Anwendungsgewahre statische Spezialisierung vormals dynamischer Systemaufrufe zur Verbesserung nichtfunktionaler Eigenschaften eingebetteter Echtzeitsysteme

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von Herrn
Björn Fiedler, M.Sc.
geboren am 13. Mai 1992
in Hannover, Deutschland

2023

1. Referent	**Prof. Dr.-Ing. habil. Daniel Lohmann**
2. Referent	**Prof. Dr. Michael Rohs**
Tag der Promotion	**08.06.2023**

Abstract

Embedded systems are an omnipresent part of our daily life. They are ubiquitously present in almost every moment to support and secure our activity. At the same time, we expect these systems to be both highly cost-efficient in development and production. Without restrictions, we expect them to work reliably and always respond timely and as expected. This leads to an immense pressure on the development process of new systems, especially with the large number of units and the further increasing occurrence of these systems.

A finished system has a defined task according to its environment and thus a defined software application that it executes. For the tools used to implement and execute it, this is not the case. Those tools are not specifically designed for that exact task, but for a variety of possible applications. They provide a wide range of functionality and flexibility, and, hence, allow a wide usage spectrum. In this thesis, I focus on the real-time operating system which serves as the base software layer to execute the designated application. Such an operating system provides a wide range of abstractions, system object classes, and associated interaction methods, of which an actual application implementation uses only a subset. Using dynamically configured systems, which I consider in this thesis, enforce to instantiate and configure all system objects and their interworking during runtime by code, exclusively. As a result, an operating system needs to be able to accept every system call at every point in time, even if not issued by the actual application. This flexibility causes pessimistic assumptions for possibly never appearing interaction patterns and forces dynamical management of system objects and state.

To solve this problem when not needed by the application, I present methods to systematically and automatically specialize formerly dynamic system calls statically. While considering the requirements of a given application, these methods improve the non-functional properties of the resulting specialized system without changing the functional properties. Using static analyses, I determine the system objects forming the application structure and their possible interactions. Backed by this knowledge, I apply static specializations on both, the startup and the working phase of the application during compile time. At the startup phase, I apply static system-object instantiation by transforming the effects of the system calls into compile-time constants. To improve the working phase, I statically exchange the generic implementations of system objects and their interaction system calls by a version suited for the actual usage patterns.

With these specializations, I am able to reduce both runtime and memory requirements of a specialized system. I can speed up system startup by up to 67 %. During working phase, an execution time reduction by up to 43 % for a single system call is achievable.

With this thesis, I show that an automatic application-aware static specialization of formerly dynamic system calls is both feasible and beneficial. I am able to precompute the effects of dynamic system calls during compile time, and, thus, reduce both run-time overhead and memory requirements, while removing unused system-call implementations. By using system call implementations specialized to the actual application, I reduce superfluous administrative overhead, and, hence, reduce runtime overhead even further. This specialization takes place without any disturbance for application parts, truly relying on the dynamic operating system interface, as all specializations are applied in a non-breaking manner. This results in a continuous transition between dynamically and statically configured systems, improving the system performance by only removing superfluous flexibility without ever violating functional properties.

Keywords — real-time operating system, whole-system optimization, static system specialization, dynamic system model, application specific instance and interaction optimization

Kurzfassung

Eingebettete Systeme sind aus unserem heutigen Leben nicht mehr wegzudenken. Sie sind allgegenwärtig in fast jedem Moment unseres täglichen Lebens um uns vorhanden und unterstützen unseren Alltag. Wir erwarten von diesen Systemen gleichzeitig sowohl hohe Kosteneffizienz in Entwicklung als auch Produktion. Gleichzeitig erwarten wir, dass diese zuverlässig arbeiten und stets erwartungsgemäß reagieren. Dies führt gerade bei der großen Stückzahl und dem weiter steigenden Vorkommen dieser Systeme zu einem immensen Druck auf den Entwicklungsprozess neuer Systeme.

Während ein fertiges System entsprechend der Umgebung eine festgelegte Aufgabe und damit eine festgelegte Software-Anwendung hat, die es ausführt, sind die für dessen Implementierung und Ausführung verwendeten Werkzeuge nicht speziell für genau diese Aufgabe gedacht, sondern für eine Vielzahl möglicher Anwendungen. Dies bedeutet, dass sie einen deutlich größeren Funktionsumfang und eine größere Flexibilität in der Verwendung dessen ermöglichen, als von der konkreten Anwendung benötigt wird. In dieser Arbeit beschäftige ich mich mit den Echtzeitbetriebssystemen (EZBS), die als Ausführungsgrundlage dienen. Diese stellen ein breites Spektrum an Primitiven verschiedener Systemobjektklassen dazugehöriger Interaktionsmethoden zur Verfügung, von denen eine Anwendung nur eine Teilmenge verwendet. Bei den hier betrachteten dynamisch konfigurierten Systemen werden alle Systemobjekte zur Laufzeit konfiguriert und auch ihre Interaktionen sind ausschließlich durch den Verlauf des Programmcodes bestimmt. Ein Betriebssystem muss dementsprechend jederzeit beliebige Systemaufrufe akzeptieren können, auch wenn diese von der Anwendung nicht ausgeführt werden. Diese Freiheit verursacht pessimistische Annahmen für mögliche Interaktionsmuster und erzwingt eine dynamische Verwaltung aller Systemzustände und Systemobjekte.

In dieser Arbeit stelle ich daher Verfahren vor, mit denen systematisch und automatisiert vormals dynamische Systemaufrufe unter Beachtung der Anforderungen einer gegebenen Anwendung statisch spezialisiert werden können, sodass sich insgesamt die nichtfunktionalen Eigenschaften des Gesamtsystems verbessern. Mittels statischer Analyse ermittle ich die von der Anwendung verwendeten Systemobjekte und deren mögliche Interaktionen. Mit diesem Wissen führe ich in Spezialisierungen in der Phase des Systemstarts und in der Arbeitsphase des Systems zur Übersetzungszeit durch. Der Systemstart optimiere ich, indem semantisch statische Systemobjekte bereits zur Übersetzungszeit instanziiert werden. Interaktionen während der Arbeitsphase optimiere ich, indem ich auf die tatsächlichen Verwendungsmuster spezialisierte Implementierungen von Systemobjekten und deren Interaktionen einsetze.

Mit diesen Spezialisierungen bin ich in der Lage, sowohl Laufzeit als auch Speicherbedarf eines spezialisierten Systems zu reduzieren. Den Systemstart kann ich um bis zu 67 % beschleunigen. Bei der Ausführungszeit eines einzelnen Systemaufrufs zur Kommunikation zweier Systemobjekte sind bis zu 43 % Reduktion möglich.

Als Ergebnis dieser Arbeit kann ich zeigen, dass eine automatische anwendungsgewahre statische Spezialisierung von vormals dynamischen Systemaufrufen gewinnbringend möglich ist. Dabei kann ich das Ergebnis von Systemaufrufen zur Laufzeit vorausberechnen und damit sowohl die sonst benötigte Laufzeit reduzieren, als auch eventuell nicht mehr benötigte Systemaufrufimplementierungen im Betriebssystem einsparen. Durch den Einsatz von anwendungsangepassten Implementierungen von Systemaufrufen ist eine weitere Verbesserung gegeben. Dies ist in einem fließenden Übergang möglich, sodass diejenigen Komponenten, die die Flexibilität der dynamischen Betriebssystemschnittstelle benötigen, diese weiterhin uneingeschränkt zur Verfügung steht. Die funktionalen Eigenschaften und Anforderungen werden dabei unter keinen Umständen verletzt.

Schlüsselwörter — Echtzeitbetriebssystem, ganzheitliche Systemoptimierung, statische Systemspezialisierung, dynamisches Systemmodell, anwendungsgewahre Instanz- und Interaktionsoptimierung

Danksagungen

An dieser Stelle möchte ich einen großen Dank an alle die Menschen aussprechen, die mich auf dem Weg zu meiner Promotion begleitet, unterstützt und geleitet haben. Ohne euch wäre ich vermutlich nicht am Ziel angekommen.

Da ich mein Fokus in dieser Arbeit auf Echtzeitsystemen liegt und zu jedem Ergebnis auch ein Ursprung gehört, werde ich in chronologischer Reihenfolge entsprechend dem Erstkontakt mit meiner Lebenszeitachse vorgehen.

An erster Stelle stehen dementsprechend meine Eltern Birgit und Bernd. Goethe hat einmal gesagt:„Zwei Dinge sollen Kinder von ihren Eltern bekommen: Wurzeln und Flügel.“ Ich hätte mir keine bessere Umsetzung dessen vorstellen können als das mir dargebotene Umfeld, um mit Neugier, Interesse und grenzenlosem Rückhalt die Welt zu entdecken.

Für Motivation und Zuspruch, wann immer ich sie brauchte, möchte ich meiner Partnerin Miri danken, die mich nun schon seit vielen Jahren begleitet und stets an meiner Seite mit mir gemeinsam durch das Leben geht.

Als nächster hat Jürgen die Bühne betreten, mich praktisch in Betriebssysteme eingeführt und auch gleich an das SRA gebunden, noch bevor ich so richtig mit dem Studium begann. So sollte es geschehen, dass ich regelmäßig Tutor in Jürgens Lehrveranstaltung wurde.

Während des Studiums lehrten mich Michael, Markus, Max, Tim, Beren und Max am HCI die Herausforderungen von Wissenschaft, gaben mir Einblicke in ihre Forschung und ermöglichten mir zwei spannende Abschlussarbeiten. Vielen Dank für das Vertrauen und die Freiheiten, die ich bei euch genießen durfte.

Dann jedoch kam Daniel neu an die LUH, richtete das SRA neu aus und ließ mich gemeinsam mit Christian tief in den Bau von Betriebssystemen einsteigen. Auch wenn ich auf dem Brocken für ein kurzes Schlucken und einen Kloß im Hals sorgte, wurde ich teil eines wunderbaren Teams am SRA und (nach Sitzplätzen sortiert) Jürgen, Monika, Daniel, Lars, Florian, Oskar, Dominik, Tim, Romeo, Stefan, Alexander, Lars, Gerion, Tobias und Christian meine lieben Kollegen und Wegbegleiter. Mit euch durfte ich zahlreichen inspirierenden und kontroversen Fachdiskussionen führen. Gleichzeitig war aber auch Raum, um mit Flo die Wickingerschuld zu erörtern und Stefan als mein Lieblingszellengenosse hatte stets die passenden Worte parat, wenn ich an den Anfragen der Studierenden aus der GBS zu verzweifeln drohte. Ich hatte große Freude daran, mit Gerion dem Vogel das Fliegen beizubringen, auch wenn es schon wieder ein Copter geworden ist. Und Chris, ohne deine Ratschläge zu Schafen und halben Stunden Vorsprung wäre mein Leben als Doktorand nur halb so erbauend gewesen. Vielen Dank lieber Daniel, dass du mir stets das Ziel aufgezeigt und gleichzeitig die Freiheit gegeben hast, auf meinem eigenen Weg dort hinzugelangen. Mit euch habe ich mich stets benommen, wir waren aber nie die langweiligsten! Ich freue mich, Teil dieses bunten Teams zu sein und dies auch noch eine Weile als euer Hausmeister bleiben zu dürfen.

Ebenso gilt mein Dank auch den zahlreichen Studierenden, die an mit mir diskutiert und an meiner Forschung mitgewirkt haben und so zu zahlreichen Erkenntnissen Beiträge geleistet haben. Vielen Dank an Andreas, Barbara, Bastian, Benedikt, Christoph, Dana, David, Domenik, Fredo, Gabriel, Helge, Illia, Jan, Jannis, Jasper, Jonas, Kenny, Lukas, Malte, Malvin, Niklas, Nils, Nishal, Paul, Ralf, Til, Tino, Vitali und Yannick.

Zuletzt, jedoch nicht weniger wichtig, ist meine Tochter Levke Teil meines Lebens geworden. Sie besitzt die großartige Gabe, mich mit Ihrem unwiderstehlichen Lächeln zugleich zu Höchstform zu motivieren und vollständig vom Thema abzulenken. Vielen Dank, dass du mir jeden Tag erneut die gute Laune und dein bedingungsloses Vertrauen entgegenbringst.

Hannover, Juni 2023

Inhalt

1 Einleitung – Motivation, Kontext und Ziel **1**
1.1 Motivation . . . 3
1.2 Kontext der Arbeit . . . 4
1.3 Wissenschaftlicher Rahmen dieser Arbeit . . . 7
1.4 Ziel dieser Arbeit . . . 8
1.5 Aufbau der Arbeit . . . 9
1.6 Typografische Konventionen . . . 10

2 Grundlagen – Begriffe und verwandte Arbeiten **11**
2.1 Das eingebettete Echtzeitsystem als Spezialzwecksystem . . . 13
2.2 Spezialisierung: Definition, Chancen und Einschränkungen . . . 15
2.2.1 Betriebssystemmodelle und deren Programmierschnittstellen . . . 16
2.2.1.1 Programmiermodell . . . 16
2.2.1.2 Konfigurierbarkeit . . . 20
2.3 Stand der Wissenschaft . . . 22
2.3.1 Optimierende Übersetzer: Verlagerung von Laufzeitberechnungen in die Übersetzungszeit . . . 23
2.3.2 Beschleunigen des Systemstarts . . . 24
2.3.3 Anwendungswissen ermitteln und Betriebssystem spezialisieren . . . 24
2.4 Zusammenfassung . . . 27

3 Voraussetzungen – Ermittlung des notwendigen Anwendungswissens **29**
3.1 Welches Wissen wird benötigt und wie wird es gewonnen? . . . 31
3.2 Die statische Instanzanalyse (SIA) . . . 33
3.2.1 Von Kontrollflussgraphen und Werteflussgraphen . . . 33
3.2.2 Das Betriebssystemmodell . . . 35
3.2.3 Ablauf der *statische Instanzanalyse (SIA)* . . . 36
3.3 Die Interaktionsanalyse (INA) . . . 38
3.3.1 Ablauf der INA . . . 38
3.4 Ergebnisse aus SIA und *statische Interaktionsanalyse (INA)* . . . 39
3.5 Einschränkungen der Analysen . . . 39
3.6 Zusammenfassung . . . 41

4 Systemstart – Wiederholte Abläufe ohne Wiederholung ausführen **43**
4.1 Problemfeld und Zielsetzung . . . 45
4.1.1 Problemfeld . . . 45
4.1.2 Ziel . . . 46
4.2 Benötigtes Wissen . . . 47
4.2.1 Phasen eines instanziierenden Systemaufrufs . . . 47
4.2.2 Klassifizierung möglicher Spezialisierungen . . . 49
4.3 Umsetzung der Spezialisierungen . . . 51
4.3.1 Auswahl der Spezialisierung . . . 52
4.3.2 Speicherallokation . . . 53
4.3.3 Initialisierung . . . 53
4.3.4 Registrierung im Betriebssystem . . . 54

4.3.5 Seiteneffekte und Rückgabewert . . . 54
4.3.6 Rekonfiguration des Betriebssystems . . . 56
4.3.7 Bindezeitoptimierungen und die richtige Darstellung von initialisierten Datenstrukturen . . . 56
4.3.8 Kosten der Spezialisierung . . . 57
4.3.9 Aktualisierbarkeit der Anwendung . . . 58
4.4 Evaluation . . . 58
4.4.1 Mikrobenchmarks . . . 60
4.4.1.1 Warteschlangen-Instanziierung . . . 60
4.4.1.2 Faden-Instanziierung vor der Übergabe der Kontrolle an das Betriebssystem . . . 63
4.4.1.3 Faden-Instanziierung nach dem Eintritt in den Mehraufgabenbetrieb 63
4.4.2 Fallbeispiele mit Realweltanwendungen . . . 64
4.4.2.1 LibrePilot CopterControl . . . 64
4.4.2.2 GPSLogger . . . 66
4.5 Zusammenfassung . . . 68

5 Interaktionen – Bei bekannten Interaktionen weniger interagieren 71
5.1 Spezialisierungspotenzial und nicht genutzte Flexibilität generischer dynamischer Systemobjekte . . . 73
5.1.1 Spezialisierungspotenzial von Synchronisations- und Kommunikationsobjekten in FreeRTOS . . . 74
5.1.2 Laufzeitgewinne durch Einschränkung der zugelassenen Aktivitätsträger . . . 75
5.2 Selektion spezialisierbarer Systemobjekte und geeigneter Spezialisierungen . . . 76
5.2.1 Auswahl einer validen Spezialisierung . . . 77
5.3 Evaluation . . . 80
5.3.1 GPSLogger . . . 81
5.3.2 LibrePilot . . . 82
5.4 Zusammenfassung . . . 83

6 Idiomatik und Semantik – Hindernisse und Einflussnahme auf die Spezialisierbarkeit 85
6.1 Fallbeispiele . . . 87
6.1.1 LibrePilot . . . 87
6.1.2 GPSLogger . . . 87
6.2 Die Ebenen der Problemursachen . . . 90
6.3 Design der Betriebssystemschnittstelle . . . 90
6.3.1 Beliebigkeit der Instanziierung . . . 90
6.3.2 Beliebigkeit der Interaktionspartner . . . 90
6.3.3 Pessimistische Grundannahmen . . . 91
6.3.4 Einfachheit der Schnittstelle . . . 92
6.4 Verwendung des Betriebssystems . . . 93
6.4.1 Wahl des Betriebssystems . . . 93
6.4.2 Pseudodynamische Entscheidungen . . . 93
6.4.3 Echte dynamische Entscheidungen . . . 94
6.5 Übersetzer und Werkzeuge . . . 94
6.5.1 Wahl der richtigen Sprachebene . . . 94
6.5.2 Typen und Auszeichnungen in LLVM . . . 95
6.6 Idiomatik . . . 96

6.6.1 Manueller Nachbau von fehlender Sprachsemantik 96
6.6.2 Vorauseilende Änderungsvorsorge 97
6.7 Programmiersprache 98
6.7.1 Garantien und Auszeichnungskraft 98
6.7.2 Unterspezifizierte Semantik 99
6.7.3 Entscheidung der Programmiersprache 100
6.8 Zusammenfassung 100

7 Schluss – Zusammenfassung, Ergebnisse und Ausblick 103
7.1 Zusammenfassung der Arbeit 105
7.2 Forschungsfragen 106
7.3 Ausblick und offene Fragestellungen 108
7.4 Fazit 109

Verzeichnisse 111
Abkürzungsverzeichnis 111
Literatur 113
Abbildungsverzeichnis 125
Tabellenverzeichnis 127
Quellcodeverzeichnis 129
Lebenslauf 131

1

Einleitung

Motivation, Kontext und Ziel

1.1 Motivation

Rechenmaschinen als Vorgänger der heutigen Computer waren Spezialzwecksysteme, deren Algorithmen für den angedachten Einsatzzweck fest verdrahtet und verlötet wurden. Mit dem Aufkommen der speicherprogrammierbaren Turing-mächtigen Allzweckmaschinen begannen diese die bisherigen Systeme zu ersetzen. Die Flexibilität der Software erlaubt wechselnde Einsatzzwecke, sodass dieselbe Maschine wahlweise für wissenschaftliche Forschungsberechnungen oder auch für die Finanzbuchhaltung verwendet werden kann [Rou81]. Damit verlagert sich die Problemlösung von der Hardware in die Software. Die Programme wurden zunächst problemspezifisch handgeschrieben, um jeweils den benötigten Funktionsumfang zu bieten. Nach den Ingenieursprinzipien der Modularität und Wiederverwendbarkeit hat die Verwendung von Softwarebibliotheken und Betriebssystemen Einzug in den Entwicklungsprozess eingebetteter Steuergeräte gehalten. Dies erlaubt ein gewisses Maß an Portabilität und beschleunigt auch den Entwicklungsprozess, da gleichbleibende Grundbausteine nicht jedes Mal neu implementiert werden müssen. Gleichzeitig bedeutet dies jedoch immer eine Übererfüllung der Anforderungen, da nicht alle gebotenen Funktionalitäten benötigt werden. Parallel dazu gibt es stets die Bestrebung hin zu anwendungsangepassten Minimallösungen, die ihre Vorteile in verbesserten nichtfunktionalen Eigenschaften haben [HFC76; PMI88; EKO95; Die19].

