Dateien benötigt, ist es von Bibliotheken des L4Env abhängig. Um die Abhängigkeit zu umgehen, könnte Valgrind auch direkt mit den Servern per IPC kommunizieren. Dies hätte jedoch keine weiteren Vorteile.

3.1.4 Anfragen des Gastes

Um es dem Gast zu ermöglichen, Anfragen an Valgrind oder das Tool zu senden, muss zum Code des Gastes weiterer hinzugefügt werden. Unter Linux können mit dem `LD_PRELOAD`-Mechanismus zusätzliche Bibliotheken vor der Programmausführung geladen werden. L4 verfügt jedoch über keinen solchen Mechanismus. Als Ersatz kann der zusätzliche Code, den Valgrind benötigt, um die Anfragen des Gastes zu realisieren, in das Gast-Binary gelinkt werden.

Durch das Hinzufügen des Codes wird der eigentliche Code des Gastes nicht verändert. Das Gast-Programm ist weiterhin direkt ohne Valgrind ausführbar.

3.2 Speicherverwaltung

Valgrind und der Gast benötigen zur Laufzeit Speicher. Da Valgrind und Gast gemeinsam in einem Adressraum laufen, ist es notwendig, diesen zu partitionieren. Während in Linux die Speicherverwaltung vollständig durch den Kern erfolgt, wird in L4 der Speicher im Nutzermodus außerhalb des Kerns verwaltet.

Im Folgenden zeige ich, welche Änderungen an Valgrind deshalb notwendig waren. Dazu stelle ich im ersten Teil vor, wie die Verwaltung von Speicher in L4 und im L4Env erfolgt. Im zweiten Teil diskutiere ich deren Realisierung in Valgrind.

3.2.1 Speicherverwaltung in L4

In L4 findet die Speicherverwaltung im Nutzermodus statt. Jede Anwendung hat ihren eigenen virtuellen Adressraum, der aus Seiten zusammengesetzt ist. Speicherverwaltung umfasst das Ein- und Ausblenden von Seiten und die Behandlung von Pagefaults (Seitenfehler). Dies geschieht durch Modifikation der Seitentabellen, die jedoch nur durch den Kern erfolgen kann. Um die Speicherverwaltung trotzdem im Nutzermodus durchzuführen zu können, ist also Unterstützung durch den Kern nötig. Diese Unterstützung erfolgt durch Flexpages und Operationen, die auf diese angewendet werden können. Flexpages beschreiben Bereiche des virtuellen Adressraumes und enthalten alle in diesen Bereich eingeblendeten Seiten. Das Ein- und Ausblenden von Seiten erfolgt durch Versenden von Flexpages per IPC. Dazu stellt der Kern folgende Operationen zur Verfügung:

- `grant` Die Speicherseite wird an eine andere Task übergeben und steht der ursprünglichen Task nicht mehr zur Verfügung.
- `map` Die Speicherseite wird an eine andere Task übergeben und steht anschließend beiden beteiligten Tasks zur Verfügung.
- `flush` Die Speicherseite wird aus den Adressräumen aller Tasks, die Mappings vom Aufrufer erhalten haben, entfernt.

Pager In L4 erfolgt die Verwaltung des Speichers, also das Ein- und Ausblenden von Seiten und die Behandlung von Pagefaults, durch Pager. Jeder L4-Thread kann seinen eigenen Pager nutzen. Die Pager selbst sind ebenfalls separate L4-Threads. Pagefaults werden durch den Kern mittels IPC zugestellt.

In L4 können hierarchische Pager-Strukturen konstruiert werden. Dabei wird der Adressraum eines Pagers von einem weiteren Pager verwaltet. Die Wurzel

der Hierarchie bildet der Rootpager (*sigma0*). Er bildet den ersten Adressraum im System. Beim Start des Systems wird der gesamte verfügbare physische Speicher in den Adressraum des Rootpagers eingeblendet. Dies geschieht durch Verwendung von Flexpages und der Operation `grant`.

Ein wichtiger Mechanismus, den Pager nutzen, sind Pagefaults. Diese treten auf, sobald ein Programm auf Speicherseiten zugreift, denen kein Speicher zugeordnet ist. Der Pager löst – unter Nutzung des beschriebenen Flexpage-Mechanismus – die Pagefaults auf. Anschließend wird das Programm fortgesetzt, das den Pagefault auslöste. In folgendem Abschnitten möchte ich den grundlegenden Unterschied der Behandlung von Pagefaults in mikrokern-basierten Systemen gegenüber monolithischen Systemen erläutern.

Pagefaults In klassischen monolithischen Systemen werden Pagefaults vom Kern aufgelöst. Abbildung 3.2 zeigt, wie Programm und Kern bei Pagefaults interagieren. Dies läuft in drei Schritten ab:

- **Schritt 1:** Auftreten eines Pagefaults in einem Programm.
- **Schritt 2:** Behandlung des Pagefaults durch den Kern.
- **Schritt 3:** Fortsetzung des Programms.

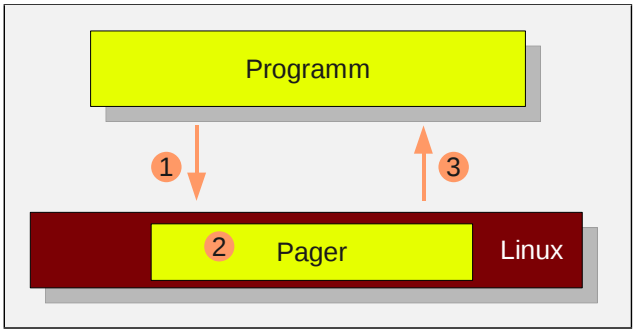

Abbildung 3.2: Pagefault-Handling in Architekturen mit monolithischem Kern, z.B. in Linux, Windows

Im Gegensatz dazu erfolgt bei mikrokern-basierten Systemen die Behandlung eines Pagefaults im Nutzermodus [18]. Entsprechend anders läuft die Behandlung gegenüber dem monolithischen System ab (siehe Abbildung 3.3):

- **Schritt 1:** Auftreten eines Pagefaults in einem Programm.

- **Schritt 2:** Kern leitet den Pagefault in Form einer IPC an Pager weiter.
- **Schritt 3:** Behandlung des Pagefaults.
- **Schritt 4:** Pager beantwortet IPC mit Mapping.
- **Schritt 5:** Fortsetzen des Programms.

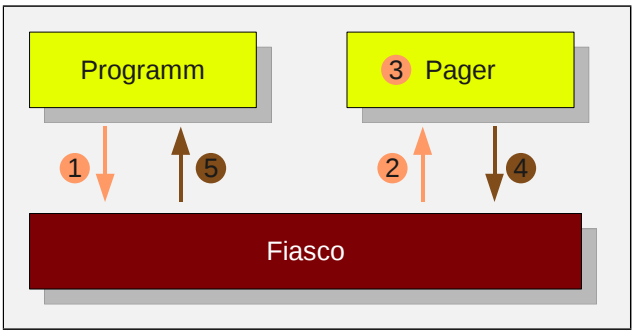

Abbildung 3.3: Pagefault-Handling in Mikrokern-Architekturen wie L4

3.2.2 Region-Mapper und Dataspace-Manager

Die Verwaltung von Speicher im L4Env basiert auf dem Konzept der Dataspaces. Dataspaces sind Container für unterschiedliche Arten von Speicher, zum Beispiel für physischen Speicher, Dateien oder I/O. Die Verwaltung von Dataspaces erfolgt durch Dataspace-Manager; beispielsweise dient DMphys zur Verwaltung des physischen Speichers. Dataspaces können in Bereiche (Regions) des Adressraumes eingeblendet werden. In den Adressraum eines Programmes können mehrere Dataspaces eingeblendet werden. Um Konflikte zu vermeiden, die beispielsweise durch sich überlappende Dataspaces entstehen, ist es notwendig, den virtuellen Adressraum jeder Task einzeln zu verwalten. Dazu stellt das L4Env als Pager und Verwalter des Adressraumes den L4-Region-Mapper zur Verfügung.

Die Aufgabe des L4-Region-Mappers ist die Verwaltung des virtuellen Adressraumes einer Task [24] und das Auflösen von Pagefaults. Dazu verwaltet dieser eine Liste von Regions (Region-List), die benutzte Bereiche des Adressraumes enthält. Darauf basierend bietet der Region-Mapper Funktionen zum Finden und Reservieren von freien Bereichen im virtuellen Adressraum und zum Ein- und Ausblenden von Dataspaces in Regions. Mit `l4rm_do_attach()` und

`l4rm_detach()` können beispielsweise Dataspaces in den Adressraum ein- und ausgeblendet werden.

Neben der Verwaltung des Adressraumes fungiert der Region-Mapper als Pager. In L4, ist es möglich für jeden Thread einen anderen Pager festzulegen. Im Falle einer L4Env-Task nutzen alle Threads den gleichen Pager. Auftretende Pagefaults innerhalb einer L4Env-Task werden daher durch den Kern zum Region-Mapper weitergeleitet. Entsprechend der Pagefault-Adresse wird der verantwortliche Dataspace-Manager mit Hilfe der Region-List ermittelt; an diesen wird der Pagefault dann weitergeleitet.

3.2.3 Valgrind, Gast und Region-Mapper

Im ersten Teil dieses Abschnittes setze ich mich mit Problemen und Lösungen bei der Verwaltung von Speicher in Valgrind auf L4 auseinander. Im zweiten Teil stelle ich die Implementation dessen vor.

In L4 wird Speicher, wie in Abschnitt 3.2.1 beschrieben, im Nutzermodus verwaltet. Die Aufgaben der Speicherverwaltung sind:

1. Verwaltung des virtuellen Adressraumes und

2. Behandlung von Pagefaults.

Die Verwaltung des Speichers im L4Env wird mit dem im vorherigen Abschnitt beschriebenen Region-Mapper realisiert. Da Valgrind eine eigene Anwendung ist und in seinem Adressraum den Gast als vollständige Anwendung ausführt, existieren zwei Region-Mapper in einem Adressraum. Dies führt zu Konflikten und Problemen.

Hinzu kommt Valgrinds Speicherverwaltungssystem, das zur Vermeidung von Konflikten bei der gemeinsamen Nutzung von Speicher durch Gast und Valgrind dient. Dazu benötigt es die vollständige Kontrolle über alle Speicherallokationen und -Freigaben von Valgrind und Gast.