Die Systemsoftware und insbesondere das Betriebssystem erfüllen dabei keinerlei Funktionen zum Selbstzweck. Vielmehr sind sie Dienstleister im Auftrag der auszuführenden Anwendung, um eine geeignete Ausführungsumgebung bereitzustellen [Loh14; SGG12]. Demnach hat ein die (funktionalen) Anforderungen der Anwendung übererfüllendes Betriebssystem keinen Mehrwert für diese. Vielmehr ist es so, dass ein ideales Betriebssystem eines eingebetteten Spezialzwecks genau den Funktionsumfang erfüllt, der von der Anwendung gefordert wird und nicht mehr [Lam83]. Ein erhöhter Funktionsumfang bedeutet nichtfunktionale Kosten im Sinne eines erhöhten Speicherbedarfs oder Laufzeitaufwands. Ist es notwendig, dass ein eingebettetes Spezialzwecksystem in der

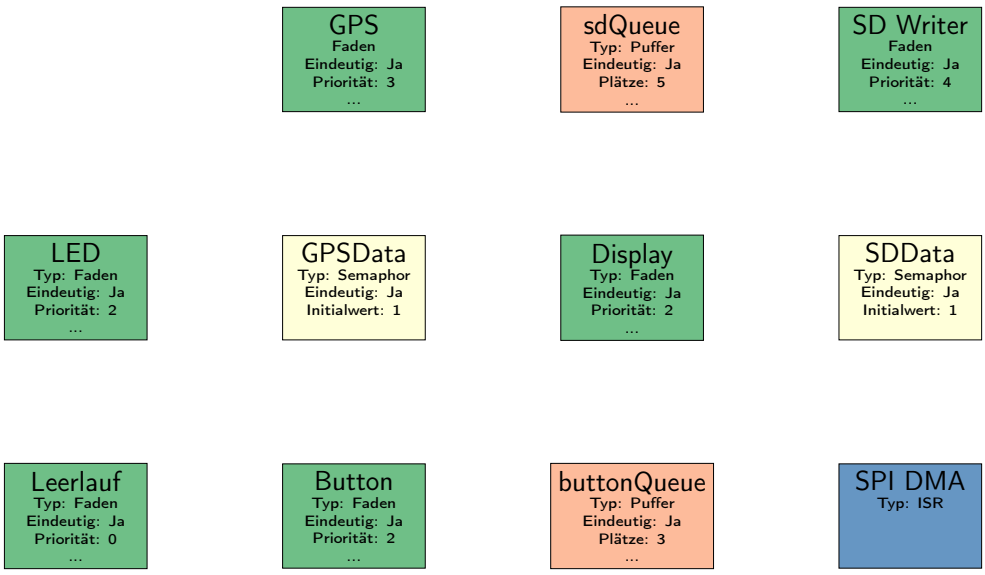

Abbildung 1.1 – Der Instanzgraph der Beispielanwendung GPSLogger. Sechs Fäden (grün) interagieren über zwei Warteschlangen (orange) und zwei Semaphore (gelb). Die Ansteuerung externen Peripherie geschieht über Unterbrechungsbehandlungsroutinen (ISR) (blau).

Lage ist, sich selber zu übersetzen? Braucht es einen Texteditor und die Möglichkeit, Komponenten des Betriebssystemkerns nachträglich dynamisch dazu zu laden? Einige Echtzeitbetriebssysteme erlauben diese Freiheit [Wit00]. Mangels Benutzerschnittstellen am eingebetteten System und aufgrund vorhandener potenter Arbeitsplatzrechner wird aber wohl kein eingebettetes System als Entwicklungs- und Übersetzungsrechner verwendet, sondern die Programmierarbeit am Arbeitsplatzrechner durchgeführt und dort für die Zielplattform übersetzt.

Die Entscheidung für das richtige Betriebssystem und die korrekte Konfiguration dessen bedeutet jedoch eine mentale Belastung für die Anwendungsentwickler. Diese Entscheidung wird nicht unbedingt projekt- und anwendungsspezifisch getroffen [EET19]. Vielmehr werden bereits bekannte Systeme weiterverwendet oder die Entscheidung wird nicht anhand technischer Kriterien getroffen. Obliegt die Entscheidung den Entwicklern, sind ein geringes mentales Modell und ein eindeutiger Lösungsweg wichtige Kriterien einer guten Schnittstelle zwischen Mensch und Maschine [Ras00]. Eine automatische Spezialisierung des Betriebssystems führt dazu, dass diese Belastung reduziert wird, da die Auswahl durch den automatischen Spezialisierungsprozess übernommen wird. Werden die funktionalen Anforderungen des Systemaufrufs automatisch auf die am besten passende Implementierung abgebildet, so müssen Softwareentwickler weder wissen, welche Methode gewählt wurde, noch müssen sie zwischen verschiedenen Funktionen wählen, deren Resultat funktional gleichwertig ist.

Neben der Konfiguration des Betriebssystems zählt hierzu auch die Entscheidung über den richtigen Ausführungszeitpunkt. Dynamische Schnittstellen, die zur Laufzeit evaluiert werden, sind hilfreich, wenn ihr Ergebnis von Entscheidungen abhängt, die eine Laufzeitberechnung erfordern. Ist aber ihr Aufruf oder gar ihr Ergebnis frei von Laufzeitzuständen, so können diese durch äquivalente Vorausberechnungen zur Übersetzungszeit ersetzt werden. Diese Technik ist auf dem Feld der Übersetzer bewährt und in diversen Varianten bekannt. Arithmetische Ausdrücke werden im Rahmen der Konstantenfaltung vorausberechnet. Deren Ergebnisse werden teils über Funktionsgrenzen hinweg propagiert oder ganze Funktionsaufrufe eliminiert [ASU86]. Damit kann zusätzlich auch die Arbeitslast des resultierenden Systems reduziert werden, da sowohl Rechenzeit als auch Speicher eingespart werden.

Beispielhaft lässt sich dazu eine der Anwendungen betrachten, die auch im weiteren Verlauf dieser Arbeit zu Evaluationszwecken betrachtet wird. Abbildung 1.1 zeigt den Instanzgraphen des GPSLoggers, eines Geräts zum Empfangen, Anzeigen und Speichern von Positionsdaten eines globalen Satellitennavigationssystems. Eine genauere Betrachtung dieses Systems findet in Abschnitt 4.4.2.2 statt. Diese Anwendung verwendet sechs Fäden, zwei Puffer und zwei Semaphore zur Umsetzung ihrer Funktionalität. Das verwendete Betriebssystem FreeRTOS bietet über dies hinaus jedoch auch unter anderem MessageBuffer, QueueSets, StreamBuffer an. Zudem bietet FreeRTOS die Flexibilität, dass alle Instanzen dieser Systemobjektklassen zur Laufzeit erzeugt werden. Die Anwendung benötigt aber wie hier dargestellt, nur die genannten acht Systemobjekte. Auch die Interaktionen dieser untereinander sind bereits bekannt. Aller von FreeRTOS getätigter Verwaltungsaufwand für Instanzen und Interaktionen über die in der Abbildung gelisteten hinaus ist dementsprechend überflüssig. Durch Einsparung dieser ließen sich sowohl der Laufzeitaufwand der einzelnen Systemaufrufe als auch der benötigte Speicherplatz für den Betriebssystemcode und Laufzeitzustand verringern.

1.2 Kontext der Arbeit

Die Anwendungsfälle für Computersysteme sind sehr vielfältig geworden. Sie reichen von großen Datenfluten verarbeitenden Serverfarmen und Rechenzentren über Allzwecksysteme als Arbeitsplatzrechner hin zu tief eingebetteten Steuergeräten, die als Teil eines Gesamtsystems agieren.

Auf jedem dieser Themenfelder werden unterschiedliche Anforderungen an die Computersysteme gestellt, um die jeweiligen Anwendungsfelder abzudecken. Der Fokus dieser Arbeit liegt auf jenen eingebetteten Systemen, die nicht Hauptbestandteil eines Produktes sind, sondern darin integriert am Gesamterfolg durch Erfüllung einer Teilaufgabe mitwirken. Kernaspekt dieser Arbeit ist die Anpassung des Betriebssystems an die Anwendung. Hierbei werden alle funktionalen Eigenschaften des Betriebssystems aus Sicht der Anwendung unangetastet umgesetzt. Die nichtfunktionalen Eigenschaften des Zusammenspiels aus Anwendung und Betriebssystem hingegen können variieren und sind Ziel der in dieser Arbeit vorgestellten Optimierungen.

Bei den betrachteten eingebetteten Systemen liegen Ressourcenbeschränkungen im Sinne der verfügbaren Rechenleistung und des verfügbaren Speichers, aber auch oft des erlaubten Energieverbrauchs vor. Neben diesen funktionalen Rahmenbedingungen sind auch wirtschaftliche Kriterien ausschlaggebend. Der Stückpreis einer Einheit hat aufgrund hoher Produktionszahlen auch bei geringen Änderungen einen großen Einfluss auf die Gesamtbilanz. In einem Produkt wie einem Automobil, in dem heutzutage einige zehn bis mehrere Hundert Steuergeräte verbaut sind, bedeuten bereits einstellige Centbeträge kritische Materialkosten. Bei einer Jahresproduktion von 9 305 000 Fahrzeugen und einem Mehrpreis von einem Cent pro Auto bedeutete dies für den Volkswagenkonzern im Jahr 2020 bereits eine Mehrausgabe von 93 050 € [AG21]. Diese Einschränkungen verbieten es, komplexe Probleme durch Überdimensionierung des Systems zu lösen. Vielmehr ist eine gute Anpassung der bereitgestellten Funktionalität an die Anforderungen notwendig.

Um die Soft- und Hardwarekomponenten bestmöglich aufeinander abstimmen zu können, ist es essenziell, die gesamte Anwendungssoftware und ihren Quellcode zu kennen. Für Systeme, die als Gesamtsystem in den Programmspeicher eines Steuergeräts geschrieben werden, ist zum Zeitpunkt des Übersetzens des Systems der gesamte Quellcode bekannt. Zu diesem Zeitpunkt setzt diese Arbeit an, das Gesamtsystem zu betrachten und zu optimieren. Aus Sicht dieser Arbeit besteht der Programmcode aus zwei Bereichen: Die Anwendung, die die eigentliche Funktionalität implementiert, die von dem Steuergerät erwartet wird und das Betriebssystem als Unterbau unter der Anwendung, um deren Ablauf und Funktion mittels geeigneter Primitiven zu ermöglichen.

Die Optimierungen und Anpassungen, die im Rahmen dieser Arbeit behandelt werden, beziehen sich allesamt auf die Interaktion der Anwendung mit einem Betriebssystem. Die Anwendung wird hierbei stets als gegeben und unveränderlich angesehen und ihre Optimierung ist ebenso wenig Teil dieser Arbeit wie die Anpassung der Hardware an die Anwendung. Vielmehr werden im Rahmen dieser Arbeit das Betriebssystem angepasst und die Aufrufstellen entsprechen an die angepassten Systemaufrufe adaptiert. Dies ermöglicht eine passgenaue Adaption der generischen Betriebssystemschnittstelle an die konkreten Anforderungen und Verwendungsmuster der Anwendung. Nicht notwendige Funktionalität und Flexibilität können so aus dem Betriebssystem und dessen Schnittstelle entfernt werden, ohne dass sich diese aus Sicht des Programmierers ändert [DL17; Sch11].

Die gesamte Gewinnung des notwendigen Anwendungswissens wird mittels statischer Codeanalyse umgesetzt. Die Analysen und Anpassungen betrachten jeweils die Anwendung in Gänze. Dies ermöglicht funktionsübergreifende Optimierung des Gesamtsystems [Wä+18; Die19]. Es kann daher an allen Stellen vorausgesetzt werden, dass keine Informationen im späteren Übersetzungsprozess hinzukommen, die eine Optimierung verbieten würden.

Eine händische Anpassung des Betriebssystems an die Anwendung ist nicht nur fehleranfällig, sondern auch sehr aufwendig. Aus diesem Grund wurde bisher eine solche Anpassung, wenn möglich, vermieden. Im Rahmen dieser Arbeit werden dementsprechend alle Anpassungen in die Kette der Übersetzungswerkzeuge integriert, sodass die Arbeitsschritte vollautomatisch und nahtlos in den Übersetzungsprozess integriert werden können, ohne menschliches Eingreifen zu erfordern. Durch die Integration in die Übersetzungswerkzeugkette verhalten sich die Anpassungen wie ein regulärer

optimierender Übersetzer, ziehen jedoch zusätzlich die Semantik des jeweiligen Betriebssystems mit in Betracht.

Entsprechend dem Konzept eines optimierenden Übersetzers verhalten sich alle im Folgenden in dieser Arbeit entwickelten Anpassungen tolerant gegenüber nicht spezialisierbaren Anwendungen. Diese werden nicht negativ beeinflusst, sondern arbeiten weiterhin regulär mittels der unveränderten Betriebssystemimplementierung. Diejenigen Betriebssysteminteraktionen, die optimierbar sind, werden weitestmöglich optimiert, sodass ein fließender Übergang entsteht.

Die bereitstehenden Programmiermodelle von Echtzeit- und eingebetteten Betriebssystemen lassen sich in zwei große Kategorien teilen. Den einen großen Bereich belegen diejenigen Systeme, die statisch konfiguriert werden. Hierzu zählen Systeme wie die OSEK/AUTOSAR-Familie [OSE05] oder µTRON [Ass10], die mittels Konfigurationsdateien zur Übersetzungszeit konfiguriert werden und deren Betriebssystemcode teils erst anhand dieser Konfiguration für die jeweilige Anwendung generiert wird. Hierbei stehen zur Übersetzungszeit bereits die Systemobjekte fest, die zur Laufzeit existieren und wie diese miteinander interagieren. In verwandten Arbeiten von Dietrich; Scheler wurden derartige Systeme bereits tiefgreifend analysiert und entsprechend den Anforderungen der Anwendungen spezialisiert [Die19; Sch11].

Dem entgegen steht der zweite Bereich der dynamisch konfigurierten Systeme. Dynamische Konfiguration bedeutet hierbei, dass die Systemobjekte erst zur Laufzeit erzeugt werden und ihre Existenz zur Übersetzungszeit nicht bekannt ist. Der Vorteil dieser liegt darin, dass auf Anforderungen zur Laufzeit reagiert werden kann und Systemobjekte nach Bedarf erzeugt werden können. Diese Flexibilität und das gewohnte Gefühl bei dem Umgang mit den Programmierschnittstellen führen zu einer starken Verbreitung derartiger Betriebssysteme in eingebetteten Anwendungen. Prominente Vertreter dieses Systemmodells sind EmbeddedLinux und FreeRTOS, welches laut aktueller Studie einen Marktanteil von 21 % und 18 % abdecken [EET19]. Sie bilden gemeinsam mit selbst entwickelten Betriebssystemen die drei meistverwendeten Betriebssysteme.

Ohne tiefgreifende Analysen scheint das Optimierungspotenzial statisch konfigurierter Systeme zwar anfänglich größer, da das notwendige Wissen expliziter genannt ist. Es besteht jedoch die Annahme, dass für viele Anwendungsfälle die Flexibilität der dynamisch konfigurierten Systemmodelle nicht ausgeschöpft wird, sondern im Regelfall die Konfiguration aus dem Quellcode entnommen werden kann. Dies resultiert in einem mindestens ebenbürtigen, wenn nicht sogar durch Entfernen der zum Abbilden der nicht genutzten Freiheit überflüssigen Komplexität erhöhtem Optimierungspotenzial. Diese Arbeit befasst sich ausschließlich mit den letztgenannten dynamisch konfigurierten Systemen. Diese sind weit verbreitet im Einsatz in eingebetteten Systemen [EET19], jedoch existieren bisher keine derart weitgreifenden anwendungsgewahren Spezialisierungsverfahren wie für die oben genannten statisch konfigurierten Systeme. Das Ziel dieser Arbeit ist es, diese Lücke zu schließen.

Der Lebenszyklus einer Echtzeitanwendung in Ausführung besteht aus mehreren Phasen: In der ersten Phase, dem Systemstart, werden die Hardwarekomponenten und üblicherweise auch das Betriebssystem initialisiert und die Softwarekomponenten vorbereitet und gestartet. Sobald alle Komponenten einsatzbereit sind, geht das System in die zweite Phase, die Arbeitsphase, über. In der Arbeitsphase arbeiten alle Komponenten ihre zeit- und ereignisgesteuerten Aufgaben und Interaktionen ab. In dieser Phase verbleibt das System für seine gesamte weitere Laufzeit. Ausschließlich im Falle eines nicht behebbaren Fehlers oder der Abschaltaufforderung verlässt das System die Arbeitsphase in eine Rücksetzphase. In dieser werden eventuell Hardware- und Softwarekomponenten abgeschaltet oder in ihren ursprünglichen Zustand versetzt. Aus dieser Phase ist keine direkte Rückkehr in die Arbeitsphase möglich, sondern ausschließlich über einen Neustart und die Systemstartphase. Dementsprechend lassen sich auch die Optimierungen den Phasen zuordnen. Das für statisch konfigurierte Systeme bereits existierende breite Spektrum von Spezialisierungen [Die19; Hof16; Hof14; Sch11] werde ich in dieser Arbeit um statische Spezialisierungen für die Start- und die Arbeitsphase dynamisch konfigurierter eingebetteter Systeme erweitern.

Statische Optimierung ist für übersetzte Sprachen grundlegender Teil des Übersetzungsprozesses. Bereits frühe Übersetzer konnten arithmetische Konstanten falten [Wul+73]. In aktuellen Übersetzern erstrecken sich diese Optimierungen auf die Evaluation vollständiger Funktionen und auch Konstruktoren zur Übersetzungszeit [LA04]. Auch bei dynamisch übersetzten und interpretierten Programmiersprachen halten Optimierungstechniken wie statische Evaluation Einzug. Die Analysen und Optimierungsmöglichkeiten sind hierbei jedoch immer auf die Semantik der jeweiligen Programmiersprache beschränkt. In dieser Arbeit hingegen wird neben der Semantik der Programmiersprache auch die der verwendeten Betriebssystemschnittstelle beachtet. Daraus resultiert die Möglichkeit, auch deren Auswirkungen statisch zu berechnen.

1.3 Wissenschaftlicher Rahmen dieser Arbeit

Diese Arbeit ist Teil des Forschungsprojekts AHA (DFG LO 1719/401, „Automatisierte Hardware-Abstraktion im Betriebssystembau“). In diesem geht es darum, das Betriebssystem bestmöglich anhand des anwendungsspezifischen Betriebssystemverwendungsmodells an die Bedürfnisse der Anwendung anzupassen. Diese Anpassungen erstrecken sich über die Auswahl und Generierung angepasster Komponenten aus Software und Hardware. Um die Bedürfnisse der Anwendungen besser beschreiben zu können, haben wir die Notation der Spezialisierungsebenen eingeführt [▷Fie+18]. Demnach lassen sich die Anforderungen einer Anwendung in verwendete *Abstraktionen*, deren vorhandene *Instanzen* und den *Interaktionen* zwischen diesen einteilen. Auch haben wir in dieser Arbeit den Begriff der Spezialisierung und insbesondere einer gültigen Spezialisierung hinsichtlich einer gegebenen Anwendung definiert. Eine gültige Spezialisierung muss dabei alle funktionalen Eigenschaften, die von der Anwendung gefordert sind, exakt erfüllen wie die generische Implementierung der Betriebssystemschnittstelle. Die Spezialisierung hat jedoch die Freiheit, nicht verwendete Funktionalität auszusparen und auch in nichtfunktionalen Aspekten von der generischen Implementierung abzuweichen. Dies führt dazu, dass die spezialisierte Variante der Hardware-Software-Kombination nicht mehr in der Lage ist, beliebige Anwendungen auszuführen, sondern nur solche, die die gleichen oder eine Teilmenge der Anforderungen an das Betriebssystem stellen, wie diejenige Anwendung auf die spezialisiert wurde. Um dies auszudrücken, haben wir in [▷Fie+18] die Notation der Instanzgraphen eingeführt. Diese beschreiben die Menge aller durch eine Betriebssystemimplementierung ausführbaren Anwendungen als diejenigen Instanzgraphen, die als eine Teilmenge eines solchen Instanzgraphen darstellbar sind. Abbildung 2.4(g) zeigt beispielhaft einen Instanzgraphen einer generischen Betriebssystemimplementierung, die beliebig viele Fäden, Unterbrechungsbehandlungsroutinen (ISRs) und Ereignisse verwalten kann. Zwischen allen Instanzen dieser Abstraktionen sind alle durch Kanten dargestellten Interaktionen beliebig möglich. Verwendet eine konkrete Anwendung exakt zwei Fäden und eine ISR, aber keine Ereignisse, so ist eine Spezialisierung entsprechend Abbildung 2.4(b) für diese Anwendung eine gültige Spezialisierung. Für eine Anwendung mit drei Fäden wäre dies jedoch keine gültige Spezialisierung.

In dieser Arbeit konzentriere ich mich auf eben diese Beschneidung des Interaktionsgraphens auf die Notwendigkeiten der Anwendung. Dabei betrachte ich diejenigen Betriebssysteme, die regulär keine statische Konfiguration ihrer Instanzen und Interaktionen aufweisen, sondern diese dynamisch zur Laufzeit bestimmen. Ziel dieser Arbeit ist es, trotz der fehlenden expliziten Angabe der funktionalen Anforderungen des anwendungsspezifischen Betriebssystemverwendungsmodells die Betriebssystemimplementierung entsprechend zu spezialisieren. Hierzu werden im Rahmen dieser Arbeit die zwei relevanten Phasen des Lebenszyklus der Anwendung betrachtet: In der Systemstartphase werden die verwendeten Instanzen erzeugt, die während der Arbeitsphase miteinander interagieren.

1.4 Ziel dieser Arbeit

Die Entwicklung und Implementierung der Systemsoftware eines eingebetteten Systems unter Verwendung eines Betriebssystems mit dynamischer Programmierschnittstelle hat diverse Vorteile gegenüber der Verwendung einer Betriebssystemschnittstelle, die statisch konfiguriert werden muss. So können Entscheidungen, die zur Laufzeit des Systems ausgewertet werden, oftmals einfacher beschrieben werden und verringern die Notwendigkeit von redundantem oder duplizierten Programmcode in der Anwendung, um diese Entscheidungen durch den Programmierer statisch auszudrücken. Auch werden keine externen Generatoren mit dazugehörigen anwendungsspezifischen Konfigurationsdateien benötigt, da die gesamte Konfiguration der Anwendung und insbesondere deren Verwendung des Betriebssystems im Programmcode ausgedrückt ist. Auf diese Weise ist die gesamte Information an einem Ort vorhanden und nicht über mehrere Komponenten verteilt, die stets zueinander synchron gepflegt werden müssen. Somit entfällt diese Aufgabe und die daraufhin frei gewordene Kapazität der Entwickler kann in andere Bereiche der Anwendungsentwicklung investiert werden.

Gleichzeitig führt diese dynamische Ausdrucksweise jedoch zu einer erhöhten Betriebslast, da das Auswerten der immer gleichen Entscheidung unnötige Rechenzeit und Programmcode in Anspruch nimmt. Besser wäre hierbei diejenigen Systemaufrufe, die statisch berechenbar sind, im Übersetzungsprozess in statisch vorausberechnete Äquivalente zu wandeln. Nur die Systemaufrufe, die tatsächlich auf Laufzeitentscheidungen angewiesen sind, führen diese weiterhin aus.

Daraus resultiert das Ziel dieser Arbeit: Die Umwandlung von vormals dynamischen Systemaufrufen in statisch vorausberechnete Systemaufrufe sobald möglich. Dies führt zu einer Betriebssystemschnittstelle, deren Auswirkungen zur Laufzeit entschieden werden können, falls diese Flexibilität notwendig ist und andernfalls davon profitiert, dass die Auswirkungen bereits zur Übersetzungszeit vorausberechnet wurden und dadurch eine geringere Betriebslast verursachen. Im Rahmen dieser Arbeit wird dies für Systemaufrufe der Initialisierungsphase und der Arbeitsphase umgesetzt. Verläuft die Systemstartphase regelmäßig, so werden bei jedem Systemstart immer wieder die gleichen Systemobjekte neu erzeugt, ohne dass dieser Ablauf sich wandelt. Nur wenige (Teil-)Entscheidungen werden tatsächlich zur Laufzeit evaluiert und führen dementsprechend auch nur zu kleinen Änderungen im Startvorgang. Dementsprechend ergibt sich hier ein großes Potenzial, die Vorausberechnung auszunutzen. Während der Arbeitsphase findet die Kommunikation der Systemobjekte auch zumeist zwischen den gleichen Kommunikationspartnern statt. Dies ermöglicht auch hier eine Teilvorausberechnung der Kommunikationsaufrufe, sodass diese direkt zugeschnitten sind auf den Ablauf der jeweiligen Anwendung.

Aus diesen Zielen der Arbeit ergeben sich die folgenden drei Forschungsfragen, die mittels dieser Arbeit beantwortet werden:

Forschungsfrage 1 Lässt sich die Flexibilität eines dynamischen Systemmodells mit den Laufzeitvorteilen eines statischen Systemmodells kombinieren, sodass die Flexibilität falls notwendig erhalten bleibt, ansonsten aber die Laufzeitvorteile des statischen Systemmodells ausgenutzt werden können?

Forschungsfrage 2 Wie ändern sich der Laufzeitaufwand und der Speicherbedarf durch die Verringerung des Verwaltungsaufwands zur Laufzeit durch die statische Vorausberechnung von Systemaufrufen in dynamischen Betriebssystemmodellen?

Forschungsfrage 3 Welche Hindernisse stehen der statischen Vorausberechnungen kategorisch entgegen und wie kann damit umgegangen werden?

1.5 Aufbau der Arbeit

In den folgenden Abschnitten dieser Arbeit gebe ich Einblicke in die notwendigen Grundlagen zur detaillierten Auseinandersetzung mit den aufgestellten Forschungsfragen und beantworte diese anhand einer konkreten Umsetzung des Vorhabens und einer Evaluation der Resultate. Diese Ausarbeitung ist dazu in die folgenden Abschnitte geteilt.

Kapitel 2 *Grundlagen – Begriffe und verwandte Arbeiten* (pp. 11–28)
Um sich detailliert mit den Forschungsfragen auseinanderzusetzen, bedarf es einer Einführung wichtiger Grundbegriffe und einer gemeinsamen Sicht auf eingebettete Systeme. Dieses Kapitel enthält außerdem eine Einführung in die verschiedenen Systemmodelle und deren Unterschiede im Hinblick auf die nachfolgenden Verwendungen und Anpassungen. Statische Berechnungen sind insbesondere im Umfeld der optimierenden Übersetzer bereits weit verbreitet. Dementsprechend werden die Grundlagen aus diesem Bereich eingeführt und Schnittstellen, Parallelen und Unterschiede zu dieser Arbeit aufgezeigt.

Kapitel 3 *Voraussetzungen – Ermittlung des notwendigen Anwendungswissens* (pp. 29–41)
Die in dieser Arbeit vorgestellten Optimierungen und Anpassungen des Betriebssystems setzen Wissen über die auszuführende Anwendung und das verwendete Systemmodell voraus. Dieses Wissen erstreckt sich über die verwendeten Systemobjekte wie Fäden und Synchronisationsobjekte hinaus zu deren Interaktionen untereinander. In diesem Kapitel werden die notwendigen Informationen und deren Extraktion aus dem Programmcode der Anwendung betrachtet.

Kapitel 4 *Systemstart – Wiederholte Abläufe ohne Wiederholung ausführen* (pp. 43–69)
Auch wenn der Systemstart von Ferne betrachtet wie ein immer wiederkehrender gleichbleibender Ablauf erscheint, so wird er bei Systemen mit dynamischem Systemmodell dennoch bei jedem Durchlauf neu ausgeführt und es werden neu zur Laufzeit Entscheidungen getroffen, die dennoch immer ein gleiches Ergebnis hervorrufen. Das Systemstartkapitel behandelt die Optimierung dieser ersten Phase des Lebenszyklus einer Anwendung in Ausführung mit dem Ziel, die Auswirkungen gleichbleibender Systemaufrufe aus der Laufzeit in die Übersetzungszeit zu verlagern.

Kapitel 5 *Interaktionen – Bei bekannten Interaktionen weniger interagieren* (pp. 71–83)
Auch die Interaktion zwischen den Systemobjekten ist nicht immer so dynamisch, wie das Systemmodell dies zuließe. In diesem Kapitel widme ich mich den Interaktionen der Systemobjekte während der Arbeitsphase des Systems. Anhand des statisch erlangten Wissens werden Interaktionspartner und auch Auswirkungen der Interaktionen vorausberechnet.

Kapitel 6 *Idiomatik und Semantik – Hindernisse und Einflussnahme auf die Spezialisierbarkeit* (pp. 85–101)
Nachdem die vorherigen Kapitel verschiedene Spezialisierungen und deren Auswirkungen beleuchtet haben, behandelt dieses Kapitel die Untersuchung der Idiomatik und Semantik der verwendeten Programmiersprachen, die der Spezialisierung hinderlich sind. Es wird evaluiert, welche Konzepte und Verwendungen dem Analysieren oder Spezialisieren entgegenstehen und wie durch Mitwirkung der Entwickler oder Konfiguratoren des Systems die Spezialisierung dennoch ermöglicht werden kann.

Kapitel 7 *Schluss – Zusammenfassung, Ergebnisse und Ausblick* (pp. 103–110)
Abschließend fasst dieses Kapitel die gewonnenen Erkenntnisse zusammen. Es werden die

Auswirkungen zusammenfassend diskutiert sowie ein Ausblick auf weitere Forschungsmöglichkeiten in diesem Themenfeld gegeben.

1.6 Typografische Konventionen

Bei Zitaten markiert ein Dreieck, dass ich der Hauptautor oder einer der Mitautoren bin (z. B. [▷Fie+18]). Neu eingeführte Begriffe werden durch *kursive* Schrift hervorgehoben und die Randnotizen geben einen groben Überblick über die behandelten Themen. Ebenso wird *kursive* Schrift als stilistisches Mittel für allgemeine Hervorhebungen verwendet. Funktionen und Programmvariablen werden in dicktengleicher Schrift gesetzt (`Funktion()`, `Variable`). Namen von Werkzeugen und Programmen werden in Kapitälchen gesetzt (WERKZEUG, PROGRAMM). Bei der Verwendung von Abbildungen aus veröffentlichten Werken verwende ich den Ausdruck „In Anlehnung an [Zitat]“, um eine abgewandelte Variante der Ursprungsversion zu kennzeichnen und „Aus [Zitat]“ für eine unveränderte Verwendung.

2

Grundlagen

Begriffe und verwandte Arbeiten

Als Grundlage für die folgenden Abschnitte dieser Arbeit ist es notwendig, eine geeignete Basis an Grundbegriffen und gemeinsamen Definitionen anzulegen, sodass im späteren Verlauf aufbauend auf diese die durchgeführten Maßnahmen der Spezialisierung aufgesetzt werden können.

Spezialisierung von Betriebssystemen hat eine lange Tradition in der Informatik und der Wunsch nach anwendungsangepasster Systemsoftware ist bereits früh aufgekommen [Dij68]. Dementsprechend befasst sich der zweite Abschnitt dieses Kapitels mit den verwandten Arbeiten, die auf dem Gebiet der anwendungsgewahren Spezialisierung von Betriebssystemen bereits existieren.

2.1 Das eingebettete Echtzeitsystem als Spezialzwecksystem

Die Vielfalt von Computersystemen und deren Einsatzmöglichkeiten sind divers. Dies macht die Klassifizierung dieser notwendig, da je nach Eigenschaften andere Anforderungen und Möglichkeiten und Einschränkungen auftreten.

Eingebettetes System

Ein *eingebettetes System* ist ein Computersystem, welches als informationsverarbeitende Einheit integraler Bestandteil eines größeren Gesamtsystems arbeitet. Es ist dabei so in das Gesamtsystem integriert, dass es nicht als eigenständiges Rechensystem wahrgenommen wird, sondern erfüllt hierbei maßgeblich zur Aufgabenerfüllung des Gesamtsystems beitragende Steuerungs- und Überwachungsaufgaben. Es ist dazu mit Sensoren und Aktoren zur Wahrnehmung und Beeinflussung der Umwelt und des umgebenden Systems ausgestattet [BU02; Mar06].

Beispielhaft kann hierzu das Steuergerät eines Quadrokopters betrachtet werden. Es führt die Analyse der Umweltsensoren und die Berechnung der daraus resultierenden Steuersignale der Motoren zur Fluglageregelung aus. Dieses Steuergerät alleine ist nicht in der Lage zu fliegen, es benötigt die anderen Systemkomponenten wie Lagesensoren, Rotormotoren und Chassis des Quadrokopters. Ohne das Steuergerät und die darauf ausgeführte Anwendung zur Fluglageregelung wären die anderen Komponenten des Systems aber ebenso flugunfähig. Sind alle Komponenten in das Gesamtsystem integriert, ist die Einzelleistung des eingebetteten Systems jedoch nicht mehr trennbar. Das eingebettete System ist somit integraler Bestandteil des Gesamtsystems.

Echtzeitsystem

Aufgrund der umgebenden Umwelt und ihrer Anforderungen sind eingebettete Systeme zeitlichen Anforderungen ausgesetzt. Anders als bei Vielzwecksystemen wie beispielsweise Arbeitsplatzrechnern ergibt sich hieraus, dass es sich um Echtzeitsysteme handelt. Ein *Echtzeitsystem* ist eines, bei dem neben der Richtigkeit eines Ergebnisses auch dessen zeitlich korrektes Vorhandensein Teil der funktionalen Anforderungen ist [Liu00; Mar06; BU02]. Muss ein System nicht nur die Anforderung richtige Ergebnisse, sondern auch deren rechtzeitiges Vorhandensein erfüllen, so wird es als Echtzeitsystem bezeichnet. Dies bedeutet, dass bei einem Echtzeitsystem zeitliche Eigenschaften wie die Garantie einer Antwortzeit zu den funktionalen Eigenschaften des Systems zählen [Kop97].

Echtzeitbetriebssystem

Ein *Echtzeitbetriebssystem (EZBS)* ist die Grundlage und Ausführungsplattform der jeweiligen Anwendung eines Echtzeitsystems. Es vermittelt als zentrales Bindeglied zwischen den Komponenten der Anwendung und der zur Ausführung verwendeten Hardwareplattform, stellt Abstraktionen wie Fäden, Semaphore und Warteschlangen bereit, die die einzelnen Aufgaben ausführen und zur Synchronisation dieser verwendet werden und ermöglicht so die Arbeit der Anwendung. Es ist dabei die Instanz, die die Einhaltung der Echtzeitanforderungen überwacht und durchsetzt.

In dieser Arbeit betrachte ich Anwendungen für eingebettete Echtzeitsysteme und insbesondere deren Echtzeitbetriebssysteme als Ziel der Spezialisierungen. Die in dieser Domäne üblichen Programmierstandards [ISO11; Mis] verbessern die Anwendbarkeit, jedoch sind sie keine zwingende Voraussetzung. Zudem haben die Auswirkungen der Spezialisierung, insbesondere der Laufzeitveränderung, einen direkten Beitrag auf das Erfüllen der funktionalen Eigenschaften des Gesamtsystems und sind somit lohnenswertes Optimierungsziel. Die resultierenden Spezialisierungen und deren Anwendung sind aber ebenso auf Anwendungen ohne Echtzeitanforderungen übertragbar.

Spezialzwecksystem

Im Gegensatz zu Arbeitsplatzcomputern, bei denen die auszuführende Anwendung während der Benutzung frei variieren kann [MSS99], handelt es sich bei den in dieser Arbeit betrachteten eingebetteten Rechensystemen um *Spezialzwecksysteme*, deren auszuführende Anwendung zum Konzeptions- oder Entwicklungszeitpunkt festgelegt wird und danach nicht mehr veränderlich ist [Mar06; Kop97]. Dies bedeutet, dass sowohl Hardware als auch Software entsprechend dem designierten Zweck ausgewählt oder auch für diesen spezialisiert werden können.