Verwaltung des virtuellen Adressraumes

Ein erstes Problem entsteht bei der Ausführung eines Gastes in Valgrind: Da Valgrind selbst Speicher benötigt, können Konflikte bei der Benutzung des Adressraumes auftreten. Um diese Konflikte zu vermeiden, verwaltet Valgrind den Adressraum selbst. Dazu versucht Valgrind alle Änderungen am Adressraum zu verfolgen und auch zu beeinflussen. Sowohl der Region-Mapper von Valgrind sowie Valgrinds Speicherverwaltungssystem wollen also den Adressraum verwalten.

Lösung 1: Dieses Problem lässt sich lösen durch Synchronisation von Valgrind und Region-Mapper. Dabei müssen sich Valgrinds internes Speicherverwaltungssystem und der Region-Mapper bei Änderungen im Adressraum abgleichen.

Lösung 2: Eine weitere Lösung wäre die Integration des Region-Mappers in Valgrind. Dadurch würde die doppelte Verwaltung des Adressraumes wegfallen, folglich könnten auch keine Konflikte mehr auftreten. Der Nachteil dieser Lösung ist der Aufwand: In Valgrind müsste die gesamte Funktionalität eines Pagers eingebaut werden.

Ein zweites Problem entsteht bei der Ausführung eines Gastes in Valgrind. Wie oben beschrieben, verfügt Valgrind über ein eigenes Speicherverwaltungssystem und der Gast über seinen eigenen Region-Mapper. Zur Verwaltung des virtuellen Adressraumes führen beide Listen über benutzte Bereiche. Da die Verwaltung der Listen unabhängig voneinander geschieht, haben beide eine inkonsistente Sicht auf den Adressraum. Wenn der Gast oder Valgrind Speicher allozieren wollen, müssen sie dazu einen freien Bereich im virtuellen Adressraum finden. Falls nun bei der Anfrage nach einem freien Bereich Valgrinds Speicherverwaltungssystem und der Region-Mapper des Gastes das gleiche zurückgeben, entsteht folgender Konflikt: Valgrind und Gast versuchen in den gleichen Bereich Speicher einzublenden.

Lösung 1: Eine Lösung für dieses Problem ist die Partitionierung des Adressraumes. Valgrind und der Region-Mapper des Gastes verwalten jeweils unterschiedliche Bereiche des Adressraumes. Die Möglichkeit, dass beide bei der Anforderung eines freien Bereichs im Adressraum den gleichen zurückgeben, wird so ausgeschlossen.

Lösung 2: Außerdem kann das Problem durch Synchronisation von Valgrinds Speicherverwaltungssystem mit dem Region-Mapper des Gastes gelöst werden. Sobald einer der beiden Änderungen in seiner Liste vornimmt, müssen diese dem anderen mitgeteilt werden. Dadurch sind die von Valgrind und dem Region-Mapper des Gastes verwalteten Listen immer gleich.

Lösung 3: Durch Beschränkung auf einen gemeinsamen Region-Mapper für Valgrind und Gast entsteht das Problem nicht. Nachteil der Lösung ist, dass der Region-Mapper des Gastes nicht ausgeführt wird und damit auch nicht analysiert werden kann. Außerdem kann Valgrind dadurch nur Anwendungen ausführen, die sich an das L4Env-Konzept halten. Praktisch ist das aber kein Problem, da dies auf die meisten L4-Anwendungen zutrifft.

Pagefaults während der Ausführung des Gastes

Valgrind führt den Gast, wie in Abschnitt 2.3 beschrieben, in Form von Basic Blocks aus. Zusätzlich werden die Threads des Gastes nacheinander ausgeführt.

Beides zusammen führt zu Problemen bei der Behandlung von Pagefaults.

Vom Gast ausgelöste Pagefaults werden durch den Kern zum Region-Mapper des Gastes weitergeleitet. Der den Pagefault auslösende Thread und der Region-Mapper des Gastes sind Threads innerhalb von Valgrind. Sobald der Region-Mapper aktiv wird, laufen also mehrere Threads in Valgrind. Dies verletzt Valgrinds Annahme, dass immer nur ein Thread aktiv ist.

Durch den Pagefault wird die Ausführung des Gastes innerhalb eines Basic Blocks unterbrochen. Da nur der Region-Mapper des Gastes den Pagefault auflösen kann, muss dieser jetzt ausgeführt werden und verletzt dadurch die Annahme, dass Basic Blocks atomar sind und immer als Ganzes ausgeführt werden.

Lösung 1: Einen Region-Mapper für Valgrind und Gast
Das Problem lässt sich durch Beschränkung auf einen Region-Mapper lösen. Dieser wird gemeinsam von Valgrind und Gast benutzt. Der Region-Mapper wird nicht interpretiert ausgeführt, läuft also auch nicht unter Valgrinds Kontrolle. Die Behandlung von Pagefaults des Gastes erfolgt dann in folgenden Schritten:

1. Ein Pagefault tritt im Gast auf.
2. Der Kern leitet den Pagefault zum gemeinsamen Region-Mapper weiter.
3. Der Region-Mapper löst den Pagefault auf.
4. Der unterbrochenen Gast-Threads wird fortgesetzt.

Von Nachteil bei dieser Lösung ist, dass der Region-Mapper des Gastes und somit die *l4rm*-Bibliothek nicht geprüft werden kann. Ein weiterer Nachteil ergibt sich durch die Beschränkung auf L4Env-Programme, da die Lösung auf der gemeinsamen Nutzung eines Region-Mappers für Valgrind und Gast basiert. (Wie bereits weiter oben beschreiben, ist die Beschränkung aber kein Problem.) Abbildung 3.4 zeigt, wie Pagefaults des Gastes unter Benutzung eines Region-Mappers behandelt werden.

Lösung 2: Jeweils einen Region-Mapper für Gast und Valgrind
Um auch den Region-Mapper des Gastes analysieren zu können, muss dieser unter Valgrinds Kontrolle ausgeführt werden. Pagefaults des Gastes werden von Valgrinds Region-Mapper weitergeleitet; der interpretiert ausgeführte Region-Mapper des Gastes löst diese dann auf. Beim Auftreten eines Pagefaults im Gast wird der verursachende Thread unterbrochen. Durch Weiterleiten des Pagefaults wird der Region-Mapper des Gastes aktiv. Es sind also zwei Threads des Gastes aktiv; beide befinden sich in der Ausführung von Basic Blocks. Die Behandlung von Pagefaults ändert sich: Valgrind Region-Mapper behandelt Pagefaults des Gastes nicht selbst, sondern leitet diese zum Region-Mapper des Gastes weiter.

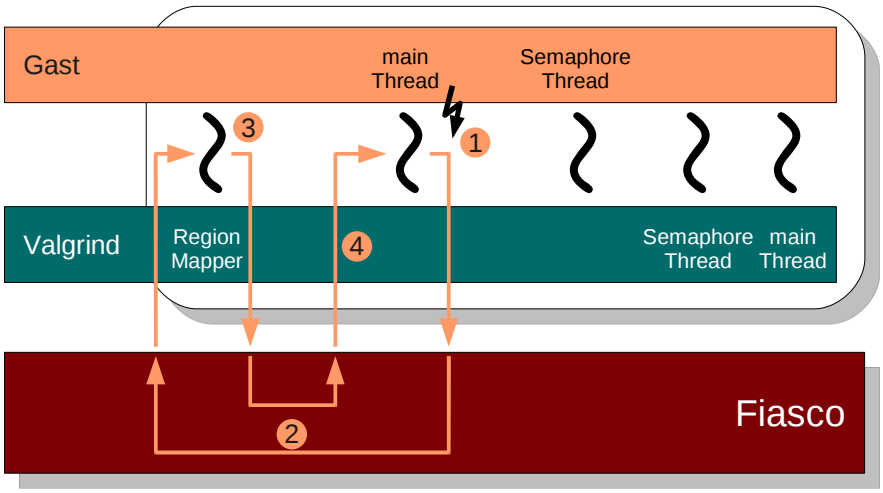

Abbildung 3.4: Beschränkung auf einen Region-Mapper

Die Behandlung eines Pagefaults verläuft, wie in Abbildung 3.5 dargestellt, in folgenden Schritten ab:

1. Ein Pagefault tritt im Gast auf.
2. Der Kern leitet den Pagefault zu Valgrinds Region-Mapper weiter.
3. Der Pagefaults wird zum Region-Mapper des Gastes weitergeleitet.
4. Der Region-Mapper löst den Pagefault auf.
5. Der unterbrochene Gast-Thread wird fortgesetzt.

Zur Implementation wären Änderungen notwendig, die zwei grundlegende Konzepte von Valgrind – Basic Blocks sind atomar und nur ein Thread ist aktiv – verletzen.

Lösung 3: Rollback von Basic Blocks

Eine weitere Lösungsmöglichkeit besteht darin, die Aktionen des Gastes die zum Pagefault geführt haben rückgängig zu machen.

Wenn der Gast einen Pagefault auslöst, befindet sich dieser innerhalb eines Basic Blocks. Es müssen also alle Aktionen, die der Gast innerhalb des Basic Blocks bereits ausgeführt hat, rückgängig gemacht werden. Danach erfolgt die Behandlung des Pagefaults. Der Gast wird fortgesetzt, indem er den rückgängig gemachten Basic Block nochmals ausführt.

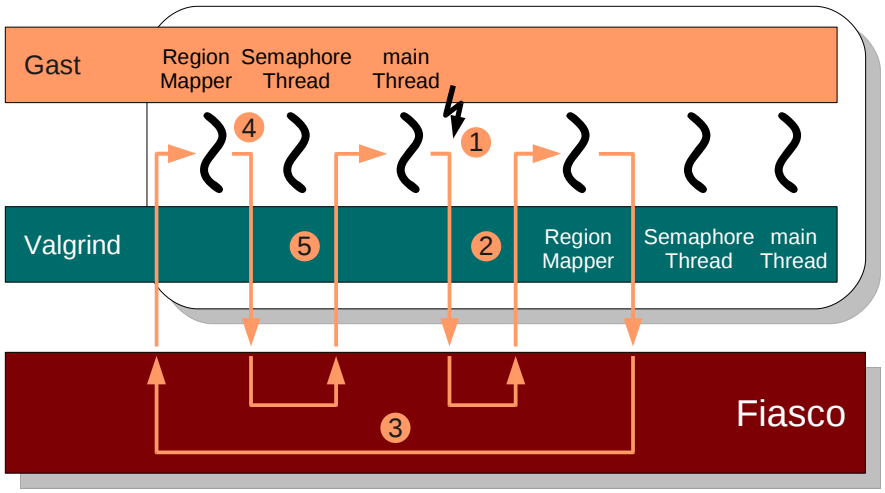

Abbildung 3.5: Verletzen der Atomaritäts-Annahme

Zum Zurücksetzen des Gastes muss der Zustand der Register, des Speichers usw. vor der Ausführung des Basic Blocks gesichert werden. Mit dieser Sicherung des Zustandes kann ein Rollback eines Basic Blocks gemacht werden.