Dies ist jedoch kein Widerspruch zu der Möglichkeit, die Anwendung und das Betriebssystem zu aktualisieren und zu warten. Entscheidend ist nur, dass zum Zeitpunkt der Integration von Anwendung und Betriebssystem in das entsprechende Speicherabbild der gesamte Code vorliegen muss, sodass die Spezialisierungen auf diesem durchgeführt werden können. Für die Aktualisierung des Programmcodes funktioniert der klassische Weg, das gesamte Speicherabbild auszutauschen [Kop97], weiterhin ohne Einschränkungen, da so Anwendung und Betriebssystem stets synchron gehalten werden. Diese sind dementsprechend analog einer Erstauslieferung behandelbar. Ein Urlader kann beispielsweise mehrere Versionen derselben Anwendung oder auch verschiedener Anwendungen samt dazugehörigen spezialisierten Betriebssystemen im Programmspeicher verwalten und die richtige Version laden. Es existieren jedoch auch feingranularere Methoden der Programmcodeaktualisierung [VA20], die mit den in dieser Arbeit vorgestellten Verfahren kompatibel sind, sodass der Anwendungscode auch partiell aktualisiert werden kann. Die genauen Voraussetzungen dazu behandle ich bei den entsprechenden Spezialisierungen. Dies ermöglicht eine Aktualisierbarkeit des Gesamtsystems und die Voraussetzung, Anwendung und Betriebssystem zum Zeitpunkt der Spezialisierung als Einheit vorliegen zu haben, ist keine grundsätzliche Einschränkung für die Aktualisierbarkeit als wichtige Systemeigenschaft.

Durch die Festlegung des Einsatzzweckes eines eingebetteten Systems wird auch das darauf auszuführende Programm festgelegt. Dieses im folgenden *Echtzeitanwendung* oder kurz *Anwendung* genannte Programm implementiert die notwendige Ansteuerung der Sensoren und Aktoren sowie die zur Aufgabenerfüllung notwendige Steuerungslogik. Ausgeführt wird diese Anwendung auf der gewählten *Hardware* bestehend aus einem Prozessor und dessen Peripherie und Schnittstellen.

Im Rahmen dieser Arbeit werden die Softwarekomponenten, die das Spezialzwecksystem implementieren, in die zwei Bestandteile Anwendung und Betriebssystem klassifiziert. Die Anwendung beinhaltet alle Softwarekomponenten, die zur Implementierung des festgelegten Systemzwecks notwendig sind. Hierzu zählen Gerätetreiber für die verwendeten Sensoren und Aktoren ebenso wie die eigentliche Logik, die aufgrund der Umgebungswahrnehmung die entsprechenden Steuerungs- und Kommunikationsvorgänge veranlasst. Davon abgetrennt ist das Betriebssystem, welches als Vermittlungsschicht zwischen Anwendung und Hardware in vermittelnder und zuteilender Funktion agiert. Aufgabe des Betriebssystems ist es, durch Virtualisierung und Zuteilung der Systemressourcen eine geeignete Ausführungsumgebung für die Anwendung bereitzustellen.

Die im Folgenden dargestellten Spezialisierungen des Betriebssystems sind nur vollständig möglich, wenn der Einsatzzweck und die daraus resultierende und die Aufgabe des Systems implementierende Anwendung in Form eines abgeschlossenen Gebindes zum Zeitpunkt der Übersetzung vorliegt. Für Spezialzwecksysteme ist diese Eigenschaft gegeben. Es gibt keine Eventualitäten oder Programmteile, die nach der Übersetzung des Systems nachgerüstet werden können. Insbesondere in Kombination als eingebettetes Spezialzwecksystem ist es üblich, dass zum Integrationszeitpunkt die gesamte Anwendung inklusive aller Bibliotheken und sonstiger benötigter Quellcodeelemente vorliegt und zu einem Maschinenprogramm zusammengebaut wird. Zu diesem Zeitpunkt ist es dementsprechend möglich, eine vollständige Spezialisierung durchzuführen.

Ist diese Voraussetzung nicht gegeben, so sind die Methodiken jedoch weiterhin anwendbar. Es muss dann jedoch als pessimistische Überabschätzung davon ausgegangen werden, dass weitere unbekannte Teile der Anwendung vorhanden sind, die nicht spezialisierbar sind. Dies bedeutet eine Verringerung der Möglichkeiten der Spezialisierung, sie wird jedoch nicht unmöglich.

2.2 Spezialisierung: Definition, Chancen und Einschränkungen

Bevor wir uns der Umsetzung der Spezialisierung widmen, bedarf es einer Definition des Begriffes *Spezialisierung*. Im Rahmen dieser Arbeit wird die Definition analog der in [▷Fie+18] verwendet.

Eine Implementierung eines eingebetteten Echtzeitsystems besteht aus den drei Komponenten Anwendung, Echtzeitbetriebssystem und Hardwareplattform.

Die Anwendung beinhaltet die Logik des Systems. In ihr sind sowohl die speziell für dieses System entwickelten Logikkomponenten enthalten wie auch die dafür verwendeten Bibliotheken. Sie umfasst damit den größten Teil der Software und codiert die funktionalen Eigenschaften des Systems.

Das Betriebssystem dient der Anwendung als Ausführungsplattform zur Umsetzung der gewünschten Funktionalität. Es bietet Abstraktionen, die die Implementierung der Anwendung erleichtern, indem es Grundbausteine zur Nebenläufigkeit und Synchronisation bereitstellt. Im Fall eines Echtzeitsystems ist es zudem für die Garantie der zeitlichen Eigenschaften zuständig.

Darunter liegt die Hardware, auf der die Anwendung und das Betriebssystem ausgeführt werden. Diese definiert den Satz gültiger Maschineninstruktionen und die Schnittstellen zu Sensoren und Aktoren.

Definition

Für ein Echtzeitsystem beschreibt dessen Spezifikation (S) dessen korrekte beobachtbare Systemantwort als Vektor von Ausgabewerten ($\vec{A}$) inklusive deren zeitlichen Vorkommens auf einen gegebenen Vektor von zeitanotierten Eingabedaten ($\vec{E}$). Eine konkrete Implementierung dieses Echtzeitsystems ($EZS^{A}_{EZBS\,HW}$), bestehend aus der Anwendung (A) unter Verwendung des Echtzeitbetriebssystems (EZBS) ausgeführt auf einer Hardware (HW), erfüllt die Spezifikation (S) durch Erzeugen des richtigen beobachtbaren Systemverhaltens ($\vec{E}$). Dabei bestimmt die Spezifikation den Gleichheitsoperator ($\overset{S}{=}$) zwischen zwei Ausgabevektoren:

$$S(\vec{E}) = \vec{A} \quad \overset{S}{=} EZS^{A}_{EZBS\,HW}(\vec{E})$$

Jede Implementierung, die das geforderte Ausgabeverhalten produziert, ist demnach eine gültige Implementierung. Demnach muss auch jede Spezialisierung $EZS^{A}_{EZBS\,HW}{}'$ eine gültige Implementierung sein. Nicht jede gültige Implementierung ist jedoch als *Spezialisierung* anzusehen, sondern nur diejenigen, die im Hinblick auf eine gegebene gültige Implementierung deren Funktionsumfang und deren Flexibilität einschränkt oder anpasst. Hierbei muss die Gültigkeit weiterhin erhalten bleiben, die nichtfunktionalen Eigenschaften der Implementierung werden jedoch verbessert [▷Fie+18]. Speicher- und Energiebedarf sowie der Preis der benötigten Hardware sind hierbei typische Effizienzkriterien der nichtfunktionalen Eigenschaften [Mar06].

Spezialisierung kann in jeder der drei Komponenten des Systems durchgeführt werden. Im Rahmen dieser Arbeit findet die Spezialisierung primär innerhalb des Betriebssystems und an dessen Einstiegspunkten innerhalb der Anwendung statt. Hierbei wird das Betriebssystem an die konkreten Anforderungen der Anwendung angepasst, unnötige Flexibilität entfernt und angepasste Implementierungen selektiert und generiert. Dazu werden teilweise die Aufrufstellen der Anwendung und deren Zugriffe auf das Betriebssystem angepasst, sodass die spezialisierten Implementierungen verwendet werden.

Chancen

Durch die Verringerung der Flexibilität des Betriebssystems ergibt sich die Möglichkeit, auf einfachere Implementierungen der Betriebssystemfunktionalität zurückzugreifen oder deren Ergebnis bereits zur Übersetzungszeit vorauszuberechnen. Dazu gehört beispielsweise die Erzeugung von Systemobjekten zur Übersetzungszeit. Dadurch muss die dazu notwendige Programmlogik nicht mehr zur Laufzeit ausgeführt werden und führt zu einer Reduktion der Programmlaufzeit und gleichzeitig zu einer Speicherreduktion durch Entfernen nicht mehr benötigter Initialisierungsfunktionen.

Allgemein bedeutet dies, dass das statisch vorhandene Wissen über die Anwendung genutzt werden kann, um die Kosten eines Produkts zu senken und gleichzeitig dessen Robustheit zu erhöhen [Kop97; Hof14; Hof16; Die19].

Einschränkungen

Wird für ein System ermittelt, dass alle Systemobjekte bereits zur Übersetzungszeit erstellt werden können und daraufhin die Fähigkeit des Betriebssystems zur Erstellung neuer Systemobjekte entfernt, so hat das zur Folge, dass das spezialisierte System keine neuen Objekte zur Laufzeit mehr erstellen kann. Dies bedeutet, dass das spezialisierte System nicht mehr den gleichen Funktionsumfang bereitstellt wie das generische System. Für ein konkretes System bedeutet dies jedoch keine Einschränkung, da alle funktionalen Anforderungen gemäß Spezifikation weiterhin erfüllt werden. Ausschließlich Funktionalitäten außerhalb der Spezifikation sind davon betroffen. Die Einschränkungen sind somit ausschließlich auf dem Feld der nichtfunktionalen Eigenschaften auffindbar. Eine Abwägung von Robustheit gegen Speicherbedarf [HDL13] oder Hardwareaufwand [DL17] abgewogen ist durchaus möglich. Funktional erfüllt jedoch jede Spezialisierung per Definition alle Anforderungen, sodass es keine funktionalen Einschränkungen geben kann. Es wird stets nur von der Anwendung ungenutzte Flexibilität genommen und deren Wegfall ausgenutzt.

2.2.1 Betriebssystemmodelle und deren Programmierschnittstellen

Betriebssystemmodelle und deren Programmierschnittstellen unterscheiden sich je nach Einsatzzweck, Entwicklungsprozess und domänenspezifischen Anforderungen. Für diese Arbeit sind zwei Kriterien besonders von Interesse. Das Kriterium des Programmiermodells beschreibt das Modell, welches Semantik und Idiomatik des Betriebssystems vorgibt. Dazu gehört die Betriebssystemschnittstelle und wie Systemobjekte instanziiert werden. Das Kriterium der Konfigurierbarkeit beschreibt, in welcher Granularität eine Anpassung des Betriebssystems an eine gegebene Anwendung vorgesehen ist.

2.2.1.1 Programmiermodell

Im Rahmen dieser Arbeit wird zwischen zwei *Programmiermodellen* für Betriebssysteme unterschieden: statische und dynamische Betriebssystemmodelle. Diese unterscheiden sich hinsichtlich der Instanziierung von Systemobjekten und der Semantik von Systemaufrufen hinsichtlich der Interaktion der Systemobjekte.

statisches Programmiermodell

Das *statische Programmiermodell* erzwingt die Instanziierung von Systemobjekten zur Übersetzungszeit der Anwendung. Eine in eine in betriebssystemspezifischer Konfigurationssprache ausgedrückte Konfiguration der Systemobjekte ist dabei notwendig zwingender Bestandteil der Anwendung und Voraussetzung für die Verwendung eines solchen Betriebssystems. Diese Konfiguration bestimmt neben den vorhandenen Instanzen und deren Eigenschaften ebenfalls die zwischen diesen erlaubten Interaktionen.

Systeme, die die OSEK-Spezifikation [OSE05] erfüllen, sind klassische Vertreter dieses Paradigmas. Bei diesen beschreibt die Systemspezifikation in der domänenspezifischen Sprache *OSEK Implementation Language (OIL)* neben den existierenden Systemobjekten auch deren Interaktionsmöglichkeiten [OSE04]. Ein Systemgenerator wandelt diese in entsprechende Systemobjekte um und generiert das dazu passende Betriebssystem [OSE05]. Gleiches gilt analog für den Nachfolgestandard AUTOSAR [AUT13].

Abbildung 2.1 zeigt eine Beispielanwendung bestehend aus zwei Aufgaben (`TASK`). Die Aufgabe `MEASURE` wird periodisch von einem Alarm gestartet und führt eine Messung durch. Liegt das Ergebnis vor, so aktiviert sie die zweite Aufgabe `STORE`, welche aufgrund ihrer höheren Priorität

```c
#include "os.h"

DeclareTask(MEASURE);
DeclareTask(STORE);
DeclareAlarm(A1);
DeclareCounter(C1);

TEST_MAKE_OS_MAIN(StartOS(0))

measurement_t measurement;
TASK(MEASURE) {
  prepare_measurement();
  measure_data(&measurement);
  ActivateTask(STORE);
  cleanup_measurement();
  TerminateTask();
}

TASK(STORE) {
  store_data(&measurement);
  TerminateTask();
}
```

(a) Programmcode

```
CPU Example {

    OS Example {
        STATUS = STANDARD;
        /* ... */
    };

    TASK MEASURE {
        PRIORITY = 1;
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
    };

    TASK STORE {
        PRIORITY = 2;
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
    };

    ALARM A1 {
        COUNTER = C1;
        ACTION = ACTIVATETASK {
            TASK = MEASURE;
        };
        AUTOSTART = TRUE {
            ALARMTIME = 100;
            CYCLETIME = 100;
        };
    };

    COUNTER C1 {
        MAXALLOWEDVALUE = 1000;
        TICKSPERBASE = 1;
        MINCYCLE = 1;
    };
};
```

(b) Betriebssystemkonfiguration

Abbildung 2.1 – Programmcode und Betriebssystemkonfiguration einer Beispielanwendung eines statisch konfigurierten OSEK-Systems. Die Anwendung besteht aus zwei Aufgaben. Die erste wird von einem Alarm periodisch aktiviert, führt eine Messung durch und aktiviert anschließend die zweite Aufgabe zum Abspeichern des Messergebnisses.

direkt eingelastet wird und das Messergebnis abspeichert. Alle Eigenschaften wie Prioritäten der Aufgaben und Periodizität des Alarms sind in der Konfigurationsdatei (Abbildung 2.1b) angegeben. Im Programmcode (Abbildung 2.1a) stehen nur die Funktionskörper der Aufgaben.

dynamisches Programmiermodell

Dem statischen Programmiermodell gegenüber steht das *dynamische Programmiermodell*. Bei diesem werden die Systemobjekte während der Laufzeit durch entsprechende Systemaufrufe erzeugt. Ihre Existenz ist also nicht explizit vor der Laufzeit bekannt, sondern ergibt sich aus dem ausgeführten Programmcode der Anwendung. Typischer Vertreter dieses Programmiermodells sind POSIX-konforme Systeme oder auch FreeRTOS und eCos [Pos; Bar10; Mas03; Bac+18]. Die Freiheit, beliebige Systemobjekte zur Laufzeit zu erzeugen, ermöglicht ein dynamisches Hinzufügen von

```
#include <FreeRTOSConfig.h>
#include <FreeRTOS.h>
#include <task.h>

TaskHandle_t measure_task;
TaskHandle_t store_task;

measurement_t measurement;
void measure_fn(void *param) {
  TickType_t last_wake_time = xTaskGetTickCount();
  TickType_t period = 100;
  while (1) {
    prepare_measurement();
    measure_data(&measurement);
    xTaskNotify(store_task, 1, eSetBits);
    cleanup_measurement();
    vTaskDelayUntil(&last_wake_time, period);
  }
}

void store_fn(void *param) {
  while (1) {
    if (ulTaskNotifyWait(0, 1, 0, portMAX_DELAY) == pdTRUE) {
      store_data(&measurement);
    }
  }
}

int main(int argc, char **argv) {
  xTaskCreate(measure_fn, "measure_task", 512, NULL, 1, &measure_task);
  xTaskCreate(store_fn, "store_task", 512, NULL, 2, &store_task);

  vTaskStartScheduler();
}
```

Abbildung 2.2 – Programmcode einer Beispielanwendung eines dynamisch konfigurierten FreeRTOS-Systems. Die Anwendung besteht aus zwei Fäden. Der erste führt periodisch eine Messung durch und benachrichtigt den zweiten Faden, sobald das Ergebnis vorliegt. Der zweite Faden wartet auf die Benachrichtigung und speichert den Messwert ab.

Softwarekomponenten zur Laufzeit. Ebenso ist es möglich, die Abarbeitung auftretender Ereignisse in jeweils einem eigenen Faden mit Laufzeitkontext auszuführen.

Abbildung 2.2 zeigt eine Beispielanwendung ähnlich zu Abbildung 2.1, jedoch implementiert mittels des dynamisch konfigurierten Betriebssystems FreeRTOS. Auch hier sind zwei Aktivitätsträger vorhanden, hier sind dies jedoch Fäden. Die Aufgaben sind jedoch gleich verteilt, sodass einer die Messung durchführt und dann den anderen benachrichtigt, dass das Ergebnis zum Speichern bereit ist. Im Gegensatz zu dem statisch konfigurierten Beispiel werden hier die Systemobjekte erst zur Laufzeit in der Einstiegsfunktion konfiguriert und erzeugt. Hier werden alle Eigenschaften der Systemobjekte als Parameter der jeweiligen instanziierenden Systemaufrufe an das Betriebssystem übergeben.

hybrides Programmiermodell

Eine Kombination beider Modelle erlauben Betriebssysteme nach dem *hybriden Programmiermodell*. Hierbei sind sowohl statische Konfigurationen der Betriebssystemobjekte über Konfigurationsdateien möglich wie auch die Erzeugung neuer Systemobjekte nach Bedarf zur Laufzeit.

```
1  #include <zephyr/kernel.h>
2
3  #define STACKSIZE 512
4
5  /* Statische Konfiguration von Systemobjekten */
6  K_SEM_DEFINE(measurement_available, 0, 1);
7
8  K_THREAD_DEFINE(measure_id, STACKSIZE, measure_fn, NULL, NULL, NULL, 1, 0, 0);
9
10 static struct k_thread store_thread;
11 K_THREAD_STACK_DEFINE(store_stack, STACKSIZE);
12
13 measurement_t measurement;
14
15 void measure_fn(void *param) {
16   while (1) {
17     prepare_measurement();
18     measure_data(&measurement);
19     k_sem_give(measurement_available)
20     cleanup_measurement();
21     k_sleep(100);
22   }
23 }
24
25 void store_fn(void *param) {
26   while (1) {
27     k_sem_take(measurement_available, K_FOREVER);
28     store_data(&measurement);
29   }
30 }
31
32 int main(int argc, char **argv) {
33   /* dynamische Konfiguration von Systemobjekten */
34   k_thread_create(&store_thread, store_stack,
35           K_THREAD_STACK_SIZEOF(store_stack),
36           store_fn, NULL, NULL, NULL,
37           2, 0, K_FOREVER);
38   k_thread_start(&store_thread);
39 }
```

Abbildung 2.3 – Programmcode einer Beispielanwendung eines hybrid konfigurierten Zephyr-Systems. Die Anwendung besteht aus zwei Fäden und einem Semaphor. Der erste Faden ist mittels Präprozessormakros statisch konfiguriert und verwendet den ebenfalls statisch konfigurierten Semaphor zur Signalisierung eines vorliegenden Messwertes. Der zweite Faden, welcher dynamisch konfiguriert wird, wartet auf das Vorhandensein des Messwertes und speichert diesen ab.

Die Betriebssystemspezifikation für μITRON [Ass10] sieht den Einsatz eines Konfigurators für statische Systemobjekte und das Betriebssystem vor, stellt aber ebenso Systemaufrufe bereit, die neue Systemobjekte erzeugen. Der Konfigurator interpretiert die in der Systemkonfiguration vorhandenen statischen Systemaufrufe und generiert daraus die entsprechenden Programmcodedateien, die den Betriebssystemkern initialisieren und konfigurieren. Systemobjekte werden hierbei automatisch erzeugt und ihnen eindeutige Identifikationsnummern zugewiesen [Ass10].

Das Betriebssystem Zephyr [Zep] verwendet den C-Präprozessor zur Erzeugung statischer Systemobjekte. Diese werden in speziell benannte Sektionen des Betriebssystemcodes geschrieben und

während des Bindens des Systemabbildes zusammengestellt. Parallel können über entsprechende Systemaufrufe auch zur Laufzeit weitere Systemobjekte erzeugt werden.

Abbildung 2.3 zeigt ebenfalls eine Beispielanwendung, die wie die vorherigen Anwendungen in den Abbildungen 2.1 und 2.2 Messwerte ermittelt und abspeichert. Hierbei sind der für die Messungen verwendete Faden und der zur Synchronisation verwendete Semaphor statisch konfiguriert. Ihre Eigenschaften sind in den entsprechenden Zeilen am Kopf der Datei statisch codiert und können vom entsprechenden Generator ausgewertet werden. Dies entspricht dem statischen Systemmodell. Für den Speicher-Faden hingegen sind die Eigenschaften wie beim dynamischen Modell als Aufrufparameter der instanziierenden Systemaufrufe codiert. Dementsprechend wird dieser erst zur Laufzeit angelegt und konfiguriert.

Auswahlkriterien

Das dynamische Programmiermodell ist ein Garant für große Flexibilität sowohl für Anwendungsentwickler als auch für die Umsetzung der Anwendungslogik. Bedarf es zur Abarbeitung einer Aufgabe weiterer Systemobjekte, so können diese situationsbedingt und angepasst an Laufzeitzustände neu erzeugt werden. Auch kann die Kommunikation und Interaktion abhängig von Laufzeitereignissen angepasst werden und so zu einer adaptiven Umsetzung führen.

Diese Flexibilität ist einem statischen Systemmodell inhärent nicht gegeben. Hier muss bereits während der Übersetzungszeit die Existenz von Systemobjekten vorbestimmt sein. Daraus resultiert jedoch die Möglichkeit zur feingranularen Konfiguration und Spezialisierung des Betriebssystems entsprechend den Anforderungen der auszuführenden Anwendung. Diese kann wiederum zu deutlich verbesserten Eigenschaften wie einer geringeren Laufzeit [Hof+15; DL17] und geringer Menge an Speicherbedarf führen [DL18].

Um die Vorteile beider Modelle zu vereinen, erscheint so die Verwendung eines Betriebssystemkerns als angebracht, der dem hybriden Ansatz folgt und beide Paradigmen unterstützt. Jedoch ist dies nicht das einzige Entscheidungskriterium für die Wahl eines Betriebssystems. Vielmehr spielen Faktoren wie Kompatibilität und Wiederverwendbarkeit bestehender Komponenten eine Rolle. Auch die Verfügbarkeit des Quellcodes, Lizenzmodelle, Verfügbarkeit technischer Unterstützung und Vorkenntnisse im Entwicklungsteam sind entscheidende Faktoren. Nicht zuletzt wird die Entscheidung über das zu verwendende Betriebssystem nicht von dem Entwicklungsteam selber beschlossen, sondern entstammt einer anderen Abteilung [EET19].

2.2.1.2 Konfigurierbarkeit

Konfigurierbarkeit beschreibt, wie feingranular die Anpassung des Betriebssystems an die Anforderungen einer gegebenen Anwendung vorgesehen ist. Dabei ist diese Konfigurierbarkeit in mehrere Ebenen einteilbar. Es kann dazu dieselbe Einteilung verwendet werden, die auch für Spezialisierungen gebraucht wird. Es ergeben sich dazu die drei auf Abbildung 2.4 dargestellten Ebenen: Abstraktionsebene, Instanzebene und Interaktionsebene [▷Fie+18].

generische Ebene

Ohne ein Vorhandensein von Konfigurierbarkeit beschreibt die generische Ebene den gesamten Umfang eines Betriebssystems. Hierbei sind alle unterstützten Merkmale vorhanden und verwendbar.

Dies entspricht der uneingeschränkten Betriebssystemschnittstelle, wie sie in einem Betriebssystemstandard oder einer Schnittstellendefinition beschrieben ist. Vertreter dafür sind beispielsweise POSIX [Pos], OSEK/AUTOSAR [OSE05; AUT13] oder ARINC [AEE03].

Da auf dieser Ebene noch keine Einschränkung durch eine Konfiguration oder Spezialisierung vorgenommen ist, werden alle Anwendungen auf dieser Ebene unterstützt. Abbildung 2.4(g) zeigt als Ausgangslage dementsprechend einen uneingeschränkten Interaktionsgraphen und alle Merkmale als verfügbar.

Abstraktionsebene

Konfigurierbarkeit auf Ebene der *Abstraktionen* ist die grobgranularste Ebene. Auf dieser Ebene lassen sich ganze Funktionsblöcke aktivieren oder austauschen. Die Implementierung der Konfi-

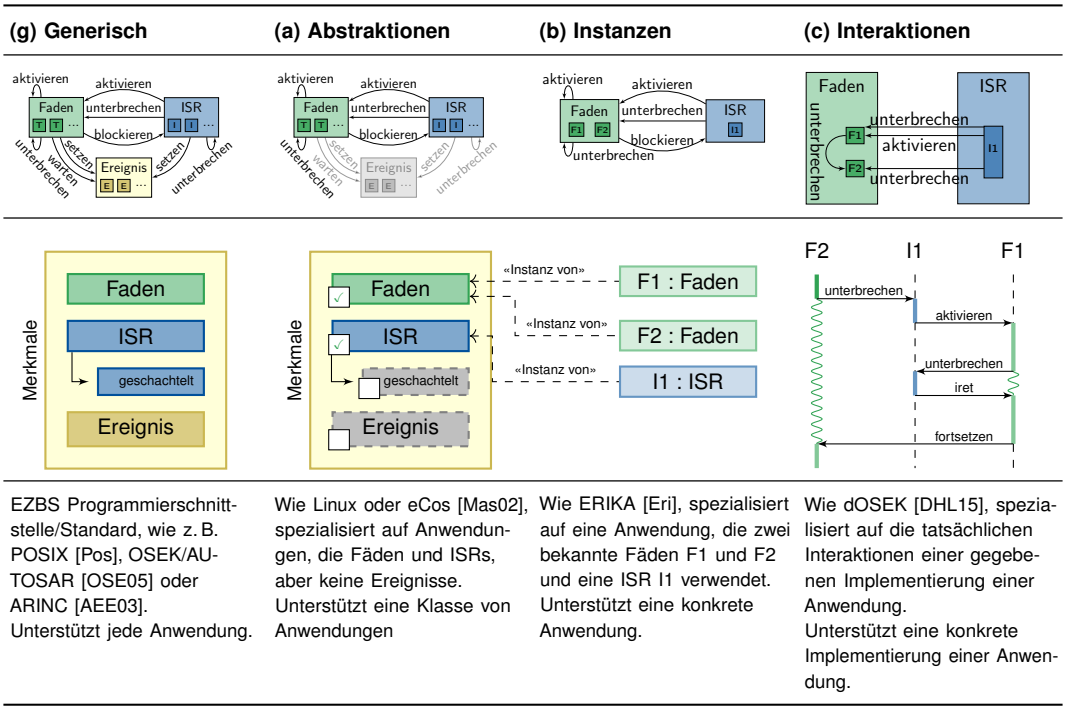

Abbildung 2.4 – Ebenen der Konfigurierbarkeit und Spezialisierung von Betriebssystemen. Der Grad steigt von der generischen Variante links bis hin zu der Ebene der Interaktionen rechts stetig an. Spezialisierungen auf den Ebenen der Instanzen (b) und der Interaktionen (c) sind Thema dieser Arbeit. Aus [▷Fie+18]

gurierbarkeit auf dieser Ebene wird je nach Betriebssystem unterschiedlich umgesetzt. FreeRTOS, Linux und eCos [Mas02] bieten Konfigurationsmöglichkeiten auf dieser Ebene. Sie verwenden den Präprozessor zur Umsetzung. Systeme nach dem OSEK/AUTOSAR-Standard spezifizieren die verwendeten Abstraktionen in einer domänenspezifischen Sprache in einer gesonderten Konfigurationsdatei. Bei anderen Systemen, wie auch bei FreeRTOS wird durch Hinzufügen oder Weglassen ganzer Übersetzungseinheiten auf Abstraktionsebene konfiguriert.

Konfigurieren auf Abstraktionsebene hat zur Folge, dass ganze Funktionalitäten für das Gesamtsystem (nicht) zur Verfügung stehen. Abbildung 2.4(a) zeigt beispielhaft ein Betriebssystem, dass Fäden, Ereignisse, Warteschlangen und Unterbrechungsbehandlungsroutinen anbietet. Durch Konfiguration auf Abstraktionsebene können bei diesem System beispielsweise Ereignisse auskonfiguriert werden. Ein so konfiguriertes System kann Ereignisse nicht mehr als Programmiermittel verwenden, muss aber auch keinen unnötigen Aufwand in dessen Implementierung investieren. Damit wird nur noch eine Klasse von Anwendungen unterstützt, deren verwendeter Funktionsumfang Teilmenge der aktivierten Abstraktionen ist. Dies wird durch den teilweise ausgegrauten Interaktionsgraphen dargestellt. Für Anwendungen, die keine Ereignisse verwenden, ist dies eine gültige Konfiguration eines spezialisierten Betriebssystems. Alle funktionalen Anforderungen sind weiterhin erfüllt, jedoch ist die nichtfunktionale Eigenschaft des Speicherverbrauchs dadurch verbessert, dass kein Programmcode für Ereignisse mehr benötigt wird.

Konfigurierbarkeit auf *Instanzebene* bedeutet, dass zur Übersetzungszeit bereits die verwendeten Instanzen der verwendeten Abstraktionen vorgegeben werden können. Am Beispiel der Abbil- *Instanzebene*

dung 2.4(b) bedeutet dies, dass die Anwendung zwei Fäden und eine ISR verwendet. Ereignisse sind nicht vorhanden. Dementsprechend enthält der Interaktionsgraph genau diese Systemobjekte, es sind jedoch weiterhin alle Interaktionen zwischen diesen möglich. Als Resultat kann ein so konfiguriertes System nur eine konkrete Anwendung, deren Interaktionsgraph eine Teilmenge des konfigurierten ist, ausführen.

Dieser Grad der Konfigurierbarkeit ist nur bei Betriebssystemen mit statischer Konfigurations- und Programmierschnittstelle zu finden. Zu deren Vertretern gehören OSEK/AUTOSAR und Zephyr.

Bei Systemen aus der OSEK/AUTOSAR-Familie ist enthält die OIL-Spezifikation des Systems eine Beschreibung aller Instanzen von Systemobjekten. Hierbei sieht der dazugehörige Standard explizit die Verwendung eines Betriebssystemgenerators vor, der anhand der Systemspezifikation nur die benötigten Bestandteile des Betriebssystems generiert [OSE05]. Dies ermöglicht eine direkte Anpassung an die Anforderungen der auszuführenden Anwendung und wird z. B. von den Betriebssystemimplementierungen ERIKA [Eri] und Sloth [Hof+09] umgesetzt.

Bei Systemen mit dynamischer Instanziierung von Systemobjekten wie FreeRTOS oder POSIX-konformen Systemen ist diese Konfiguration auf Instanzebene nicht möglich, da erst während der Laufzeit des Systems die Systemobjekte erzeugt werden. Dementsprechend ist ohne Weiteres keine Spezialisierung auf Instanzebene für diese Systeme möglich. An dieser Stelle setze ich mit dieser Arbeit an und stelle Möglichkeiten vor, auf diese Systeme auf Ebene der Instanzen zu spezialisieren. Das notwendige Anwendungswissen ermitteln dabei statische Analysen [ENL22; ▷Fie+21].

Betriebssysteme wie Zephyr wählen an der Stelle einen Zwischenweg. Hier existieren sowohl Systemaufrufe für die dynamische Erzeugung von Systemobjekten wie auch geeignete Konstruktoren zur statischen Erzeugung dieser während des Übersetzungsvorgangs. Der Programmierer kann so je nach Anforderung das geeignete Mittel wählen.

Interaktionsebene

Die *Interaktionsebene* ermöglicht die feingranularste Konfiguration innerhalb dieser Hierarchie. Hierbei ist neben dem Wissen über die vorhandenen Instanzen auch notwendig zu wissen, wie deren Interaktionsmuster sind. Abbildung 2.4(c) stellt dazu exemplarisch den Interaktionsgraphen einer Anwendung dar, die zwei Fäden verwendet. Hierbei darf Faden F1 zwar F2 unterbrechen, jedoch nicht andersherum. So sind die jeweils entgegengesetzten Systemaufrufe überflüssig. Hierdurch können Synchronisationen vereinfacht und Systemaufrufergebnisse vorausberechnet werden. Ein so spezialisiertes Betriebssystem unterstützt nur noch eine konkrete Implementierung einer Anwendung.

Dieses detaillierte Wissen auf Interaktionsebene ist nur bei wenigen Betriebssystemmodellen explizit spezifiziert. In der Spezifikation eines OSEK/AUTOSAR-Systems wird beschrieben, welche Ereignisse von welchem Faden gesetzt und konsumiert werden dürfen. Obwohl Zephyr die statische Konfiguration auf Instanzebene ermöglicht, ist jedoch keine statische Spezifikation der Interaktionen vorgesehen. Mangels bekannter Systemobjekte sind auch bei POSIX-konformen Systemen oder anderen Systemen mit dynamischer Instanziierung wie FreeRTOS keine expliziten Modellierungen der Interaktionen vorgesehen.

Für statisch konfigurierte Systeme führen bestehende Arbeiten [Die+19; DL17; Die19; Sch11] bereits verschiedene Spezialisierungen auf Ebene der Interaktionen zur Übersetzungszeit durch. Ebenfalls existieren bereits Spezialisierungen auf Interaktionsebene [PMI88] durch, dies geschieht jedoch während der Laufzeit. In dieser Arbeit werde ich Spezialisierungen für dynamisch konfigurierte Systeme auf Ebene der Interaktionen vorstellen, die bereit zur Übersetzungszeit angewendet werden.

2.3 Stand der Wissenschaft

Bevor wir in die Betrachtung der in Abschnitt 1.4 angesprochenen Ziele und Vorgehensweisen dieser Arbeit in den Kapiteln 4 und 5 einsteigen, gebietet sich ein Blick in die bisher auf den berührten

Gebieten erreichten Erfolge und Erkenntnisse. Verlagerung von Laufzeitberechnungen in die Übersetzungszeit ist eine der Kernaufgaben optimierender Übersetzer. Diese haben bereits früh Einzug in Übersetzer gefunden, um im Rahmen der Semantik der jeweiligen Programmiersprache Ausdrücke zusammenzufassen [ASU86]. Für die Generierung und Spezialisierung von Betriebssystemen ist zudem die Ermittlung von Wissen über die Anwendung notwendig. Dieses Wissen kann sowohl aus expliziten Systembeschreibungen wie auch aus der Analyse des Anwendungscodes oder der Durchführung eines Referenzlaufes der Anwendung gewonnen werden. Schließlich wird das gewonnene Wissen verwendet, um das Betriebssystem zu spezialisieren.

2.3.1 Optimierende Übersetzer: Verlagerung von Laufzeitberechnungen in die Übersetzungszeit

Bereits in frühen Übersetzern waren Mechanismen zur Verlagerung von Laufzeitberechnungen in die Übersetzungszeit integriert. Wulf u. a. beschreibt bereits 1973 die Übersetzung von Hochsprachencode insbesondere für Systemsoftware in effizient ausführbaren Maschinencode als wichtige Grundlage. Der dazu entwickelte Übersetzer für die Programmiersprache Bliss/11 fasst berechenbare arithmetische Ausdrücke bereits zur Übersetzungszeit zusammen und verlagert invariante Berechnungen aus Schleifen heraus [Wul+73].

Hinzu kommen Techniken zum Verlagern von Funktionsaufrufen kleiner Funktionen hin zu direkter Einbettung des Funktionsrumpfes in die Aufrufstelle (engl. inlining). Dieses führt zur Reduktion der Ausführungszeit. Dabei wird ausgenutzt, dass die Kosten für den Funktionsaufruf und die damit verbundene Sicherung und Wiederherstellung von Registern eingespart werden kann. Zudem vereinfacht es weitere Analysen über die Lebendigkeit und Verwendung von Variablen, da so optimierte Funktionsaufrufe nicht mehr als undurchsichtig angesehen werden müssen und etwa unverwendete Parameter nicht nutzlos aufbereitet werden müssen [RG89]. Gleichzeitig führt die Einbettung jedoch auch zu leichter Erhöhung der Menge des Programmcodes durch die Duplikation an mehrere Aufrufstellen. Dies kann sich in der Gesamtbilanz der Systemlaufzeit nachteilig auswirken, da möglicherweise mehr Worte aus langsamen Speichern geladen werden müssen. Im Allgemeinen überwiegt jedoch der Geschwindigkeitsvorteil bei Anwendung geeigneter Metriken zur Entscheidung [DH92; Cha+92; CHT91; CHT92].

Während diese Formen der Optimierung ursprünglich nur innerhalb von Übersetzungseinheiten angewendet wurden, führt der Einsatz von Optimierern zur Bindezeit zu der Möglichkeit, diese Optimierungen auch über Übersetzungseinheitsgrenzen hinweg auf das Gesamtsystem anzuwenden. Hierzu ist es notwendig, Hochspracheninformationen nicht verfrüht zu verwerfen, da diese für die Optimierung zur Bindezeit benötigt wird [LA04]. Übersetzer wie LLVM oder GCC speichern dazu ihre Zwischencoderepräsentation anstelle oder zusätzlich zu dem Maschinencode in den übersetzten Programmdateien.

Wichtig bei der statischen Optimierung von Programmcode ist, dass damit immer ein Abwägen zwischen Laufzeit und Speicherbedarf verbunden ist. Techniken wie Einbetten von Funktionsrümpfen anstelle von Funktionsaufrufen oder Ausrollen von Schleifen kann neben einer Beschleunigung auch zu einer Vergrößerung des benötigten Speicherbedarfs führen. Hierbei werden Modelle und Heuristiken zur Abschätzung dieser Auswirkungen verwendet [BGS94; GH01]. Alternativ gibt es auch Ansätze zur vollständigen Exploration des Konfigurationsraums zur Übersetzungszeit durch Ausführen von Übersetzungsvorgängen. Diese führen zwar zu besseren Ergebnissen, bedürfen aber erhöhter Übersetzungszeit für die Ausführung [Tri+03].