Implementierung

L4/Valgrind beschränkt sich auf einen Region-Mapper für Valgrind und Gast. Valgrinds Region-Mapper übernimmt dabei die Aufgaben des Region-Mappers vom Gast. Trotzdem bleibt das Problem der Synchronisation von Valgrinds Speicherverwaltungssystem mit dem Region-Mapper bestehen.

Unter Linux ermittelt Valgrind dazu beim Start den initialen Aufbau des virtuellen Adressraumes. Dies wird durch Lesen und Parsen der virtuellen Datei `/proc/self/maps` realisiert. Änderungen verfolgt Valgrind durch die Beobachtung von Systemaufrufen, die den Aufbau des Adressraumes modifizieren. Solche Systemaufrufe sind beispielsweise `mmap()` und `sbrk()`.

Demgegenüber ist die Verwaltung und der Aufbau des Adressraumes im L4Env Aufgabe des Region-Mappers. Für meine Portierung musste der Region-Mapper um die Funktion

```
void* l4rm_get_region_list(void)
```

erweitert werden. Mit dieser Funktion kann auf die Region-List zugegriffen und damit der Aufbau des Adressraumes ermittelt werden.

Alle Interaktionen des Gastes mit dem Region-Mapper und Änderungen am Aufbau des virtuellen Adressraumes können durch Beobachtung der in Abbildung 3.1 genannten Funktionen verfolgt werden.

Das Beobachten erfolgt durch den in Abschnitt 2.10 beschriebenen Mechanismus des Funktionswrappings. Dabei werden alle Anfragen an den Region-Mapper des Gastes durch Valgrind abgefangen. Valgrind leitet die Anfragen an seinen Region-Mapper weiter und aktualisiert seine eigene Speicherverwaltung.

Bei Speicherallokationen des Gastes entscheidet Valgrinds Speicherverwaltung, welchen Bereich des Adressraumes der Gast nutzen darf. Mappings, bei denen der Gast einen Bereich vorgibt, werden von der Speicherverwaltung Valgrinds auf Konfliktfreiheit geprüft und gegebenenfalls verboten.

Für Speicher, den Valgrind selbst benötigt, nutzt es unter Linux den Systemaufruf *mmap* zur Speicherallokation. In L4 alloziert Valgrind Speicher mit der Bibliotheksfunktion `mmap()` aus der uClibc. Wie unter Linux aktualisiert Valgrind in L4 dabei sein Speicherverwaltungssystem.

3.3 Kommunikation mit dem Kern

Systemaufrufe sind notwendig, damit Programme, die im Benutzer-Adressraum ausgeführt werden, mit dem Kern, den Treibern oder auch anderen Programmen kommunizieren können. Valgrind führt zur Kontrolle und zum Beobachten alle Systemaufrufe des Gast-Programmes selbst durch. Daher ist es notwendig, dass Valgrind alle Systemaufrufe kennt, die Gast-Programme nutzen könnten.

Um neue Systemaufrufe in Valgrind zu integrieren, müssen folgende Teile angepasst werden:

1. LibVEX
2. Systemaufruf-Behandlung in Valgrind (`m_syswrap`)

Die LibVEX dient zum Disassemblieren des Gast-Codes. Systemaufrufe des Gastes werden durch Funktionsaufrufe ersetzt. Ziel der Funktionsaufrufe ist der Scheduler von Valgrind; dieser führt die Systemaufrufe des Gastes dann aus.

Um Systemaufrufe beobachten und kontrollieren zu können bietet Valgrind die Möglichkeit, Funktionen zu definieren, die vor bzw. nach dem Aufruf eines Systemaufrufes ausgeführt werden. Bei der Integration eines neuen Systemaufrufes in Valgrind müssen diese entsprechend implementiert werden. In den Funktionen können Vor- und Nachbedingungen geprüft und durchgesetzt werden. Bei blockierenden Systemaufrufen ist es beispielsweise notwendig, das globale Lock vor der Ausführung abzugeben und nachher sofort wieder zu verlangen (siehe

- `int l4rm_do_attach(const l4dm_dataspace_t * ds, l4_uint32_t area,`
  `l4_addr_t * addr, l4_size_t size,`
  `l4_offs_t ds_offs, l4_uint32_t flags);`
  `int l4rm_detach(const void * addr)`
  Funktionen zum Ein- und Ausblenden von Dataspaces in den virtuellen Adressraum.

- `l4rm_do_reserve(l4_addr_t * addr, l4_size_t size,`
  `l4_uint32_t flags, l4_uint32_t * area)`
  `l4rm_area_release(l4_uint32_t area)`
  `l4rm_area_release_addr(const void * ptr)`
  `l4rm_area_clear_region(l4_addr_t addr)`
  `l4rm_do_area_setup(l4_addr_t* addr, l4_size_t size,`
  `l4_uint32_t area, int type,`
  `l4_uint32_t flags, l4_threadid_t pager)`
  Funktionen zum Reservieren, Freigeben und Verwalten von Bereichen im Adressraum.

- `l4rm_lookup(const void * addr, l4_addr_t * map_addr,`
  `l4_size_t * map_size, l4dm_dataspace_t * ds,`
  `l4_offs_t * offset, l4_threadid_t * pager)`
  `l4rm_lookup_region(const void * addr, l4_addr_t * map_addr,`
  `l4_size_t * map_size, l4dm_dataspace_t * ds,`
  `l4_offs_t * offset, l4_threadid_t * pager)`
  Beide Funktionen dienen zur Abfrage des Region-Mappers.

- `l4rm_get_userptr(const void *addr)`
  `l4rm_set_userptr(const void *addr, void *ptr)`
  Schreiben und Lesen von nutzerdefinierten Zeigern.

- `l4rm_disable_pagefault_exceptions(void)`
  `l4rm_enable_pagefault_exceptions(void)`
  `l4rm_set_unkown_fault_callback(`
  `l4rm_unknown_fault_callback_fn_t callback)`
  `l4rm_set_unkown_pagefault_callback(`
  `l4rm_pf_callback_fn_t callback)`
  Der Region-Mapper bietet die Möglichkeit, für nicht auflösbare Pagefaults oder andere Fehler Funktionen zu registrieren.

- `l4_rm_server_loop(void)`
  Zentrale Serverschleife des Region-Mappers.

- `l4rm_init(int have_l4env, l4rm_vm_range_t used[], int num_used)`
  Funktion zur Initialisierung des Region-Mappers.

- `l4rm_show_region_list(void)`
  Dient zur Ausgabe der Region-List.

- `l4rm_region_mapper_id(void)`
  Gibt die ThreadID des Region-Mappers zurück.

Tabelle 3.1: Funktionen zur Interaktion mit dem Region-Mapper
Einige Funktionen führen zu Veränderungen des Aufbaus des virtuellen Adressraumes.

Abschnitt 2.9). Zudem können die Parameter und Ergebnisse der Systemaufrufe ausgewertet werden.

3.3.1 Systemaufrufe in L4

Im Vergleich zu monolithischen Kernen enthalten Mikrokerne weniger Funktionalität im Kern. Resultierend daraus werden weniger Systemaufrufe benötigt.

Im Vergleich zu Linux werden zum Beispiel keine Systemaufrufe für Dateien (`open()`, `close()`, `read()`, `write()`) benötigt. Um mit Dateien arbeiten zu können, sind Server in Benutzer-Adressraum notwendig. Zur Kommunikation mit Servern bieten Mikrokerne IPC in Form von Systemaufrufen an. (Allgemein können per IPC Tasks und Threads kommunizieren.)

Systemaufrufe können auf x86-kompatiblen Architekturen mit Hilfe von Interrupts oder durch *SYSENTER/SYSEXIT* implementiert werden [2]. Da Valgrind für Letzteres zur Zeit noch keine Unterstützung bietet, werde ich dies auch nicht weiter ausführen.

L4 spezifiziert sieben Systemaufrufe [10]:

- ***ipc*** – Interprozesskommunikation und Synchronisation (`int 30`)
- ***id_nearest*** – Ermitteln der eigenen Thread-ID (`int 31`)
- ***fpage_unmap*** – Speicherverwaltung (`int 32`)
- ***thread_switch*** – Umschalten zu einem anderen Thread (`int 33`)
- ***thread_schedule*** – Scheduling, Festlegen von Prioritäten etc. (`int 34`)
- ***lthread_ex_regs*** – Erstellen und Modifizieren von Threads (`int 35`)
- ***task_new*** – Erstellen und Löschen von Tasks (`int 36`)

Zusätzlich zu diesen sind in Fiasco zwei weitere Systemaufrufe implementiert:

- ***privctrl*** – L4 privilege control (`int 37`)
- ***ulock*** – Lock für den Nutzermodus (`int 39`)

3.3.2 Integration der L4-Systemaufrufe in LibVEX und Valgrind

Die LibVEX bildet Systemaufrufe des Gastes auf Funktionen von Valgrind ab. Um die Systemaufrufe von L4 zu integrieren, müssen der LibVEX alle von L4 genutzten Interrupts bekannt gemacht werden. Sobald der Gast einen Systemaufruf startet, wird eine entsprechende Funktion zum Behandeln des Systemaufrufs in Valgrind ausgeführt.

Valgrinds Core stellt einen Mechanismus zur Verfügung, mit dem vor und nach der Ausführung jedes Systemaufrufes des Gastes bestimmte Aktionen durchgeführt werden können. Für jeden L4-Systemaufruf müssen jeweils zwei Funktionen definiert werden.

Die Behandlung des L4-Systemaufrufes *lthread_ex_regs* zum Erstellen und Modifizieren von Threads habe ich unter Wiederverwendung des Valgrind-Codes für den Linux-Systemaufruf *clone* implementiert. Mehr Details zu Threads beschreibe ich in Abschnitt 3.4.1.

Ein Sonderfall ist der Systemaufruf *ipc*. Dieser kann abhängig von den Parametern blockierend sein. Wie in Abschnitt 2.8 beschrieben, wird vor der Ausführung von blockierenden Systemaufrufen in Valgrind das Lock abgegeben – folglich können andere Threads ausgeführt werden – und nach dem Systemaufruf sofort wieder verlangt. Realisiert wird das Abgeben des Locks mittels eines Flags, dieses wird vor dem Systemaufruf *ipc* gesetzt. Ein weiterer blockierender Systemaufruf ist *ulock*. Vor der Ausführung wird wie bei *ipc* das Lock abgegeben. Sobald der Systemaufruf zurückkehrt wird das Lock benötigt um weiter auszuführen.