2.3.2 Beschleunigen des Systemstarts

Die Dauer des Systemstarts hat Einfluss auf die Gesamtleistung eines Systems und ist daher Thema verschiedener Forschungsfelder.

Eine bewährte Technologie ist die, einen regulären Systemstart durchzuführen und dessen Ergebnis zur späteren Wiederherstellung zu sichern. Dieser Ansatz führt zu einer Reduktion der Startzeit im Umfeld von serverlosen Funktionen um bis zu 1932 % [SFP20]. Das Duplizieren einer bereits laufenden Instanz einer Anwendung ermöglicht so eine Startzeit unter einer Millisekunde unter gleichzeitiger Beibehaltung starker Isolationsgarantien [Du+20].

Auch auf eingebetteten Systemen findet der Ansatz, einen Systemzustand zu speichern und wiederherzustellen, Anwendung. Hier steht jedoch der Energieverbrauch im Fokus. Bei Systemen mit nicht dauerhaft sichergestellter Energieversorgung müssen Zwischenstände gesichert werden, sobald die Energieversorgung nicht sichergestellt ist [UM21]. Ist die Versorgung wieder vorhanden, soll möglichst ohne Verluste an der zuvor unterbrochenen Stelle die Arbeit des Systems fortgesetzt werden. Die Dauer des Wiederherstellens bedeutet hierbei Energieverbrauch, der nicht zum eigentlichen Systemziel beiträgt, also Optimierungsziel ist [KSS20]. Ein kombinierter Ansatz aus Hardware und Software führt zu weiteren Verbesserungen für Energieverbrauch und Wiederherstellungszeit, sodass eine Wiederherstellung innerhalb von 21μs möglich ist [Jay+15]. Die gesicherten Systemzustände eröffnen jedoch neue Angriffsmöglichkeiten, da diese Ziel von Manipulationen werden können [Kri21].

Das Erstellen eines solchen Zustandes bedeutet jedoch, dass eine zeitliche Vermischung von Code, der bereits Teile der Anwendungslogik ausführt und dem Startcode nicht möglich ist. Es muss einen definierten Punkt im Verlauf des Programmflusses geben, der Initialisierung von Datenverarbeitung aktueller Umgebungsinformationen trennt. Bei den in dieser Arbeit vorgestellten Mechanismen hingegen sind auch teilweise verzögerte Startvorgänge beschleunigbar und es muss keine scharfe Trennung zwischen Startvorgang und Arbeitsbetrieb mit Reaktion auf Umwelteinflüsse geben.

2.3.3 Anwendungswissen ermitteln und Betriebssystem spezialisieren

Für die Spezialisierung des Betriebssystems hinsichtlich der Anforderungen der Anwendung ist es notwendig, Wissen über diese zu sammeln. Hierzu können verschiedene Informationsquellen wie der Quellcode oder Laufzeitaufzeichnungen herangezogen werden. Die Spezialisierung lässt sich dann während der Ausführung des Systems nach Bedarf durchführen oder anhand eines Referenzlaufes vor der Ausführung.

Handelt es sich um ein Betriebssystem mit statischem Programmiermodell, so ist die erste Informationsquelle die Systemspezifikation. Der OSEK-Standard sieht diese explizit zur Verwendung vor [OSE05]. Aus dieser Systemspezifikation kann Wissen über die Instanzen der Systemobjekte extrahiert werden. Für detaillierteres Wissen über die Verwendung des Betriebssystems sind jedoch weitere Analysen notwendig, die sich in Übersetzungszeitanalysen des Programmcodes, Laufzeitanalysen eines Referenzlaufes und Laufzeitanalysen des aktuellen Programmlaufs teilen.

dynamische Systemmodelle

Betrachten wir zunächst Ansätze zur Spezialisierung von Systemen mit dynamischer Erzeugung von Systemobjekten. Obwohl diese keine Spezifikation der existierenden Systemobjekte bereitstellen, ist eine Spezialisierung gewinnbringend möglich.

Pu, Massalin und Ioannidis präsentieren mit SYNTHESIS ein Betriebssystem, das nach Bedarf zur Laufzeit optimierte Systemobjekte bei der Erzeugung von Systemobjekten generiert, die auf dieses zugeschnitten sind. Dies geschieht durch die Verwendung teilgebundener Funktionen, bei denen die Referenzen an die jeweiligen Systemobjekte bereits festgeschrieben sind. Hierdurch ist es möglich, die Ausführungszeit dieser Systemaufrufe zu reduzieren. Diese Spezialisierung geschieht

entsprechend dem dynamischen Programmiermodell vollständig zur Laufzeit des Systems. Die Spezialisierung ist so bis auf Ebene der Interaktionen möglich, da die Systemaufrufe direkt mit den Systemobjekten verbunden werden. Jedoch ist weiterhin der Laufzeitaufwand für die generische Objekterzeugung notwendig. Zusätzlich sind auch der Übersetzer und der Optimierer für die neu generierten Systemaufrufe zur Laufzeit notwendig, die zusätzliche Laufzeitkosten bei der Erzeugung von Systemobjekten hervorrufen [PMI88].

Bei der durch Ruprecht, Heinloth und Lohmann [RHL14] vorgestellten Methode FLIPPER wird anhand von Referenzläufen der benötigte Funktionsumfang des Betriebssystems auf Ebene der Abstraktionen durch mitschneiden der aufgerufenen Systemaufrufe und deren ausgeführter Pfade innerhalb der C-Standardbibliothek und des Linux-Betriebssystemkerns ermöglichen den Aufbau von aussagenlogischen Modellen über benötigte Abstraktionen dieser. Darauf aufbauend kann ein maßgeschneiderter Betriebssystemkern erzeugt werden, der bis zu 70 Prozent weniger Speicherplatz benötigt [RHL14]. Aufgrund des gewählten Ansatzes findet die Spezialisierung nur auf Ebene der Abstraktionen statt. Hierbei werden die automatisch nur die verwendeten Komponenten des Betriebssystems selektiert und dabei das bereits vielfältige Angebot an spezialisierten Implementierungen verwendet. So entsteht zur Übersetzungszeit ein an die auszuführende Anwendung angepasstes Betriebssystem unter Ausnutzung von Laufzeitwissen.

Der gleiche Ansatz des Mitschneidens verwendeter Programmabschnitte führt auch bei der Anwendung auf Standardbibliotheken zu einer deutlichen Reduktion der enthaltenen Funktionen und dadurch ebenfalls zu einer Speicherreduktion bis zu 17 Prozent [Zie+19]. Die Spezialisierung der Implementation wird hierbei jedoch auf Ebene des fertig übersetzten Maschinencodes vorgenommen. Dadurch ist auch die Anwendung auf nicht im Quellcode vorliegende Bibliotheken möglich. Mururu u. a. hingegen arbeitet dynamisch während der Laufzeit und blendet die jeweils benötigten Funktionalitäten der Systembibliotheken entsprechend ein. Dies geschieht dynamisch zur Laufzeit der Anwendung mithilfe eines Prädiktors, der anhand der Aufrufstelle den zu erwartenden Programmcode für den voraussichtlich folgenden Aufrufbaum ermittelt. Eine Erzeugung oder Auswahl angepasster Implementierungen ist jedoch bei beiden nicht vorgesehen, es wird nur bestehender ungenutzter Maschinencode auf Sektionsgranularität entfernt.

Im Gegensatz dazu führt Heinloth u. a. eine Spezialisierung des Betriebssystems anhand der vorliegenden Hardwarekomponenten während des Systemstarts durch. Hierzu wird zunächst ein Minimalbetriebssystem gestartet, welches nur die zum Übersetzen des Zielbetriebssystems notwendigen Komponenten enthält. Dieses analysiert die gegebene Hardwareplattform, spezialisiert das Betriebssystem durch Erstellen einer angepassten Konfiguration und übersetzt das Betriebssystem aus dem Quellcode. Dies resultiert in einer Verwendung von Implementierungen, die beispielsweise das Vorhandensein mehrerer Prozessoren nur bei Bedarf unterstützt. Dabei wird das gesamte Optimierungspotenzial des verwendeten Übersetzers ausgenutzt, da architektur- und hardwareabhängige Informationen gegeben sind. Damit fallen momentan durch Betriebssystementwickler handoptimierte Maschineninstruktionen für zeitkritische Operationen wieder in den Zuständigkeitsbereich von Übersetzern, deren Kernaufgabe es ist, diese zu optimieren [Hei+19]. Die auszuführende Anwendung wird bei diesem Ansatz nicht mit betrachtet. Sie profitiert nur indirekt durch die allgemeine Verbesserung des Betriebssystems.

Einen anderen Ansatz wählten Engler, Kaashoek und O'Toole für die Betriebssystemarchitektur EXOKERNEL. Hierbei werden nur die minimal notwendigen Aufgaben der Isolation und des Schutzes im vorgestellten Betriebssystem implementiert. Alle anderen Aufgaben wie Verwaltung der Ressourcen sind Aufgabe der Anwendung, die hierzu nach Bedarf eigene Bibliotheken mitbringen kann. Somit ist die Verantwortung für die optimale anwendungsangepasste Umsetzung von Betriebssystemfunktionalitäten in den Anwendungsbereich verschoben. Das Betriebssystem ist nun Teil der Anwendung und kann mit dieser durch Bindezeitoptimierungen angepasst werden

[EKO95]. Jedoch sind so nur automatisierte Optimierungen anhand der Semantik der verwendeten Programmiersprache möglich. Eine Betrachtung der Instanzen und Interaktionen dieser miteinander ist nicht dadurch gegeben.

Neben der dynamischen Erzeugung von Softwareimplementierungen gibt es auch die Möglichkeit, rekonfigurierbare Hardware zu verwenden. Bei RECONOS werden unter anderem Fäden zur Laufzeit in entsprechend konfigurierte Hardwarekomponenten in einem FPGA ausgelagert [LP09; LP07]. Dabei ist die Programmierschnittstelle einheitlich für Software- und Hardwarevarianten derselben Betriebssystemabstraktion. Dies ermöglicht eine gleichartige Verwendung unabhängig von der dahinterstehenden Implementierung. Im Gegensatz zu den von mir vorgestellten Spezialisierungen finden diese jedoch zur Laufzeit und nicht bereits zur Übersetzungszeit statt. Das bedeutet, dass zusätzliche Mehrarbeit während der Laufzeit notwendig ist, die das zeitliche Verhalten des Gesamtsystems beeinflusst.

Bertran u. a. hingegen analysiert den globalen Kontrollfluss der Anwendung über Bibliotheken und Systemaufrufe hinweg und streicht nicht verwendete Funktionalitäten aus dem Quellcode. Dies verringert die Größe des Programmabbildes um bis zu 54 % [Ber+06]. Im Gegensatz zu dieser Arbeit werden jedoch keine Spezialisierung der Implementierung des jeweiligen Systemaufrufs und keine statische Vorausberechnung der Effekte durchgeführt. Die Auswirkungen aller Systemaufrufe werden weiterhin zur Laufzeit evaluiert.

statische Systemmodelle

Erleichtert wird die Spezialisierung des Betriebssystems bei statischen Betriebssystemmodellen. Diese können zusätzlich Wissen über die Instanzen der Systemabstraktionen verwenden.

Der OSEK Standard sieht herzu explizit die Verwendung eines Systemgenerators vor, der anhand der in der OIL vorliegenden Systemspezifikation das Betriebssystem zur Verwendung dieser Systemobjekte generiert. Das Modell sieht dabei auch die Beschreibung der potenziellen Interaktionen mittels Auflistung der erlaubten Zugriffe auf Betriebsmittel und Kommunikationskanäle vor [OSE05].

Diese Anpassung des Betriebssystems kann nicht nur unter Verwendung von Anwendungswissen vorgenommen werden. Zusätzliche Einbeziehung von Hardwarekomponenten ermöglicht die direkte Abbildung von Betriebssystemfunktionen auf diese.

Eine Methode dazu ist die Verlagerung ganzer Funktionseinheiten von der Software in die Hardware. HybridThreads [Agr+06] verlagert dazu Prozesseinplanung und Synchronisationsprimitiven in ein an den Prozessor angebundenen FPGA. Bei FlexPRET [Zim+14] werden Kontextinformation von Fäden in entsprechender Hardware zur Implementierung zyklengenauer Prozessplanung verwaltet. Dabei geschieht keine weitere Spezialisierung auf Ebene der Instanzen oder Interaktionen.

Hofer u. a. nutzt die Unterbrechungssteuereinheit (engl. interrupt controller) für die Implementierung der Prozesseinplanung und -einlastung [Hof+09]. Damit werden die bisher nur in Software existierenden Fäden direkt auf die verwendete Hardware abgebildet und diese so effektiver ausgenutzt. Neben dem reduzierten Programmcode führt dies auch zu einer deutlichen Reduktion der Latenz und deren Schwankungsbreite. Zudem wird so das Problem der Prioritätsumkehr zwischen niederprioren Unterbrechungen und hochprioren Prozessen durch einen einheitlichen Prioritätenraum gelöst [Hof14]. Dies funktioniert für statisch konfigurierte ereignisgesteuerte Systeme [Hof+09] mit passivem Warten [HLSP11] und für zeitgesteuerte Systeme [Hof+12]. Ebenso existiert Mehrkernunterstützung für global partitionierte Systeme [Mül+14], die entsprechende Hardware für kernübergreifende Planung und Einlastung [BDL22] und Speicherschutzkonzepte [Dan+14].

Für eine Spezialisierung auf Ebene der Interaktionen ist ein tiefgreifendes Verständnis der Abläufe und Anforderungen der Anwendung notwendig. Der REAL-TIME SYSTEMS COMPILER ist ein Werkzeug zur Transformation eines ereignisgesteuerten Systems in ein zeitgesteuertes System. Hierbei bleiben alle funktionalen und insbesondere auch die zeitlichen Anforderungen erhalten und erfüllt, insgesamt wird jedoch ein Paradigmenwechsel der Betriebssystemarchitektur erreicht. Grundlegende Einheit von Information ist dabei der Atomare Basisblock (ABB) (siehe Abschnitt 3.2.1) als Baustein für den

Kontrollflussgraphen der Anwendung und deren zeitliche Anforderungen. Der daraus aufgebaute Kontrollflussgraph kann, optional mit Abhängigkeiten und Zeitvorgaben konnotiert, sowohl auf zeitgesteuerten wie auch auf ereignisgesteuerten Betriebssystemen ausgeführt werden. Damit ist es möglich, die Anforderungen und die Implementierung unabhängig von dem Paradigma der späteren Ausführungsumgebung umzusetzen, da diese automatisch angepasst werden kann [Sch11].

Zusätzlich zu dem Anwendungswissen aus der Systemspezifikation sind weitere Spezialisierungen durch systematisches Analysieren des Programmcodes möglich. Ein gezieltes Codieren von Systemaufrufen und Einbetten dieser und deren statisch berechenbarer Resultate in den Anwendungscode führt zu einer höheren Robustheit gegenüber transienten Fehlern [Hof+15]. Insgesamt verbessert dies die Möglichkeit, Zusicherungen über die Integrität des Kontrollflusses und der Verwendung des Betriebssystems durch die Anwendung [Hof16].

Eine betriebssystem- und interaktionsgewahre Analyse des Kontrollflusses durch Anwendung und Betriebssystem ermöglicht eine detaillierte Modellbildung über die und explizite Aufzählung der möglichen Betriebssystemzustände [Die19]. Diese Information ermöglicht die Bestimmung eines endlichen Automaten, der die validen Betriebssystemzustandsübergänge abbildet. Dieser endliche Automat kann entsprechend als spezialisierte Betriebssystemimplementierung in Hardware generiert werden, um die Auswirkungen der Systemaufrufe direkt in die Arbeitsschritte des Prozessors zu integrieren. Dieser führt zu einer Reduktion der Latenzzeiten und deren Schwankungsbreite von Systemaufrufen. Gleichzeitig wird durch diese Vorausberechnung möglicher valider Übergänge zwischen Systemzuständen die Vermeidung falscher Übergänge verbessert, da bereits zur Übersetzungszeit die validen Pfade durch den Betriebssystemzustandsraum feststehen und dies durch die Hardware erzwungen wird [DL17].

Neben der Spezialisierung des Betriebssystems kann detailliertes Anwendungswissen auch zur Spezialisierung der Anwendung verwendet werden. Durch gezieltes Verwenden eines geteilten Laufzeitstapels erzielen SEMI-EXTENDED TASKS [DL18] eine Reduktion des Arbeitsspeicherbedarfs für statisch konfigurierte eingebettete Echtzeitsysteme. Ermöglicht wird dies durch die Analyse der möglichen Aufrufpfade unter Zuhilfenahme der Betriebssystemsemantik hinsichtlich des Blockierens von Systemaufrufen [DL18].

Insbesondere die kombinierte Betrachtung von Anwendungs- und Betriebssystemkontrollfluss ermöglicht hierbei eine feingranulare Wissensermittlung gegenüber einer getrennten Betrachtung der Komponenten. Dies führt zu deutlich präziseren Aussagen über das Zeitliche und energetische Verhalten des Gesamtsystems [Die+17; Wäg+18] und verbessert damit die Spezialisierbarkeit [Die19].

2.4 Zusammenfassung

Eingebettete Echtzeitsysteme und insbesondere deren Betriebssystem sind elementare Bausteine bei der Herstellung komplexer Systeme. Die Aufgabe des Betriebssystems ist hierbei, den reibungslosen und korrekten Ablauf der auszuführenden Anwendung sicherzustellen. Das Betriebssystem erfüllt hierbei keinerlei Selbstzweck, sondern ist dabei ausschließlich als Werkzeug einer konkreten Anwendung tätig. Dementsprechend entsteht Raum für die Spezialisierung des Betriebssystems hinsichtlich der Anforderungen der gegebenen Anwendung. Spezialisierung bedeutet hierbei ein Verringern des nicht benötigten Funktionsumfangs, sodass das beobachtbare funktionale Verhalten des Gesamtsystems sich nicht verändert. Nichtfunktionale Eigenschaften wie Speicherbedarf, Laufzeit oder auch Sicherheit und Fehlertoleranz werden hierdurch jedoch verbessert.

Spezialisierung ist unterscheidbar anhand ihrer Tiefe, in der sie in das Programmverhalten und dessen Strukturen eingreift. Auf Ebene der Abstraktionen werden Merkmale (Features) konfiguriert,

sodass ein Betriebssystem beispielsweise Alarme unterstützt oder nicht. Auf Ebene der Instanzen werden die konkreten Inkarnationen einer Abstraktion behandelt. Ein Betriebssystem kann so beispielsweise die Menge der Threads bestehend aus T1 und T2 verwalten. Auf Ebene der Interaktionen ist auch vorgegeben, welche Instanzen wie miteinander interagieren. So könnte Thread T1 sich selber und T2 beenden, T2 aber nur sich selber.