3.4 Tasks und Threads

3.4.1 Tasks und Threads in L4

In L4 besteht eine Task aus einem virtuellen Adressraum und möglicherweise mehreren, jedoch mindestens einem Thread. Mit dem Systemaufruf *task_new* können neue Tasks erzeugt oder alte gelöscht werden. Bei der Erzeugung einer L4-Task werden alle Threads mit erstellt, bleiben aber inaktiv.

Threads sind Aktivitäten, die in einem Adressraum ausgeführt werden. Sie sind eindeutig durch ThreadIDs identifizierbar und können mittels IPC miteinander kommunizieren. Um ein Thread zu starten, muss dieser mit Hilfe des Systemaufrufes *lthread_ex_regs* aktiviert werden; dabei werden valide Funktions- und Stackpointer gesetzt.

3.4.2 Tasks und Threads im L4Env

Um Tasks schnell und einfach erzeugen und löschen zu können, stellt das L4Env den *Simple Task Server* zu Verfügung. Zusätzlich vereinfacht eine Bibliothek die Verwaltung von Threads. Mit ihr können Threads erstellt (aktiviert), modifiziert oder gelöscht (deaktiviert) werden.

Wie in Abschnitt 3.2.2 beschrieben, wird Speicher in L4 im Nutzermodus verwaltet. Das L4Env stellt dazu den Region-Mapper zur Verfügung. Außerdem stellt das L4Env zur Synchronisation Semaphoren zur Verfügung. Beides wird durch je einen Thread realisiert. Zusammen mit dem Thread, der die `main()`-Funktion ausführt, besteht jede L4Env-Anwendung aus mindestens drei Threads.

3.4.3 Threads in Valgrind

Valgrind unterstützt, wie in Abschnitt 2.9 beschrieben, Threads. Dabei werden diese serialisiert nacheinander ausgeführt. Umgesetzt wird die Serialisierung durch ein globales Lock, das ein Thread zur Ausführung benötigt.

Das Lock Unter Linux ist dieses Lock mittels `pipe` implementiert. Mit dessen Hilfe können gepufferte Datenströme zwischen Threads und Tasks erzeugt werden. Da es in L4 keine Implementation von `pipe` gibt, habe ich das Lock mit den vom L4Env bereitgestellten Semaphoren realisiert. In Abbildung 3.6 ist die Implementation von Valgrinds Semaphoren dargestellt.

Threads Unter Linux können neue Threads oder Tasks mit den Systemaufrufen *clone* oder *fork* erzeugt werden. Dabei ist ein neuer Thread bzw. Task immer zuerst eine Kopie seines Erzeugers.

In L4 werden neue Task mit dem Systemaufruf *task_new* und neue Threads mit dem Systemaufruf *lthread_ex_regs* erzeugt. Zum Erstellen von neuen Threads in L4 ist ein Funktionspointer und ein Zeiger auf einen gültigen Stack notwendig.

Valgrind besitzt, da es ein L4Env-Programm ist, initial drei Threads. Der erste Thread ist der Region-Mapper. Dieser ist kein direkter Bestandteil von Valgrind, wird jedoch zur Verwaltung des Adressraumes und zur Behandlung von Pagefaults benötigt. Der zweite Thread stellt Semaphoren bereit, der dritte Thread führt die `main()`-Funktion aus. Da Valgrinds `main()`-Funktion am Ende direkt zur interpretierten Ausführung des ersten Gast-Threads übergeht, führt der dritte Thread von Valgrind den ersten Gast-Thread aus.

Wie jedes L4Env-Programm besitzt auch der Gast initial drei Threads. Der erste Thread des Gastes ist dessen Region-Mapper. Wie in Abschnitt 3.2.3 beschrieben

```
void ML_(sema_init)(vg_sema_t *sema) {
   sema->sem = L4SEMAPHORE_INIT(1);
   sema->owner_lwpid = -1;
}

void ML_(sema_deinit)(vg_sema_t *sema)
{
   vg_assert(sema->owner_lwpid != -1); /* must be initialised */
   sema->owner_lwpid = -1;
   l4semaphore_up(&(sema->sem));
}

void ML_(sema_down)(vg_sema_t *sema) {
   Int lwpid = VG_(gettid)();
   vg_assert(lwpid != 0);
   vg_assert(sema->owner_lwpid != lwpid); /* can't have it already */

   l4semaphore_down(&(sema->sem));
   sema->owner_lwpid = lwpid;
}

void ML_(sema_up)(vg_sema_t *sema) {
   vg_assert(sema->owner_lwpid != -1); /* must be initialised */
   vg_assert(sema->owner_lwpid == VG_(gettid)()); /* must have it */

   l4semaphore_up(&(sema->sem));
   sema->owner_lwpid = 0;
}
```

Abbildung 3.6: Implementation der Semaphoren in Valgrind im L4Env

wird der Region-Mapper des Gastes nicht ausgeführt. Anstatt also die Serverschleife des Region-Mappers zu starten, wird der erste Gast-Thread „schlafen gelegt“. Der zweite Thread - der Semaphore-Thread - wird wie jeder andere Thread des Gastes interpretiert von Valgrind ausgeführt. Der Dritte führt die `main()`-Funktion des Gastes aus.

Ziel von Valgrind ist es, die Ausführung des Gastes so zu gestalten, als würde das Gast-Programm direkt auf dem System ausgeführt. Um dies auch unter L4 zu gewährleisten wird an den Semaphore-Thread und den `main()`-Thread von Valgrind beim Start eine hohe ThreadID vergeben. Valgrinds Region-Mapper bleibt erster Thread, da Valgrind und Gast einen Region-Mapper zusammen nutzen, wie in Abschnitt 3.2.3 beschrieben.

Folglich bekommen der Semaphore-Thread und der `main()`-Thread des Gastes durch die „Verschiebung“ der Threads von Valgrind die gleiche ThreadID, als wären sie nicht interpretiert ohne Valgrind ausgeführt worden. Abbildung 3.7 zeigt die Threads von Valgrind und die des Gastes in L4. Dabei bezeichnen die Zahlen neben den Threads die Reihenfolge ihrer Erstellung. Über den Threads ist zusätzlich die jeweilige ThreadID angegeben.

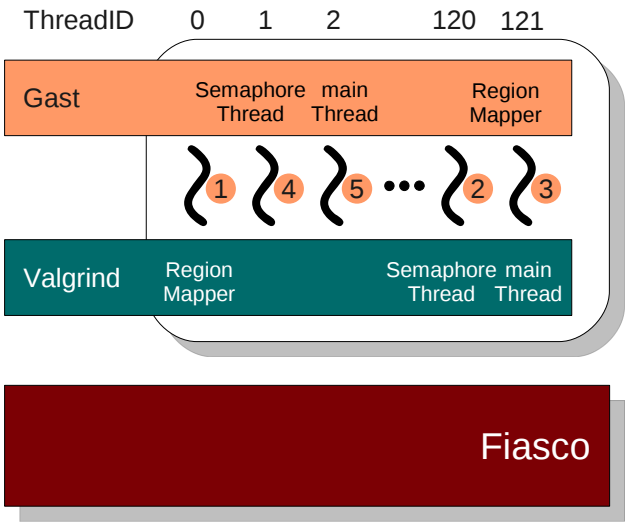

Abbildung 3.7: Threads in Valgrind

Starten von Threads

Threads werden in L4 mit dem Systemaufruf *lthread_ex_regs*, siehe Abschnitt 3.4.1,

erstellt (aktiviert). Wie in Abschnitt 2.8 beschrieben, leitet die LibVEX Systemaufrufe des Gastes zu Valgrind weiter.

Bei der Erstellung von Threads muss Valgrind sicherstellen, dass es nicht die Kontrolle über die Ausführung verliert. Das bedeutet, dass ein neuer Thread niemals direkt nach der Erstellung den Gast-Code ausführen sollte. Stattdessen sollte direkt nach der Erstellung des Threads eine spezielle Funktion in Valgrind ausgeführt werden. Aufgabe dieser Funktion ist es, den Thread für die blockweise Ausführung des Gast-Codes vorzubereiten.

Basierend auf dem Code zur Behandlung von *clone* in Linux habe ich die Behandlung von L4-Threads in Valgrind implementiert.

Beendigung von Threads

Genauso wie beim Starten von Threads muss Valgrind beim Beenden sicherstellen, dass es nicht die Kontrolle über die Ausführung des Gastes verliert. Jeder Thread in Valgrind hat zwei Kontexte – den des Gastes und den von Valgrind. Wenn es dem Gast nun möglich wäre, vorbei an Valgrind einen Thread zu beenden, würde dadurch auch ein Thread von Valgrind beendet werden.

Um dies zu verhindern, fängt Valgrind den Aufruf der Funktion `__do_exit` des Gastes ab und beendet den Thread selbst.

3.4.4 Tasks in Valgrind

Mit meiner Portierung von Valgrind auf L4 können vollständige L4Env-Tasks in Valgrind ausgeführt und analysiert werden.

In Linux „folgt“ Valgrind bei Angabe des Parameters `--trace-children` allen Kind-Tasks. „Folgen“ bedeutet, dass die neuen Tasks auch unter Valgrinds Kontrolle ausgeführt werden. Dadurch werden alle Tasks, die ein Gast (laufend in Valgrind) erstellt, automatisch analysiert. Implementiert ist dies mit Hilfe des Systemaufrufes *fork*, der eine Kopie des gesamten laufenden Tasks erzeugt.

Da es in L4 keine Implementierung von *fork* gibt, habe ich den Mechanismus des „Folgens“ in L4/Valgrind deaktiviert.

Eine mögliche Implementation von *fork* für L4 müsste als erstes eine neue Task erstellen, danach alle bestehenden Mappings duplizieren. Abschließend kann die neue Task gestartet werden. Im Detail könnte die *fork*| folgendermaßen implementiert werden:

1. Erstellen einer neuen Task und Vorbereitung des ersten Threads der neuen Task.
2. Duplizieren aller Mappings, damit der erste Thread – der Region-Mapper – der neuen Task laufen kann.

3. Starten der neuen Task, d.h. nur der Region-Mapper-Thread der Task wird gestartet.

4. Der Region-Mapper der neuen Task dupliziert alle restlichen Mappings. Dazu wäre eine Synchronisation der Region-Mapper der neuen und der alten Task notwendig. Das Duplizieren der Mappings erfordert eine Kooperation mit den Dataspace-Managern. Im Detail müssten die Dataspace-Manager entscheiden, ob der jeweilige Dataspace gemeinsam genutzt werden kann oder dupliziert werden muss.