Für Betriebssysteme mit dynamischer Programmierstelle, die beliebiges Erzeugen und verändern von Instanzen zur Laufzeit ermöglicht, führen bisherige Ansätze die Spezialisierung entweder nur auf Ebene der Abstraktionen aus [RHL14; Zie+19; Hei+19] oder sie führen die Spezialisierung während der Laufzeit des Systems aus [PMI88]. Bei statisch konfigurierten Systemen ist auch die Spezialisierung auf Instanzebene [OSE05; Hof+09] und auf Interaktionsebene [DL17; DL18] Bestandteil bestehender Forschung.

Optimierende Übersetzer führen neben der Optimierung des Anwendungscodes auch zu der Optimierung des Betriebssystemcodes, jedoch immer nur unter der Betrachtung der Semantik der verwendeten Programmiersprache. Die Semantik der Systemaufrufe bleibt hierbei unbeachtet. Wie Arbeiten zu Systemen mit statischer Programmierschnittstelle bereits gezeigt haben [Hof+15], führt das Betrachten dieser und daraus eventuelle Einbetten oder Spezialisieren dieser zu Verbesserungen der nichtfunktionalen Eigenschaften. In den folgenden Kapiteln wird dementsprechend die Spezialisierung von Systemaufrufen dynamischer Systeme und die Vorausberechnung deren vormals dynamischer Systemaufrufe behandelt.

3

Voraussetzungen

Ermittlung des notwendigen Anwendungswissens

Bevor wir uns in den folgenden Kapiteln der Umsetzung der Spezialisierung widmen, benötigen wir Wissen über das zu spezialisierende System. Dieses Kapitel behandelt daher verschiedene Möglichkeiten der Anwendungsanalyse und gibt einen detaillierten Einblick in die verwendete Methodik, auf deren Ergebnissen die folgenden Spezialisierungsvorhaben aufbauen.

Quellcode 3.1 Programmcode der Beispielanwendung eines FreeRTOS Systems mit mehreren mittels Synchronisationsobjekten kommunizierenden Fäden. Das System ermittelt die Werte diverser Sensoren, führt diese in einen fusionierten Wert zusammen und schreibt diesen in einen Speicher. Farblich hervorgehoben sind die Systemaufrufe zum Erzeugen von Systemobjekten (gelb), für die Interaktion (blau) und die Übergabe der Kontrolle über Datenstrukturen und Aktivitäten an das Betriebssystem (rot).

```
1  template <class DT> class GuardedData {
2   public:
3    virtual void get(DT *d) {
4      xSemaphoreTake(data_mutex, BLOCK_FOREVER);
5      *d = internal_data;
6      xSemaphoreGive(data_mutex);
7    }
8    virtual void set(DT *d) {
9      xSemaphoreTake(data_mutex, BLOCK_FOREVER);
10     internal_data = *d;
11     xSemaphoreGive(data_mutex);
12   }
13   GuardedData() {data_mutex = xSemaphoreCreateMutex();}
14  private:
15   SemaphoreHandle_t data_mutex;
16   DT internal_data;
17 };
18
19 QueueHandle_t storage_queue;
20 TaskHandle_t fusion_task;
21 TaskHandle_t storage_task;
22
23 int main() {
24   SensorInfo_t *si = detect_sensors();
25   xTaskCreate(fusion_entry, "fusion_task", 512, si, 1, &fusion_task);
26   for (int i=0; i < si->length; i++) {
27     xTaskCreate(sensors_entry, "sensor_task", 512, &si->sensors[i], 2,
28                 &(si->sensors[i]->task));
29     si->sensors[i]->data = new GuardedData<SensorData_t>();
30   }
31   vTaskStartScheduler();
32 }
33
34 void fusion_entry(void *param) {
35   SensorInfo_t *si = (SensorInfo_t*) param;
36   storage_queue = xQueueCreate(sizeof(message_t), 4);
37   xTaskCreate(storage_entry, "storage_task", 512, NULL, 3, &storage_task);
38   while (1) {
39     SensorData_t result = do_fusion(si);
40     xQueueSend(storage_queue, &result, BLOCK_FOREVER);
41   }
42 }
43
44 void storage_entry(void *param) {
45   init_storage();
46   while (1) {
47     SensorData_t data;
48     xQueueReceive(storage_queue, &data, BLOCK_FOREVER);
49     storage_write_data(&data);
50   }
51 }
52
53 void sensor_entry(void *param) {
54   Sensor_t *sensor = (Sensor_t*) param;
55   while (1) {
56     SensorData_t data = sensor->function(sensor);
57     sensor->data->set(& data);
58   }
59 }
```

3.1 Welches Wissen wird benötigt und wie wird es gewonnen?

Bevor wir uns der Fragestellung widmen, wie wir an Wissen über die gegebene Anwendung gelangen, muss zunächst festgelegt werden, welches Wissen für die nachfolgenden Optimierungen und Spezialisierungen benötigt wird. Die einfachste Antwort auf diese Frage wäre: der Aufrufkontext und die Aufrufparameter aller Systemaufrufe, die innerhalb der Anwendung getätigt werden und die daran beteiligten Systemobjekte. Die Systemaufrufe sind diejenigen Stellen der Anwendung, die das Kommunikations- und Interaktionsverhalten der Anwendung sowohl zwischen ihren eigenen Komponenten als auch gegenüber dem Betriebssystem manifestieren. Aus diesem Grund sind sie die entscheidenden Anknüpfungspunkte für eine anwendungsgewahre Spezialisierung des Betriebssystems.

Die Gesamtheit dieser Informationen kann aber nicht mittels statischer Analysen ermittelt werden. Dies liegt zum einen daran, dass Teile dieser Informationen echte Abhängigkeit zur Laufzeit haben und von Umweltinteraktionen und Nutzereingaben abhängen. Hierbei ist es explizit gewollt, dass diese nicht bereits zur Übersetzungszeit feststehen. Zum anderen ist bedingt durch das dynamische Programmiermodell der zu optimierenden Systeme die Bestimmung der Parameter und Kontexte der Systemaufrufe nicht uneingeschränkt möglich. In der Beispielanwendung zeigt sich dies exemplarisch in den Zeilen 13 und 25, welche als Ausschnitt in der Quellcode 3.2 dargestellt sind. In Zeile 13 wird mit dem Ziel des gegenseitigen Ausschlusses ein Semaphor erniedrigt. Dieser ist als in einer objektlokalen Zeigervariable gespeichert. Eben diese Indirektion über den Objektzeiger kann Ursache dafür sein, dass der hier verwendete Semaphor nicht dem im Konstruktor in Zeile 13 zugeordnet werden kann. Gleiches gilt für die konkrete Implementierung der Funktion, welche anhand der Tabelle virtueller Methoden des Objekts ermittelt werden muss. In Zeile 25 werden für die Erzeugung des Fadens Informationen über einen angebundenen Sensor an die designierte Einstiegsfunktion übergeben, die zur Laufzeit aus dem angebundenen Sensor abgefragt werden. Der Wert, den eben dieser Parameter zur Laufzeit haben wird, ist demnach nicht statisch analysierbar.

Quellcode 3.2 Unsichere Parameter und unsicherer Kontext von Systemaufrufen. In Zeile 13 ist der Kontext nicht eindeutig bestimmbar, da die umgebende Funktion mehrdeutig als Ziel indirekter Funktionsaufrufe kandidiert. Zeile 25 zeigt einen Systemaufruf mit nicht sicher bestimmbaren Parametern.

```
3   virtual void get(DT *d) {
4     xSemaphoreTake(data_mutex, BLOCK_FOREVER);
5     *d = internal_data;
6     xSemaphoreGive(data_mutex);
7   }

13  GuardedData() {data_mutex = xSemaphoreCreateMutex();}
```

Wie sehr die Einflüsse des dynamischen Programmiermodells und die Verwendung verschiedener Ausdrucksmittel der Programmiersprachen die Analysen beeinflussen, sei an dieser Stelle jedoch nicht von Beachtung und wird gesondert in Kapitel 6 behandelt, sodass hier von einer ausreichenden Analysierbarkeit der gegebenen Anwendung ausgegangen werden kann.

Instanz- und Interaktionsgraph

Ziel der nachfolgenden Analysen ist die Erstellung des *Instanz- und Interaktionsgraphen (IIG)*. Die Menge dessen Knoten heißt *Instanzliste*. Sinn und Zweck dieser Liste ist die Verwaltung der Informationen über die einzelnen Systemobjekte, die bei statisch konfigurierten Systemen wie z. B. OSEK-konformen Systemen bereits in der jeweiligen Konfigurationssprache gegeben sind. Für Fäden wäre dies beispielsweise deren Einstiegspunkt in den Kontrollfluss und Planungsinformationen zur Ablaufplanung. Ebenso gehören hierzu die zugeordneten Ressourcen wie der Laufzeitstapel.

Erweitert wird diese Liste anschließend zum IIG, indem die möglichen Interaktionen zwischen den Systemobjekten als Kanten eingezogen werden. Auch diese Information ist bei statisch konfigurierten Systemen wie OSEK und AUTOSAR bereits im Rahmen der Systemkonfiguration gegeben.

Informationsquelle

Im Gegensatz zu statisch konfigurierten Systemen wie z. B. OSEK/AUTOSAR, welche die Systemobjekte und deren mögliche Interaktionen in einer gesonderten Konfigurationssprache beschreiben [OSE05; OSE04], ist dies so bei den anvisierten dynamisch konfigurierten Systemen nicht möglich. Die einzige Möglichkeit, die diese Systeme zur Informationsgewinnung aufgrund des dynamischen Programmiermodells bereithalten, ist, die Informationen aus dem Programmcode zu gewinnen. Aus Sicht der Spezialisierung ist eine explizite Konfiguration eindeutig zu bevorzugen, die gefühlte höhere Flexibilität der dynamisch konfigurierten Systeme führt jedoch dazu, dass diese gerne insbesondere im weichen Echtzeitkontext oder im Bereich des Internets der Dinge weite Verbreitung finden.

Ist der Programmcode die einzige vorhandene Informationsquelle, bestehen zwei grundsätzliche Möglichkeiten, an die darin enthaltene Informationen zu gelangen. Entweder wird ein Referenzlauf beobachtet und alle darin verwendeten Systemaufrufe mitgeschnitten oder der Programmcode wird statisch vor der Laufzeit analysiert.

Dietrich und Lohmann verwendet ausschließlich statische Analysen des Programmflusses, um den vollständigen Interaktionsgraphen zu ermitteln und alle Systemzustände aufzuzählen. Damit ist der vollständige Zustandsraum der Systemzustände statisch bekannt und wird verwendet, um das Betriebssystem entsprechend zu generieren [DL17].

Bei SYNTHESIS wird während der Laufzeit des Systems die gerade benötigte Systemaufrufsemantik analysiert und auch zur Laufzeit alle dafür notwendigen Analysen durchgeführt. Damit sind zwar alle benötigten Informationen vorhanden, die Analyse und anschließende Spezialisierung ist aber nicht im Voraus, sondern nur während der Ausführung möglich [PMI88].

Ziegler u. a. verwendet den Mitschnitt eines Referenzlaufes, um die auftretenden Funktionsaufrufe zu detektieren. Zusätzlich wird auch statisch der mögliche Aufrufgraph direkter Funktionsaufrufe ermittelt und mit dem Laufzeitmitschnitt kombiniert. Dadurch kann erfasst werden, welche Komponenten der untersuchten Programmteile verwendet werden, auch wenn dies über Funktionszeiger geschieht [Zie+19]. Für den Anwendungsfall der statischen Beschneidung von Programmbibliotheken und Binärdateien auf Ebene der Abstraktionen ist der dort integrierte Ansatz ausreichend, da ganze Funktionseinheiten gemeinsam selektiert werden. Für die hier vorgestellten Spezialisierungen ist dies jedoch nicht ausreichend, da aus dem Referenzlauf nicht zwingend alle während der späteren Ausführung möglichen Pfade erreicht werden.

Ich habe im Rahmen dieser Arbeit den Ansatz gewählt, ausschließlich statische Analysen zu verwenden. Diese liefern zur Übersetzungszeit Ergebnisse und sind von der späteren Ausführung unabhängig. Insbesondere für tief eingebettete Systeme resultiert der somit nicht aufzuwendende Mehraufwand für die Erzeugung eines Laufzeitmitschnitts von Vorteil, da dieser Mehraufwand einen nicht zu vernachlässigenden Einfluss auf die Laufzeiteigenschaften des Systems hat.

Die im Folgenden vorgestellten Algorithmen arbeiten auf der Basis des statischen Kontrollflusses der Anwendung. Dieser wird aus dem gegebenen Programmcode auf Ebene des LLVM-Zwischencodes ausgewertet und mögliche Ziele für Zeiger nachverfolgt.

Die statischen Analysen werden zur Bindezeit in den Übersetzungsprozess eingegliedert. Zu diesem Zeitpunkt liegt der gesamte Anwendungscode vor und ist bereit, an die Betriebssystemimplementierung gebunden zu werden. Bevor diese Bindung jedoch geschieht, werden die nun folgenden Analysen und die in den folgenden Abschnitten vorgestellten Spezialisierungen in den Prozess eingereiht. Abbildung 3.1 zeigt die Schritte des Übersetzungsprozesses ausgehend vom Programmcode der Anwendung hin zu einem fertigen Speicherabbild, welches in den Speicher des eingebetteten Systems geladen werden kann.

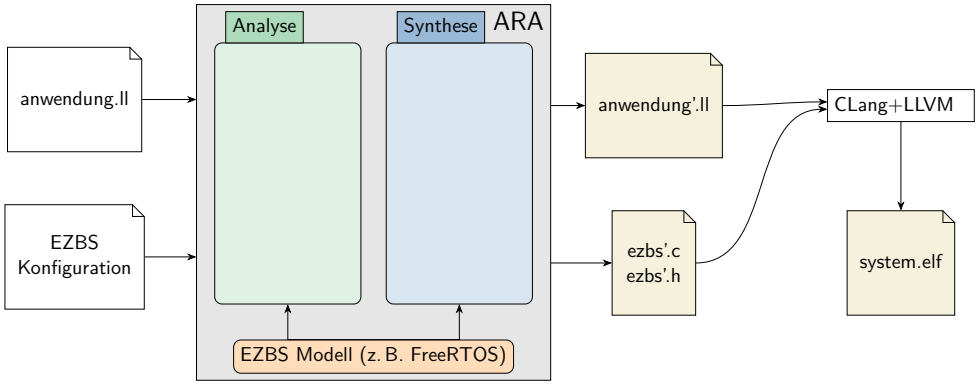

Abbildung 3.1 – Überblick über den Aufbau von ARA. Als Eingaben dienen die Anwendung als LLVM-Zwischencode und die Konfiguration des verwendeten Betriebssystems. Zunächst wird die Anwendung analysiert und der Instanz- und Interaktionsgraph gebildet. Anschließend werden in der Synthese die entsprechenden Spezialisierungen durch Modifikation des Anwendungscodes und Generierung des spezialisierten Betriebssystems durchgeführt. Das Ergebnis wird gebunden und mittels Bindezeitoptimierung zum finalen Systemabbild zusammengesetzt.

3.2 Die statische Instanzanalyse (SIA)

Bei der Frage nach den Systemobjekten und deren Interaktionen kommt zwangsläufig zuerst die Frage auf, welche Systemobjekte existieren. Diese Frage ist die Motivation für die erste notwendige Analyse: Die SIA liefert aus einem gegebenen Programmcode die Liste aller Systemobjekte [ESD19; ▷Fie+21]. Die SIA wurde primär von Gerion Entrup entwickelt und ich hatte die Ehre, die Entwicklung durch direkte Verwendung der Ergebnisse mit zu begleiten. Da die Analyse hier nur Werkzeug zum Zweck der Informationsgewinnung ist, werde ich hier nur die relevanten Aspekte im Detail beleuchten. Weitere Details sind in den entsprechenden Veröffentlichungen [ESD19; ▷Fie+21; ENL22] sowie in Gerion Entrups Dissertation nachzulesen, sobald Letztere erscheint.

Die Aufgabe der SIA ist es, aus einer als LLVM-Zwischencode gegeben Anwendung alle Systemobjekte und deren Eigenschaften zu extrahieren. Dadurch entfällt die Notwendigkeit für den Entwickler, diese in einer expliziten Konfigurationssprache zu beschreiben und auch echt dynamische Instanziierungen solcher Objekte werden ermöglicht. Zu diesem Zweck wird der interprozedurale Kontrollflussgraph beginnenden vom anwendungsspezifischen Einstiegspunkt aus traversiert. Alle vorhandenen Systemaufrufe, die ein Systemobjekt erzeugen, werden dabei analysiert. Zu diesen wird jeweils der Aufrufpfad sowie alle notwendigen Argumente zum Beschreiben des zu erzeugenden Systemobjekts ermittelt.

3.2.1 Von Kontrollflussgraphen und Werteflussgraphen

Eine Anwendung besteht im Sinne des Programmcodes aus einer Menge an Funktionen, wobei eine dieser als Einstiegspunkt ausgezeichnet ist. Jede dieser Funktionen beschreibt mittels Verzweigungen und Sprunganweisungen den Kontrollfluss innerhalb ihrer selbst. Diese lokalen Kontrollflüsse haben jeweils einen definierten Einstiegspunkt am Anfang der Funktion, im Regelfall mindestens einen Ausstiegspunkt aus der Funktion und es können weitere Funktionen aufgerufen werden. Die Analyse verwendet die aus dem Übersetzerbau [ASU86] bekannten Konzepte *Basisblock (BB)* und *lokaler Kontrollflussgraph (LCFG)* zur Darstellung dieser Kontrollflüsse.

Basisblock

Ein *Basisblock* ist ein ausschließlich sequenziell ausführbarer Bereich des Programmcodes mit genau einem Einstiegspunkt und genau einem Ausstiegspunkt [ASU86, S. 529]. Die Anweisungen innerhalb dieses Basisblocks können also nur als Einheit ausgeführt werden. Am Ende eines Basisblocks steht konzeptionell eine Sprunganweisung, die angibt, in welchem nächsten Basisblock der Kontrollfluss weitergeht. Diese Sprunganweisung kann eine Verzweigung dergestalt darstellen, dass der Kontrollfluss basierend auf berechneten Werten in verschiedenen Folgeblöcken fortgesetzt wird.

Kontrollflussgraph

Durch Darstellung der Basisblöcke als Knoten eines Graphen und Verbindung dieser Knoten anhand der jeweiligen Sprungziele am Ende der Basisblöcke mit Kanten entsteht ein *Kontrollflussgraph* [All70]. Dieser Kontrollflussgraph beschreibt alle möglichen Verläufe des Kontrollflusses innerhalb einer Funktion. Im Folgenden wird dieser Kontrollflussgraph daher *lokaler Kontrollflussgraph (LCFG)* genannt.