5. Erstellen und Starten der restlichen Threads der neuen Task.

Mit dieser Implementation könnte Valgrinds Mechanismus des „Folgens“ von Kind-Tasks in L4 bei der Analyse verwendet werden.

4 Auswertung

Ziel dieses Kapitels ist die Zusammenfassung der Ergebnisse meiner Portierung. Dazu stelle ich im ersten Teil die Standard-Tools von Valgrind vor und erläutere ihre Besonderheiten und Einschränkungen. Im zweiten Teil bewerte ich die Geschwindigkeit von L4/Valgrind.

4.1 Die Test-Umgebung

Alle in diesem Kapitel beschriebenen Messungen sind auf dem gleichen Test-Rechner durchgeführt worden:

- Intel Core 2 T7400 2.16 GHz CPU
- 2GB RAM
- Linux Ubuntu 8.10 x86
- Fiasco DD-L4(v2)/ia32

Alle benötigten Binaries und Dateien befinden sich für die Tests und die Messungen in einer Ramdisk.

4.2 Die Valgrind-Standard-Tools auf L4

Valgrind ist ein Framework zum Entwickeln von Analyse-Tools und zum Testen von Programmen. Zu Valgrind gehören acht Standard-Tools.

Im Rahmen der Portierung von Valgrind auf L4 habe ich die einzelnen Tools untersucht, inwiefern diese unter L4 einsetzbar sind. Im Folgenden beschreibe ich die Standard-Tools von Valgrind und die Einsatzmöglichkeiten unter L4.

Memcheck

Memcheck hilft beim Finden von Fehlern in der Speicherverwaltung eines Programmes. Alle Lese- und Schreibzugriffe sowie die Funktionsaufrufe der Funktionen `malloc()`, `new()`, `free()`, `delete()` werden dazu verfolgt und überprüft. Während andere „Speicherchecker“ nur auf Byte- oder Word-Ebene Fehler suchen, ist es mit Memcheck möglich, auf Bit-Ebene Fehler zu finden [26].

Prinzipiell lässt sich Memcheck für L4Env-Programme nutzen. Im L4Env gibt es neben `malloc()` noch andere Möglichkeiten, Speicher zu allozieren. Beispielsweise können sich L4Env-Programme bei DMphys einen Dataspace mit physischem Speicher allozieren und diesen mit Hilfe des Region-Mappers in den Adressraum einblenden. Um auch diese Speicherallokationen verfolgen zu können, müssen Funktionen wie `l4rm_do_attach()` und `l4rm_detach()` durch Memcheck überwacht werden.

```
{ // bug1
    int x, y, z;
    x = ( x == 0 ? y : z);
}
{ // bug2
    char *c = malloc(sizeof(char) * 16);
    int i;
    for (i = 0; i < 16; i++) {
        *(c + i) = '0';
    }
    *(c + i + 1) = '\n';
}
{ // bug3
    char *c1 = malloc(sizeof(char) * 16);
    char *c2;
    int i;

    for (i = 0; i < 15; i++) {
        *(c1 + i) = '0';
    }
    *(c1 + 15) = '\0';

    c2 = c1;
    c2 += 8;

    strncpy(c2, c1, 15);
}
```

Abbildung 4.1: Quellcode mit Programmierfehlern, die Memcheck erkennen und beschreiben kann

Ein großer Vorteil von Memcheck ist die Genauigkeit der Fehlerbeschreibungen. Memcheck liefert eine Beschreibung des Fehlers und die genaue Position im Quellcode (vorausgesetzt, das Binary wurde mit Debuginformationen übersetzt). Abbildung 4.1 zeigt drei Programmierfehler, die Memcheck erkennt. Der erste Fehler entsteht bei der Benutzung der nicht initialisierten Variable `x` zur Sprungentscheidung. Memcheck erkennt dies und meldet den Fehler:

```
Conditional jump or move depends on uninitialised value(s)
  at 0x8048409: main (bugs.c:8)
```

Beim zweiten Fehler werden 16 Byte Speicher alloziert. Insgesamt werden jedoch 17 Byte beschrieben. Der Fehler wird erkannt und Memcheck beschreibt zusätzlich, wann der Speicher alloziert wurde:

```
Invalid write of size 1
  at 0x8048453: main (bugs.c:17)
Address 0x41a2039 is 1 bytes after a block of size 16 alloc'd
  at 0x4025D2E: malloc (vg_replace_malloc.c:207)
  by 0x804842A: main (bugs.c:12)
```

Der dritte Fehler entsteht bei dem Versuch, String `c1` nach String `c2` zu kopieren. Dabei überlappen sich die diese. Memcheck erkennt den Fehler und beschreibt ihn:

```
Invalid write of size 1
  at 0x40265A3: strncpy (mc_replace_strmem.c:291)
  by 0x80484AD: main (bugs.c:37)
Address 0x41a2078 is 0 bytes after a block of size 16 alloc'd
  at 0x4025D2E: malloc (vg_replace_malloc.c:207)
  by 0x8048461: main (bugs.c:26)

Source and destination overlap in strncpy(0x41A207F, 0x41A2077, 15)
  at 0x402661B: strncpy (mc_replace_strmem.c:291)
  by 0x80484AD: main (bugs.c:37)
```

Cachegrind & Callgrind

Cachegrind ist ein Cache-Profiler. Dazu simuliert Cachegrind die L1- und L2-Caches des Prozessors. Jegliche Speicherzugriffe, die zu Cache-Misses führen, können damit gefunden werden. Cachegrind speichert während der Ausführung alle Analyse-Daten in einer Datei. Mit KCachegrind [3] können so die Ergebnisse der Analyse grafisch dargestellt werden.

Callgrind ist eine Erweiterung zu Cachegrind, mit der Funktionsaufrufgraphen erstellt werden können. Wie Cachegrind liefert auch Callgrind zusätzlich eine Datei mit den Ergebnissen, die mit KCachegrind visualisiert werden kann. Als Beispiel zeigt Abbildung 4.2 den Funktionsaufrufgraphen des L4Env-Startcodes.

Beide Werkzeuge können ohne Anpassungen auf L4 eingesetzt werden.

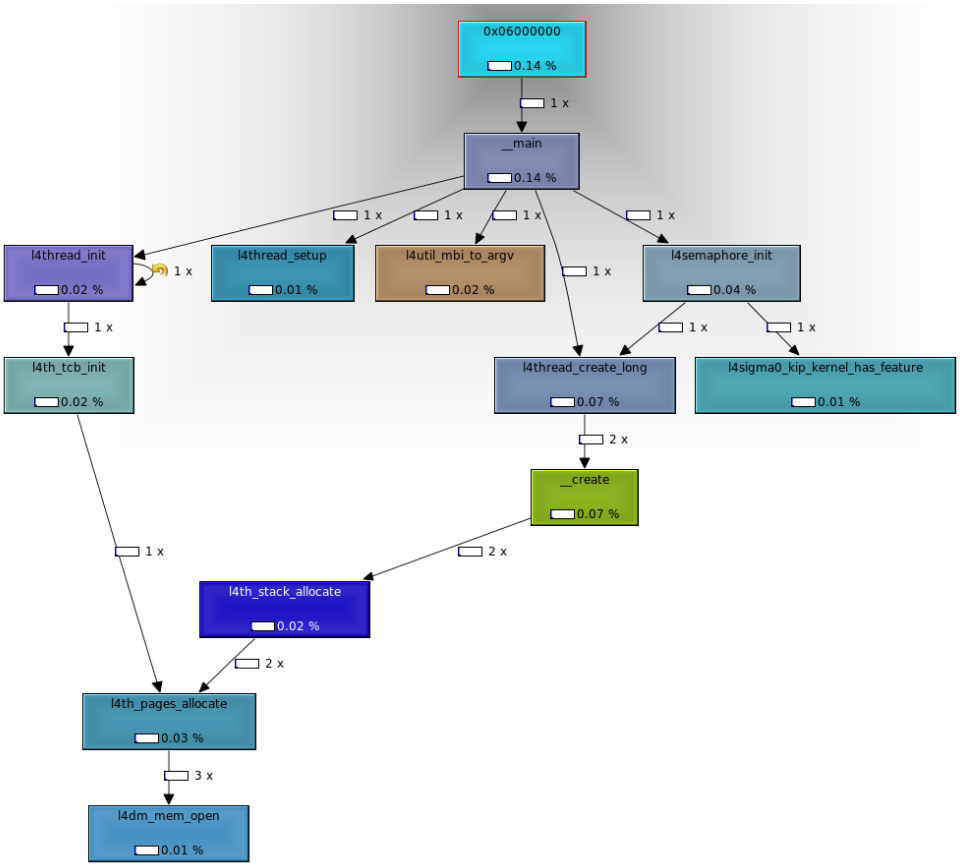

Abbildung 4.2: Aufrufgraph des L4Env-Startcodes

Massif

Massif ist ein Heap-Profiler. Mit ihm kann untersucht werden, welcher Programmteil wieviel Speicherbedarf hat. Dazu macht Massif Snapshots bei jeder Speicherallokation und -freigabe. Das Ergebnis gibt Massif in eine Datei aus. Mit dem zugehörigen Skript `ms_print` können ein Diagramm generiert und die Stellen im Programm, an denen der Speicher alloziert wird, ausgegeben werden. Abbildung 4.3 zeigt den Speicherverbrauch von *tinycc* unter L4. Massif stellt auf der X-Achse die Anzahl der ausgeführten Instruktionen und auf der Y-Achse den verbrauchten Speicher dar.

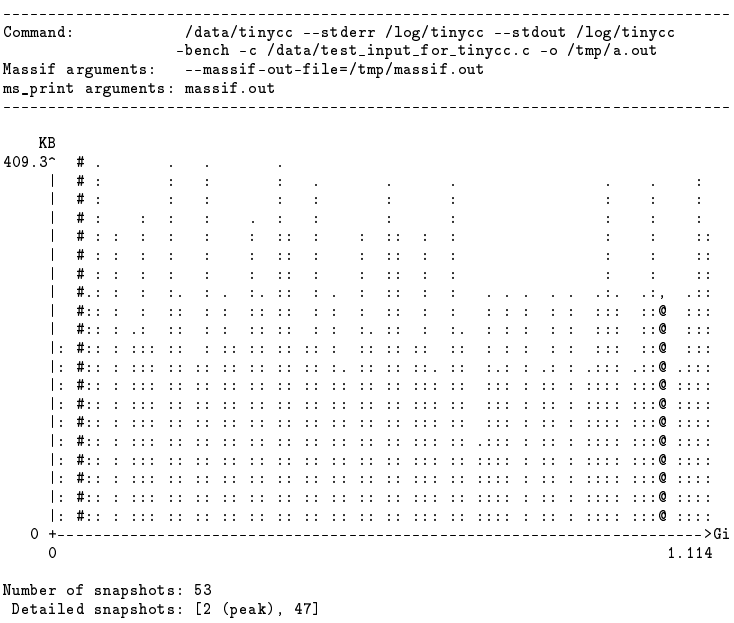

Abbildung 4.3: Speicherverbrauch von *tinycc* auf L4

Wie Memcheck lässt sich auch Massif nutzen, solange keine L4Env spezifische Funktionen zur Speicherallokation benutzt werden.

Helgrind & DRD

Mit den Valgrind-Tools Helgrind und DRD können Programme mit Threads analysiert werden. Helgrind dient zum Finden von Synchronisationsfehlern, während DRD Fehler bei gemeinsamer Speichernutzung durch Threads findet.

Beide Tools setzten für die Analyse POSIX-Threads (pthreads) voraus. Da das L4Env keine Unterstützung für POSIX-Threads bietet, sind beide Tools nicht ohne Modifikationen in L4 einsetzbar.

Lackey & Nulgrind
Lackey ist ein einfaches Valgrind-Tool, das während der Ausführung des Gastes in Valgrind Daten sammelt. Zu diesen Daten gehört beispielsweise die Anzahl der Basic Blocks, die ausführt wurden sowie die Anzahl der ausgeführten Instruktionen.

Nulgrind ist das minimalste Valgrind-Tool. Es führt keine Instrumentierung des Gastes durch. Mit Nulgrind können die Geschwindigkeitsverluste durch die dynamische binäre Analyse ermittelt werden.

Beide Werkzeuge können auf L4 ohne Modifikation eingesetzt werden.

Außer Helgrind und DRD laufen also alle Standard-Valgrind-Tools ohne Anpassungen auf L4. Massif und Memcheck müssten angepasst werden, um auch alle L4Env-spezifischen Funktionen zur Allokation und Freigabe von Speicher überprüfen zu können.

4.3 Performance

Um die Geschwindigkeit meiner Portierung zu bewerten, messe ich die Laufzeit von Gast-Programmen in Valgrind. Ziel ist es, die Geschwindigkeit der Ausführung von Gästen in Valgrind unter L4 und Linux zu vergleichen. Dazu nehme ich eine Portierung der zu Valgrind gehörenden Performance-Test-Suite vor. Mit ihr kann die Geschwindigkeit von Valgrinds Core, der LibVEX und den Tools bewertet werden. Zusätzlich zu dieser portiere ich ein Programm zur Berechnung von MD5-Summen.

Zu Valgrind besitzt zudem eine Regression-Test-Suite. Diese wird aufgrund der Abhängigkeit von POSIX-Funktionen wie `fork()`, `clone()`, POSIX-Threads und Signalen nicht portiert.

Da außerdem die Geschwindigkeit von IPC auf mikrokern-basierten Systemen eine besondere Rolle spielt, implementiere ich zwei kleine IPC-Benchmarks – *thread_ipc* und *task_ipc*. Die meisten Benchmarks eignen sich auch zur Beurteilung der Qualität, da sie bewertbare Ausgaben wie Hash-Werte (*md5sum*) oder ein Binary (*tinycc*) liefern.

Für die Bewertung der Performance meiner Portierung von Valgrind auf L4 habe ich zusammenfassend folgende Testprogramme benutzt:

- ***bigcode*** führt eine Menge Code aus. Es wird für das Vergleichen der Übersetzungsgeschwindigkeiten von Valgrinds JIT-Compiler genutzt.
- ***heap*** alloziert große Mengen Speicher und gibt sie wieder frei. Seine Aufgabe ist das Testen der Speicherverwaltung von Valgrind.

- ***sarp*** alloziert große Objekte auf dem Stack und gibt sie wieder frei.
- ***bz2*** ist ein von Julian Seward (dem Entwickler Valgrinds) entwickeltes, frei verfügbares Programm, das der verlustfreien Kompression dient.
- ***fbench*** führt Raytracing-Algorithmen aus.
- ***ffbench*** berechnet Fast-Fourier-Transformationen. Wie bei *fbench* werden dabei viele Floating-Point-Operationen ausgeführt.
- ***tinycc*** ist ein kleiner und schneller C-Compiler.
- ***md5sum*** führt den „Message-Digest-Algorithm 5“ aus. Mit *md5sum* können Hash-Werte für Dateien oder Zeichenketten erzeugt werden.
- ***thread_ipc + task_ipc*** sind zwei kleine Programme für die Bewertung der IPC-Performance. Zwei Tasks/Threads schicken sich gegenseitig eine bestimmte Anzahl von IPCs zu, um am Ende die Laufzeit einer IPC berechnen zu können. Das Testprogramm *thread_ipc* dient zum Messen der IPC-Performance zwischen Threads innerhalb eines Adressraums. Demgegenüber kann mit *task_ipc* die Geschwindigkeit von IPCs über Adressraumgrenzen hinweg bestimmt werden.

In den Tabellen 4.1, 4.2, 4.3, 4.4 und 4.5 sind die Messergebnisse der einzelnen Testprogramme dargestellt. Für jedes Programm messe ich die Laufzeit auf L4 und Linux mit und ohne Valgrind. Bei der Ausführung in Valgrind berücksichtige ich außerdem, welches Valgrind-Tool den Gast instrumentiert. Zusätzlich zur absoluten Laufzeit ist der Speeddown gegenüber der nativen Ausführung auf der jeweiligen Plattform angegeben. Jede einzelne Messung wird fünf Mal durchgeführt. Die Standardabweichung der gemessenen Werte wird in den genannten Tabellen dargestellt. Abbildung 4.4, 4.5, 4.6, 4.7 und 4.8 zeigen die Speeddowns der einzelnen Valgrind-Tools in L4 und Linux.

Als Nächstes möchte ich Gründe für Geschwindigkeitsunterschiede zwischen Valgrind auf L4 und Linux diskutieren.

Bedingt durch das Multi-Server-OS (DROPS) laufen viele Operationen wie das Lesen oder Schreiben von Dateien oder die Allokation und das Freigeben von Speicher durch zusätzliche Kontextwechsel langsamer ab. Bei monolithischen Systemen muss zum Ausführen solcher Operationen in den Kern und wieder zurück in den Nutzermodus gewechselt werden, also werden dann insgesamt 2 Kontextwechsel benötigt. Bei mikrokern-basierten Systemen muss zu einem Server gewechselt werden, der in einem anderen Kontext im Nutzermodus ausgeführt wird. Da Kontextwechsel nur durch den Kern durchgeführt werden können,

sind für die oben genannten Operationen 4 Kontextwechsel notwendig. Dies wirkt sich besonders bei Memcheck und Callgrind aus.

Die Portierung L4/Valgrind unterstützt mangels vollständiger Implementation im L4Env keine POSIX-Signale (siehe 3.1.3). Dies kann Geschwindigkeitssteigerungen bei der Ausführung von Programmen in L4/Valgrind zu Folge haben.

Durch die interpretierte Ausführung des Gastes und der Serialisierung von Threads in Valgrind entsteht ein Overhead. Beim Vergleich der Speeddowns der Testprogramme *thread_ipc* und *task_ipc* wird dies besonders deutlich. Allgemein ist zu erwarten, dass das Senden und Empfangen von IPCs innerhalb eines Adressraums schneller ist als über Adressraumgrenzen hinweg. Dies wird durch die absoluten Messwerte der nativen Ausführung von *thread_ipc* und *task_ipc* belegt. Werden die Testprogramme jedoch in Valgrind ausgeführt, führt der durch die Serialisierung von Threads und durch die interpretierte Ausführung des Gastes verursachte Overhead dazu, dass die Kommunikation innerhalb eines Adressraums langsamer ist als die über Adressraumgrenzen hinweg.

Zusammengefasst zeigt ein Vergleich der Speeddown-Durchschnitte der einzelnen Tools, dass Valgrind in L4 mehr Overhead produziert als unter Linux. Der Overhead resultiert aus meiner nicht einzig auf Performance basierenden Portierung und den zu erwartenden Geschwindigkeitsverlusten in mikrokern-basierten Systemen. In Abbildung 4.9 sind die Mittelwerte der Speeddowns der einzelnen Valgrind-Tools unter Linux und L4 in Form einer Tabelle und eines Diagramms dargestellt.

Ausführungs-Umgebung	OS	md5sum absolut (in s)	md5sum speed-down	md5sum σ	sarp absolut (in s)	sarp speed-down	sarp σ
nativ	Linux	0,020878		0,000053	0,158114		0,000683
	L4	0,020407		0,000036	0,162271		0,000492
Valgrind/ Nulgrind	Linux	0,125090	5,99	0,000248	1,386197	8,77	0,001673
	L4	0,124462	6,10	0,000288	0,988605	6,09	0,000466
Valgrind/ Memcheck	Linux	0,326405	15,63	0,001025	8,661692	54,78	0,095358
	L4	0,504258	24,71	0,000371	12,988696	80,04	0,000890
Valgrind/ Callgrind	Linux	0,248065	11,88	0,001142	10,956045	69,29	0,174846
	L4	0,293102	14,36	0,000325	14,212325	87,58	0,000803
Valgrind/ Cachegrind	Linux	0,608405	29,14	0,001875	13,382577	84,64	0,058727
	L4	0,785899	38,51	0,003754	13,908240	85,71	0,000565
Valgrind/ Massif	Linux	0,117554	5,63	0,000301	1,549279	9,80	0,002438
	L4	0,204990	10,05	0,000042	0,701126	4,32	0,000479
Valgrind/ Lackey	Linux	0,813959	38,99	0,003838	16,438216	103,96	0,056762
	L4	0,802292	39,32	0,000180	10,815995	66,65	0,000317

Speeddown ist das Verhältnis der Ausführungsgeschwindigkeit des Gastes in Valgrind zu der Ausführungsgeschwindigkeit bei nativer Ausführung. Bei der Berechnung werden die Zeiten der jeweiligen Plattformen zueinander in Verhältnis gesetzt.
σ bezeichnet die Standardabweichung der Messwerte.

Tabelle 4.1: Absolute Ausführungszeiten und Speeddown von *md5sum* und *sarp*

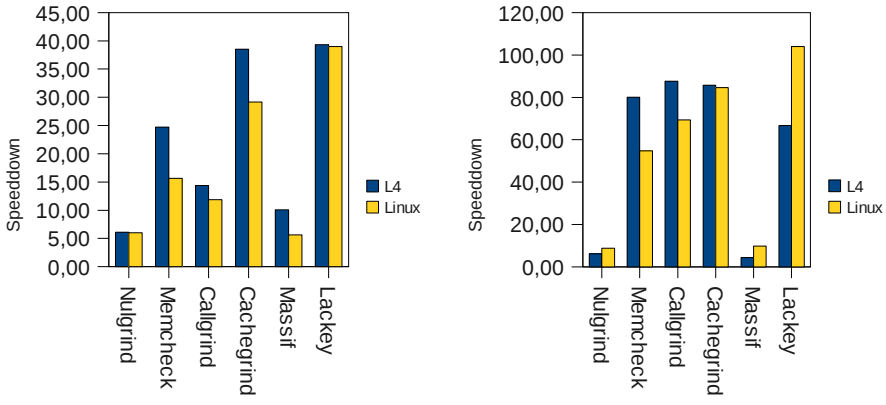

Abbildung 4.4: Speeddown von *md5sum* (links) und *sarp* (rechts)

Ausführungs-Umgebung	OS	bigcode absolut (in s)	bigcode speed-down	bigcode σ	tinycc absolut (in s)	tinycc speed-down	tinycc σ
nativ	Linux	1,422873		0,000990	0,538167		0,013854
	L4	1,485350		0,000327	1,085501		0,000842
Valgrind/ Nulgrind	Linux	8,884175	6,24	0,003240	3,914253	7,27	0,007953
	L4	37,078642	24,96	0,010935	4,593944	4,23	0,000680
Valgrind/ Memcheck	Linux	23,603235	16,59	0,131359	18,054462	33,55	0,084041
	L4	54,302207	36,56	0,007712	81,804047	75,36	0,061867
Valgrind/ Callgrind	Linux	70,777851	49,74	0,054840	32,121045	59,69	0,045854
	L4	137,709932	92,71	0,008231	52,084906	47,98	0,001255
Valgrind/ Cachegrind	Linux	90,048666	63,29	0,285969	36,858018	68,49	0,081796
	L4	145,010640	97,63	0,012097	51,137975	47,11	0,001540
Valgrind/ Massif	Linux	10,480120	7,37	0,004898	5,206044	9,67	0,040083
	L4	39,171893	26,37	0,002589	7,662775	7,06	0,001405
Valgrind/ Lackey	Linux	108,357856	76,15	0,063031	59,423723	110,42	0,108168
	L4	136,311110	91,77	0,003621	124,04885	114,28	0,001601

Tabelle 4.2: Absolute Ausführungszeiten und Speeddown von *bigcode* und *tinycc*

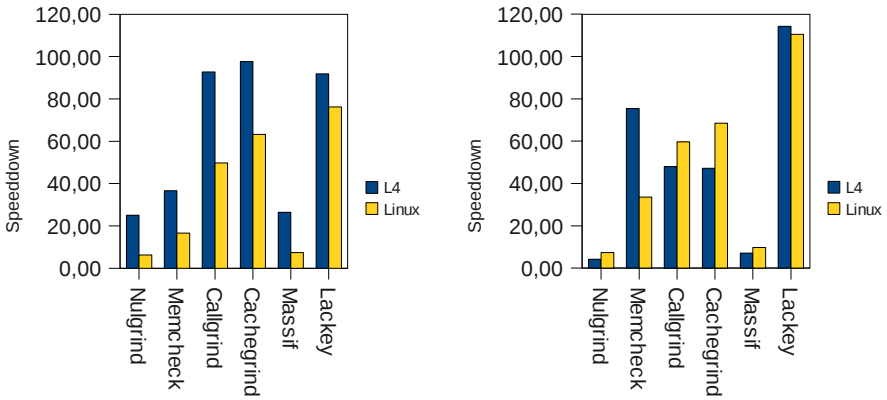

Abbildung 4.5: Speeddown von *bigcode* (links) und *tinycc* (rechts)

Ausführungs-Umgebung	OS	heap absolut (in s)	heap speed-down	heap σ	bz2 absolut (in s)	bz2 speed-down	bz2 σ
nativ	Linux	0,261012		0,012070	0,370418		0,002650
	L4	0,358165		0,001689	0,986811		0,004217
Valgrind/ Nulgrind	Linux	1,565423	6,00	0,048036	1,496535	4,04	0,010765
	L4	2,666474	7,44	0,021380	2,911520	2,95	0,011135
Valgrind/ Memcheck	Linux	9,720407	37,24	0,104030	5,287746	14,28	0,045698
	L4	16,635801	46,45	0,024112	16,532406	16,75	0,010701
Valgrind/ Callgrind	Linux	20,718314	79,38	0,035441	9,351953	25,25	0,046698
	L4	41,759480	116,59	0,026078	29,128020	29,52	0,010769
Valgrind/ Cachegrind	Linux	18,880785	72,34	0,072055	22,052116	59,53	0,019033
	L4	35,344580	98,68	0,020948	75,776531	76,79	0,095402
Valgrind/ Massif	Linux	6,860661	26,28	0,040234	1,649219	4,45	0,012863
	L4	11,029209	30,79	0,011434	4,053913	4,11	0,005164
Valgrind/ Lackey	Linux	34,637121	132,70	0,366217	23,803989	64,26	0,118678
	L4	55,317933	154,45	0,099244	68,562864	69,48	0,096773

Tabelle 4.3: Absolute Ausführungszeiten und Speeddown von *heap* und *bz2*

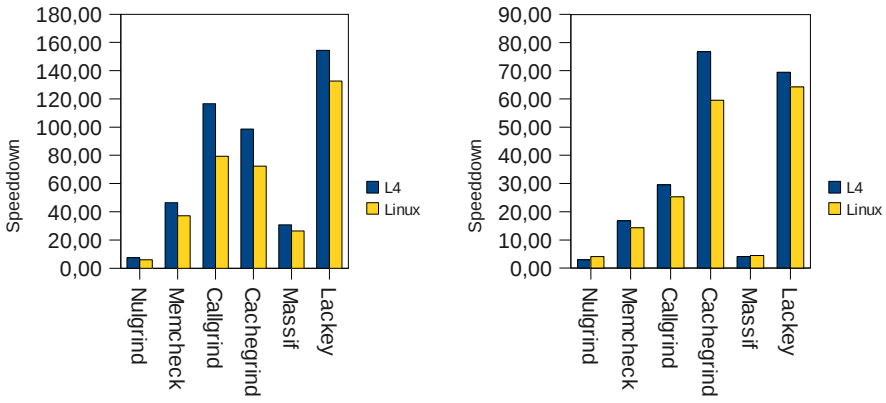

Abbildung 4.6: Speeddown von *heap* (links) und *bz2* (rechts)

Ausführungs-Umgebung	OS	fbench absolut (in s)	fbench speed-down	fbench σ	ffbench absolut (in s)	ffbench speed-down	ffbench σ
nativ	Linux	6,514041		0,015798	0,724180		0,007163
	L4	8,373969		0,024873	0,692319		0,000227
Valgrind/ Nulgrind	Linux	26,408286	4,05	0,189409	2,067543	2,86	0,001497
	L4	55,234870	6,60	0,004825	2,090899	3,02	0,000447
Valgrind/ Memcheck	Linux	85,948925	13,19	0,273270	8,399294	11,60	0,139332
	L4	97,849089	11,68	0,012302	10,077690	14,56	0,002198
Valgrind/ Callgrind	Linux	116,716291	17,92	0,536137	6,670265	9,21	0,099952
	L4	161,832875	19,33	0,016005	9,433415	13,63	0,005423
Valgrind/ Cachegrind	Linux	201,291373	30,90	0,683056	45,724534	63,14	0,099293
	L4	286,906538	34,26	0,069892	63,289816	91,42	0,002270
Valgrind/ Massif	Linux	27,320280	4,19	0,257252	2,099264	2,90	0,003758
	L4	26,122159	3,12	0,001774	2,223904	3,21	0,000406
Valgrind/ Lackey	Linux	384,658861	59,05	2,146621	33,726396	46,57	0,362043
	L4	391,093999	46,70	0,008230	35,885793	51,83	0,000704

Tabelle 4.4: Absolute Ausführungszeiten und Speeddown von *fbench* und *ffbench*

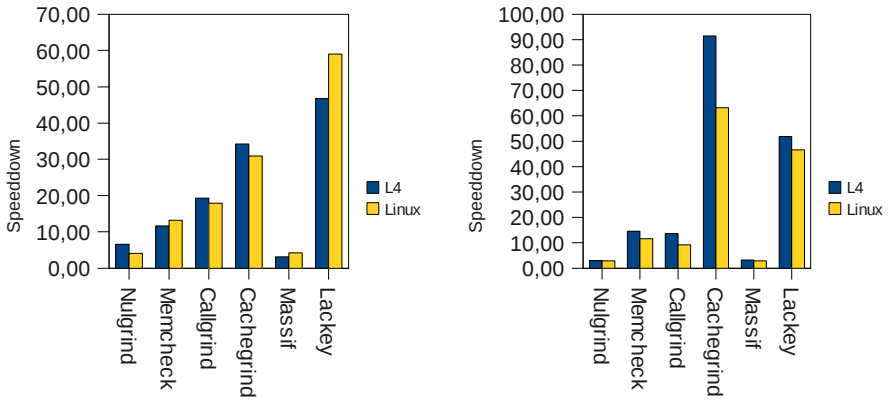

Abbildung 4.7: Speeddown von *fbench* (links) und *ffbench* (rechts)

Ausführungs-Umgebung	task_ipc absolut (in s)	task_ipc speed-down	task_ipc σ	thread_ipc absolut (in s)	thread_ipc speed-down	thread_ipc σ
nativ	0,099310	1,00	0,007074	0,078684	1,00	0,000370
Valgrind/ Nulgrind	0,205182	2,07	0,020509	0,266674	3,39	0,001760
Valgrind/ Memcheck	0,268694	2,71	0,000371	0,395482	5,03	0,001566
Valgrind/ Callgrind	0,348677	3,51	0,006562	0,546231	6,94	0,000260
Valgrind/ Cachegrind	0,376359	3,79	0,000436	0,611359	7,77	0,000853
Valgrind/ Massif	0,242669	2,44	0,043065	0,268454	3,41	0,000415
Valgrind/ Lackey	0,374199	3,77	0,022782	0,587072	7,46	0,010988

Tabelle 4.5: Absolute Ausführungszeiten und Speeddown von *task_ipc* und *thread_ipc*

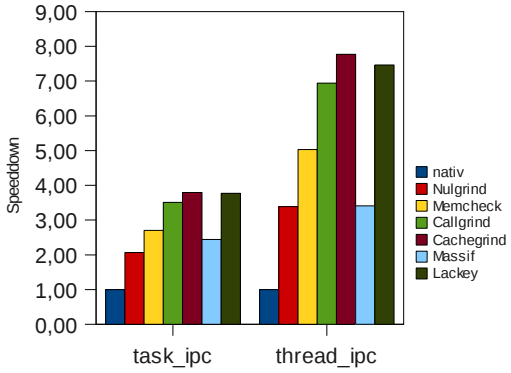

Abbildung 4.8: Speeddown von *task_ipc* (links) und *thread_ipc* (rechts)

Valgrind	Linux	L4
Nulgrind	5,65	7,67
Memcheck	24,61	38,26
Callgrind	40,29	52,71
Cachegrind	58,93	71,26
Massif	8,79	11,13
Lackey	79,01	79,31

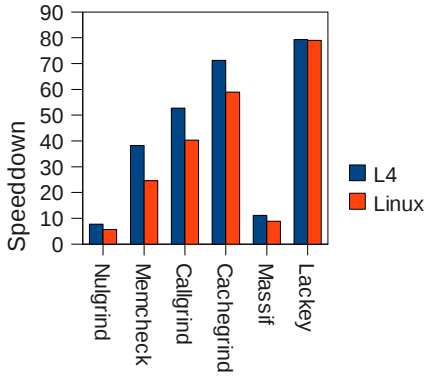

Abbildung 4.9: Durchschnitt der Speeddowns der einzelnen Tools in Linux und L4

5 Zusammenfassung

5.1 Ausblick

Die mir gestellte Aufgabe ist zwar mit vorliegender Arbeit abgeschlossen. Es hat sich aber gezeigt, dass weiterführende Fragen einer Lösung bedürfen. Dazu bedarf es einer Weiterentwicklung von L4/Valgrind mit entsprechenden Tests. Außerdem können mit L4/Valgrind bestehende L4Env-Anwendungen auf Fehler überprüft oder ihr Verhalten während der Laufzeit analysiert werden. Dazu könnten neue Tools, basierend auf dem Framework Valgrind, für L4 entwickelt werden. Konkret ist die Fortsetzung der Arbeit in folgenden Punkten möglich:

Um das Problem der Synchronisation von Region-Mapper und Valgrinds Speicherverwaltungssystem (siehe Kapitel 3.2.2) zu lösen, könnte die Funktionalität des Region-Mappers in Valgrind integriert werden. Valgrind würde dadurch absolute Kontrolle über den Aufbau des Adressraums erlangen.

Wie im vorherigen Kapitel beschrieben, können nicht alle Standard-Tools von Valgrind im L4Env eingesetzt werden. Helgrind und DRD sind Tools zur Analyse von Anwendungen, die POSIX-Threads nutzen. Damit erstere eingesetzt werden können, müssen sie so angepasst werden, dass auch die Untersuchung nativer L4-Threads möglich wird. Alternativ könnte eine POSIX-Thread-Bibliothek für das L4Env entwickelt werden.

Neben den Standard-Tools von Valgrind gibt es viele weitere. Diese könnten hinsichtlich ihrer Eignung für mikrokern-basierte Systeme untersucht und gegebenenfalls portiert werden. Außerdem ist die Entwicklung von Tools für Probleme, die auf mikrokern-basierten Systemen zu Fehlern führen, denkbar. Dazu sind Modelle und Beschreibungen für typische Fehler notwendig.

L4Linux ist eine para-virtualisierte Version von Linux ([11], [15]). Aufbauend auf meiner Arbeit wäre die Analyse von L4Linux in Valgrind möglich. Dazu müssten die existierenden Valgrind-Tools auf ihre Fähigkeit der Analyse para-virtualisierter Betriebssysteme untersucht werden.

Meine Portierung beschränkt sich auf L4.Fiasco für die x86-Architektur. Um auch die x86/64-Architektur zu unterstützen, sind vor allem Anpassungen der Systemaufruf-Behandlung in L4/Valgrind nötig.

Das *L4RE* (*L4 Runtime Enviroment*) [16] ist der Nachfolger des L4Envs. In einer weiteren Arbeit könnte Valgrind auf L4RE portiert werden. Damit wäre die Analyse von L4RE-Anwendungen möglich. Da das L4RE POSIX-Threads unterstützt, können die Valgrind-Tools Helgrind und DRD zur Fehlersuche in Anwendungen mit Threads genutzt werden.

5.2 Abschließende Worte

> „Program testing can be used to show the presence of bugs, but never to show their absence! “ (Edsger Dijkstra)

Das Ziel dieser Arbeit war es, das Framework Valgrind auf den L4.Fiasco-Mikrokern und das L4-Environment zu portieren. Zusätzlich dazu habe ich Valgrinds Standard-Tools Memcheck, Callgrind, Cachegrind, Massif und Lacky portiert, ohne große Änderungen vornehmen zu müssen. Helgrind und DRD können mangels einer Bibliothek für POSIX-Threads im L4Env nicht eingesetzt werden.

Dennoch gibt es viele Möglichkeiten, die Arbeit weiterzuführen und zu verbessern. Mikrokerne werden sich in Zukunft weiter verbreiten. Die steigende Komplexität von Software fordert Methoden zur automatischen Fehlersuche.

Während meiner Arbeit und bei späteren Tests wurden Fehler im L4Env entdeckt und behoben. Außerdem waren Erweiterungen des L4Env für Valgrind notwendig.

Als Ergebnis dieser Arbeit kann Valgrind zusammen mit den Tools als Werkzeug zur Unterstützung der Entwicklung von L4Env-basierten Anwendungen eingesetzt werden. Für Fehler, die nicht oder nur teilweise mit den existierenden Tools gefunden werden können, bildet das Framework Valgrind eine gute Grundlage zur Entwicklung neuer Tools.

Glossar

ABI	*Application Binary Interface*, Schnittstelle auf Maschinenebene zwischen einem Programm und dem Betriebssystem	11
API	*Application Programming Interface*, Programmierschnittstelle zur Anbindung anderer Software an das System	7
Basic Blocks	Maschinencode-Sequenzen, die an Funktions-Einsprungpunkten beginnen und mit einem den Kontrollfluss beeinflussenden Befehl enden.	17
Binary	ausführbare Datei, besteht aus Maschinen-Instruktionen oder Bytecode	7
Dataspace	Container für unterschiedliche Arten von Speicher	36
Flexpages	beschreiben Bereiche des virtuellen Adressraumes in L4	34
Framework	Skelett einer Anwendung, welches angepasst werden kann.	15, 22
IPC	*Inter Process Communication*, Kommunikation zwischen zwei Tasks und Threads zur Datenübertragung	32, 34, 44
L4	L4 ist eine Familie von Mikrokernen der zweiten Generation, ursprünglich entwickelt und implementiert von Jochen Liedtke [17].	11, 34

L4Env	Das L4Env ist eine Umgebung zur Entwicklung von Programmen basierend auf L4-Mikrokernen.	13
L4VFS	*L4 virtual file system layer*, stellt POSIX-Funktionalität für L4Env-Anwendungen zur Verfügung.	32
LibVEX	Eine Bibliothek für dynamisch binäre Übersetzung und Instrumentierung	15, 16
Region-Mapper	Dient der Verwaltung des virtuellen Adressraumes einer L4Env Task und ist verantwortlich für das Auflösen von Pagefaults	36
RISC	*Reduced Instruction Set Computing*, Computerarchitektur mit reduziertem Befehlssatz	4
UCode	Der Maschinencode des Gastes wird durch die LibVEX in einen Bytecode, genauer in UCode, übersetzt.	17, 23
Valgrinds Core	Kern des Frameworks Valgrind; steuert die Ausführung des Gastes	15

Literaturverzeichnis

[1] Parasoft Insure++ enables fast, reliable detection and resolution of elusive runtime memory errors. http://www.parasoft.com/jsp/products/home.jsp?product=Insure&itemId=63. 10

[2] IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, Order Number 253668, 2004. 44

[3] KCachegrind. http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindIndex, 2009. 54

[4] OpenWorks: Software Design and Engineering. http://www.open-works.co.uk/, 2009. 17

[5] Parasoft delivers quality as a continuous process. http://www.parasoft.com, 2009. 10

[6] uClibc – a C library for embedded Linux. http://www.uclibc.org/, 2009. 32

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. 8

[8] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000. 9

[9] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991. 9

[10] Michael Hohmuth. The fiasco kernel: Requirements definition. Technical report, Technische Universität Dresden, 1998. 44

[11] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998. `http://os.inf.tu-dresden.de/papers_ps/part98.ps`. 65

[12] Michiel Ronsse Jonas Maebe and Koen De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *In Proc. 4th Workshop on Binary Translation (WBT'02)*, 2002. 8

[13] S. Karthik and H. G. Jayakumar. Static Analysis: C Code Error Checking for Reliable and Secure Programming. In *IEC (Prague)*, pages 434–439, 2005. 1

[14] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005. 7

[15] Adam Lackorzynski. L4Linux on L4Env. Großer Beleg, Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Professur Betriebssysteme, 2002. `http://os.inf.tu-dresden.de/papers_ps/adam-beleg.pdf`. 65

[16] Adam Lackorzynski and Alexander Warg. Taming Subsystems - Capabilities as Universal Resource Access Control in L4. In *Workshop on Isolation and Integration in Embedded Systems (IIES)*, 2009. 66

[17] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM. 11, 12, 67

[18] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996. 35

[19] Nicholas Nethercote. Bounds-checking entire programs without recompiling. In *In SPACE 2004*, 2004. 1

[20] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004. 3, 4, 12, 19

[21] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003. 5, 15, 16

[22] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA, June 2007. 15, 18

[23] Operating Systems Research Group. - L4 Env - An Environment for L4 Applications, 2003. 13

[24] Lars Reuther. L4 Region Mapper Reference Manual. `http://os.inf.tu-dresden.de/l4env/doc/html/l4rm/index.html`. 36

[25] Anand Gaurav Satish Chandra Gupta. Advanced features of IBM Rational Purify: Debugging with Purify. `http://www.ibm.com/developerworks/rational/library/08/0205_gupta-gaurav`, 2008. 9

[26] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association. 52