Abbildung 3.2 zeigt die resultierenden LCFGs für die gegebenen Beispielfunktionen mit schwarzen Kanten. Die roten Kanten stellen die Erweiterung des lokalen Kontrollflussgraphen zum *interprozeduralen Kontrollflussgraphen ICFG* dar, indem die Aufrufstellen jeweils mit den aufgerufenen Funktionen mit Kanten verbunden werden. Für direkte Funktionsaufrufe ist diese Zuordnung eindeutig. Handelt es sich jedoch um einen indirekten Funktionsaufruf, so muss die aufzurufende Funktion erst ermittelt werden. Dies bedeutet, dass der Wert des Funktionszeigers aus dem Kontext ermittelt werden muss. Eine eindeutige Zuordnung ist daher nicht immer möglich, sodass hier mit einer Überabschätzung gearbeitet wird. Die gewählte Überabschätzung ist hierbei die Menge aller Funktionen, deren Signatur der an der Aufrufstelle geforderten entspricht und von der die Adresse programmatisch ermittelt wird, also anders verwendet wird als für direkte Funktionsrufe. Diese Überabschätzung führt dazu, dass eine Menge von Funktionen als mögliches Aufrufziel behandelt werden muss. Hieraus erwächst ein größerer Raum für den Kontrollfluss, es geht aber kein Pfad verloren, sodass die Ergebnisse der Analyse valide bleiben. Weitere Details zu dem Problem der Berechnung von Funktionszeigern werden in Abschnitt 6.5.2 behandelt.

ABB

Für die nun folgenden Analysen ist der exakte Kontrollfluss nicht in allen Bereichen im Detail notwendig. Insbesondere sind nur die Systemaufrufe von Interesse, der genaue Verlauf des Kontrollflusses innerhalb der Anwendungslogik kann in weiten Teilen als irrelevant aus Sicht des Betriebssystems gesehen werden, solange er zu denselben Systemaufrufen führt. Hierzu wird das durch Scheler[Sch11; SSP10] eingeführte Konzept des *Atomaren Basisblocks ABB* verwendet. Sie sind analog zu den BBs ebenso Regionen mit genau einem Ein- und Ausstiegspunkt, die linear ohne Verzweigungen abgearbeitet werden und nur am Ende einen Sprung in andere Blöcke haben. Die Eigenschaft atomar (altgriechisch átomos, „unteilbar“) bezieht sich auf die verzweigungsfreie lineare Abarbeitung der darin enthaltenen Anwendungslogik aus Sicht des Betriebssystems. Der Kontrollflussgraph innerhalb eines *Atomarer Basisblocks (ABBs)* kann dabei Verzweigungen enthalten, solange alle Pfade den Ausstiegspunkt und damit denselben nachfolgenden Systemaufruf erreichen, da hier nur die Sicht des Betriebssystems relevant ist. Vielmehr wird der nicht systemaufrufrelevante Kontrollfluss in diesen zusammengefasst, während Systemaufrufstellen jeweils ihren eigenen ABB erhalten. Lineare Abarbeitung bedeutet aber nicht, dass der Handlungsstrang nicht durch einen höherprioren Handlungsstrang verdrängt und in seiner Ausführung pausiert werden kann. Logisch werden alle in einem ABB enthaltenen Instruktionen sequenziell ausgeführt. Die Knoten des *interprozeduraler Kontrollflussgraph (ICFG)*s können so in ABBs überführt werden.

Die ABBs sind anhand ihres Inhaltes in drei Kategorien teilbar: Reine Berechnungs-ABBs enthalten nur Anwendungscode, der Berechnungen durchführt, ohne jemals einen Systemaufruf auszulösen. Systemaufruf-ABBs enthalten neben dem finalen Sprung in den nächsten ABB genau einen Systemaufruf. Zuletzt existieren indirekte Systemaufruf-ABBs, die zu einem Systemaufruf führen, diesen jedoch nicht selber enthalten, sondern auf dem Kontrollflusspfad zu diesen liegen. Abbildung 3.2 zeigt die Einordnung der ABBs der Beispielanwendung.

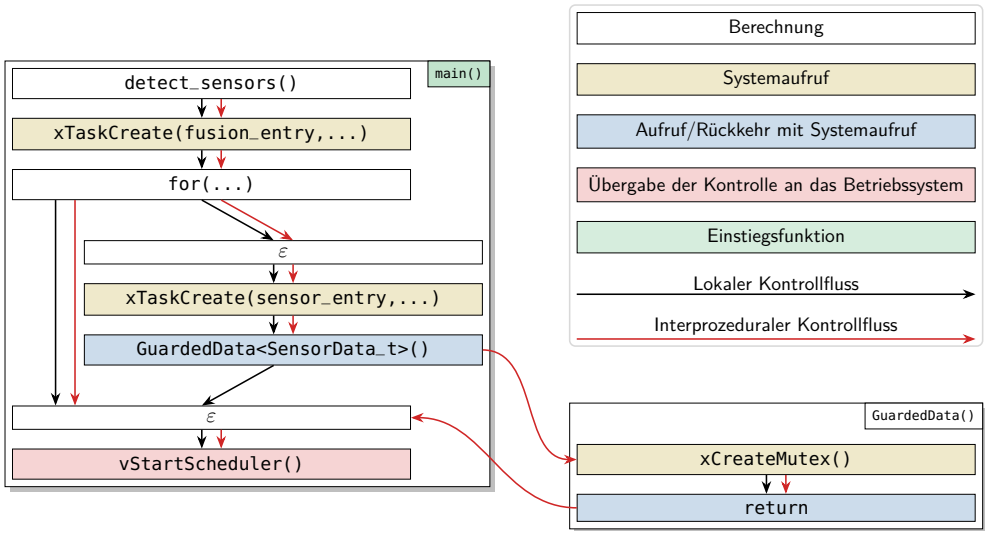

Abbildung 3.2 – Kontrollflussgraph der Beispielanwendung. Gezeigt ist ein Ausschnitt aus der Einstiegsfunktion `main()` und dem Konstruktor `GuardedData()`. Während der lokale Kontrollfluss innerhalb der Funktion verbleibt, verbindet der interprozedurale Kontrollfluss über Funktionsgrenzen hinweg. Die Knoten sind die nach Inhalt klassifizierten ABBs.

Eine Menge Berechnungs-ABBs, die sich gegenseitig dominieren und postdominieren, können zu einem ABB zusammengefasst werden. Zusätzlich genügt es für die nun folgenden Analysen, sich zunächst auf diejenigen ABBs zu beschränken, die im Zusammenhang mit Systemaufrufen stehen. Dies führt zu einer Reduktion der Komplexität des Kontrollflussgraphen und der zu untersuchenden Knoten und somit auch zu einer verbesserten Effizienz der Analysen [Die19].

Werteflussgraphen

Entlang des Kontrollflusses werden während der Laufzeit der Anwendung Werte von Variablen und Parametern übergeben, die für spätere Berechnungen und Funktionsaufrufe verwendet werden. Zur Auswertung der Systemaufrufe werden dementsprechend auch für die hier verwendeten statischen Analysen diese Werte benötigt. Die statische Werteflussanalyse ist dabei ein eigenständiges Forschungsgebiet, welches nicht Thema dieser Ausarbeitung ist. Für die Implementierung wird daher auf das Werkzeug SVF [SX16] zurückgegriffen. Dieses führt eine statische Werteflussanalyse auf Basis desselben Programmcodes durch wie auch die Kontrollflussanalysen, sodass die Ergebnisse gut miteinander kombinierbar sind. Genauso wie die Zeigeranalyse, welche ein Spezialfall der statischen Wertanalyse ist, hat auch die allgemeine statische Wertanalyse Grenzen und Einschränkungen. Diese werden gemeinsam im Abschnitt 3.5 näher betrachtet. Zunächst gehe ich jedoch davon aus, dass Wissen über die notwendigen Werte gewonnen werden kann.

3.2.2 Das Betriebssystemmodell

Systemaufrufe und reguläre Funktionsaufrufe sind aus der Sicht der Aufrufstelle zunächst nicht zu unterscheiden. Beide werden üblicherweise als Funktionsaufrufe in der Hochsprache dargestellt und erst später in der Maschinensprache ergibt sich eventuell ein Unterschied, wie das jeweilige Betriebssystem und die Prozessorarchitektur in Kombination die Betriebssystemrufschnittstelle implementieren. Dementsprechend sind Systemaufrufe bis dahin lediglich eine Menge an aufrufbaren Funktionen, die als Systemaufruf ausgezeichnet sind. Ein Betriebssystemmodell gibt zu diesem Zweck die Menge der als Systemaufruf zu interpretierenden Funktionen vor. Die Analysen fragen

beim Abgehen der Funktionsaufrufstellen das Betriebssystemmodell, ob der gerade betrachtete Funktionsaufruf ein Systemaufruf ist. Ist dies der Fall, so gibt das Betriebssystemmodell ebenfalls Auskunft darüber, wie die Parameter und Rückgabewerte zu interpretieren sind. Dieser Umstand ermöglicht es, die Algorithmen der statischen Analysen getrennt von der konkreten Betriebssystemschnittstelle zu konstruieren. Durch Wahl eines entsprechenden Betriebssystemmodells sind die Algorithmen in der Lage, Anwendungen für unterschiedliche Betriebssystemschnittstellen zu analysieren.

In Abbildung 3.3 ist ein vereinfachter Ausschnitt des in ARA verwendeten FreeRTOS-Betriebssystemmodells für die Erzeugung eines Puffers und das Senden von Daten in diesen Puffer dargestellt. Die Annotationen beschreiben die Bedeutung der Parameter des Systemaufrufs. Der Funktionskörper übernimmt die abstrakte Interpretation des Systemaufrufs und die notwendigen Eintragungen in Instanz- und Wertegraphen.

3.2.3 Ablauf der SIA

Ausgehend von der Existenz der interprozeduralen Wert- und Kontrollflussgraphen ist es die Aufgabe der SIA, diese zu traversieren und die jeweiligen systemobjektinstanziierenden Systemaufrufe zu analysieren. Hierzu wird der Kontrollflussgraph ausgehend vom Einstiegspunkt der gegebenen Anwendung nach eben solchen durchsucht. Quellcode 3.3 zeigt den Algorithmus der SIA in Pseudocode. Die Funktion `find_soc()` durchläuft den gegebenen Kontrollflussgraphen. Auf der Suche nach Systemaufrufen wird hierbei der Aufrufpfad mitgespeichert und ebenso die Information, ob sich die Ausführung auf einem nur bedingt erreichbaren (Verzweigung) oder eventuell mehrfach betretenen (Schleife) Pfad befindet. Diese Information beeinflusst die Eindeutigkeit der später zu erzeugenden Systemobjekte.

Wird ein Systemaufruf gefunden, wird dieser durch die Funktion `handle_soc()` interpretiert. Dabei werden im Rahmen der SIA nur diejenigen Systemaufrufe analysiert, die neue Systemobjekte

```
@syscall(categories={SyscallCategory.create},
         signature=(Arg("task_function", hint=SigType.symbol, ty=Function),
                    Arg("task_name"),
                    Arg("task_stack_size"),
                    Arg("task_parameters", hint=SigType.symbol),
                    Arg("task_priority"),
                    Arg("task_handle_p", hint=SigType.instance)))
def xTaskCreate(graph, os_state, cpu, args):
    v = graph.iig.add(Task(cpu, os_state, args))
    FreeRTOS.handle_create(os_state, v)
    graph.vfg.assign_system_object(args.task_handle, v)


@syscall(categories={SyscallCategory.comm},
         signature=(Arg('handler', ty=Queue, hint=SigType.instance,
                        candidates=list_all_queues()),
                    Arg('item'),
                    Arg('ticks')))
def xQueueReceive(graph, state, cpu_id, args, va):
    queue_node = graph.vfg.find_instance_node(state.instances, args.handler)
    graph.iig.connect_interaction(state, cpu, queue_node, "xQueueReceive",
                                  ticks=[args.ticks])
```

Abbildung 3.3 – Ausschnitt aus dem von ARA verwendeten FreeRTOS Betriebssystemmodell

erzeugen oder den Zustand künftig neu zu erzeugender Systemaufrufe beeinflussen, alle übrigen werden für diese Analyse ignoriert.

Die Parameter werden je nach Signatur des Systemaufrufs unterschiedlich behandelt. Ist ein Parameter als Wert vorgesehen, so ist das Ziel an diesen Wert zu gelangen und ihn als Konstante zu ermitteln. Handelt es sich hingegen um einen Referenzparameter, so die entsprechende Referenz ermittelt. Ebenso werden die ermittelten Umgebungsparameter wie Aufrufpfad, Verzweigungs- und Schleifeninformationen mit vermerkt. Diese wertet die Parameter des Aufrufs anhand des Werteflussgraphen aus und speichert diese für die späteren Optimierungen in Stellvertreterobjekten in der Instanzliste.

Gleichzeitig werden Referenzen auf diese Stellvertreter in den Werte- und Kontrollflussgraphen vermerkt, um diese später wieder finden zu können. Für den Puffer `storage_queue` aus Zeile 36 bedeutet dies, dass die Zuweisungsoperation der lokalen Variable `storage_queue` im Datenflussgraph das Stellvertreterobjekt referenziert. Ebenso wird an den Referenzparameter `fusion_task` der Zeile 25 eine Referenz auf das Stellvertreterobjekt des dort erzeugten Fadens gesetzt.

Handelt es sich um ein Systemobjekt, welches einen neuen Handlungsstrang repräsentiert, so wird eine weitere SIA für dessen Einstiegspunkt ausgeführt. Für den Faden `fusion_task` wird demnach eine SIA ausgehend von der Funktion `fusion_entry()` gestartet. Auf diese Weise werden alle Handlungsstränge der Anwendung detektiert und die jeweiligen Zustandsinformationen korrekt zu den jeweiligen Systemobjekten propagiert.

Für die in Quellcode 3.1 gegebene Beispielanwendung ist die SIA somit in der Lage, eine Menge von zwei Fäden (`fusion_task`, `storage_task`) und einem Kommunikationsobjekt (`storage_queue`) sicher zu ermitteln. Hinzu kommt jeweils eine unbekannte Anzahl gleichartiger Fäden (`sensor_task`) und Kommunikationsobjekte (`data_mutex`), deren Parameter zwar teilweise bekannt sind, aber nicht sicher gesagt werden kann, wie viele solcher gleichartiger Objekte es im System zur Laufzeit gegeben wird. Tabelle 3.1 zeigt die resultierende Instanzliste und die darin gespeicherten Informationen. Dieser ist zu entnehmen, dass zwar alle Systemobjekte von der SIA ermittelt werden konnten, jedoch nicht alle Informationen über diese bestimmbar sind. Zu den `sensor_task`-Fäden beispielsweise (Zeile 27) konnte der Wert des Parameters der Einstiegsfunktion (`si`) nicht ermittelt werden, da dieser echt dynamisches Laufzeitwissen über die angeschlossenen Sensoren benötigt. Die so ermittelten Informationen über die Systemobjekte werden in Kapitel 4 dazu verwendet, die bisher dynamisch durchgeführten Instanziierungen während des Systemstarts in statische Instanziierungen und Initialisierungen analog einem statisch konfigurierten System zu transformieren.

Quellcode 3.3 Schematische Darstellung des SIA-Algorithmus in Pseudocode. Der Algorithmus iteriert über alle vom Betriebssystemmodell als relevant klassifizierten Systemaufrufe und aktualisiert den Instanzgraph anhand der Ergebnisse, die die Interpretation des Systemaufrufs durch das Betriebssystemmodell liefert. In Anlehnung an [ENL22]

```
1 def SIA(einstiegs_funktion) -> Instanzgraph:
2     instanzgraph = Instanzgraph()
3     for aufruf in einstiegs_funktion.cfg():
4         if os_model.ist_relevanter_systemaufruf(aufruf):
5             for kontext in alle_aufrufkontexte(einstiegs_funktion, aufruf):
6                 instanzgraph.aktualisiere(
7                     os_model.interpretiere(aufruf, kontext))
```

Systemobjekte	fusion_task	storage_task	storage_queue	sensor_task	data_mutex
Erzeugt vor dem Start des Betriebssystems	✓	×	×	✓	✓
Genau einmal erzeugt	✓	✓	✓	×	×
Betriebssystem alloziert den Speicher	✓	✓	✓	✓	✓
Alle Parameter sind bekannt	×	✓	✓	×	✓
Wird nicht gelöscht	✓	✓	✓	✓	✓

Erzeugungsparameter:
fusion_task = {Typ=Faden, Laufzeitstapel=512, Funktion=fusion_entry, Initialparameter=unbekannt, ...}
storage_task = {Typ=Faden, Laufzeitstapel=512, Funktion=storage_entry, Initialparameter=NULL, ...}
sensor_task = {Typ=Faden, Laufzeitstapel=512, Funktion=sensor_entry, Initialparameter=unbekannt, ...}
storage_queue = {Typ=Puffer, Elemente=4, Elementgröße=sizeof(message_t), ...}
data_mutex = {Typ=Semaphor, Initialwert=1, ...}

Tabelle 3.1 – Instanzliste der Beispielanwendung aus Quellcode 3.1. Dargestellt sind die gesammelten Informationen bezüglich der Instanziierung der jeweiligen Systemobjekte. Symbolbedeutung: ✓ wahr/vollständig möglich; (✓) partiell möglich; × falsch/nicht möglich

3.3 Die Interaktionsanalyse (INA)

Die Systemobjekte der gegebenen Anwendungen sind nun bekannt. Der nächste logisch folgende Schritt ist die Frage nach der Interaktion dieser untereinander, denn eben diese Interaktion repräsentiert das Kommunikationsverhalten der gegebenen Anwendung mittels der durch das Betriebssystem bereitgestellten Primitiven. Zu diesem Zweck wird basierend auf den Grundlagen und dem Algorithmus der SIA die INA durchgeführt.

3.3.1 Ablauf der INA

Der grundsätzliche Algorithmus der statischen Interaktionsanalyse (INA) ist gleich dem der SIA in dem, dass anhand des Kontrollflusses die Systemaufrufe abgegangen werden und deren Bedeutungen interpretiert werden [ENL22]. Anstelle der instanziierenden Systemaufrufe werden nun jedoch diejenigen Systemaufrufe betrachtet, die eine Interaktion durchführen. Die Interaktionspartner werden hierbei aus den Parametern des jeweiligen Systemaufrufs mittels derselben statischen Werteflussanalysen ermittelt, sodass die zuvor im Datenflussgraph gespeicherten Stellvertreterobjekte als Interaktionspartner dienen. In der vorliegenden Beispielanwendung lässt sich somit für Zeile 40 ermitteln, dass durch den Faden `fusion_task` Daten in den Puffer `data_queue` geschrieben werden. Für Zeile 48 wird der Faden `storage_task` eine Leseoperation auf selbigen Puffer ausführen. In Abbildung 3.4 ist dabei die Verbindung zwischen der Verwendung des Puffers in Zeile 40 und der Erzeugung dieses in Zeile 36 über die Funktionsgrenzen hinweg dargestellt.

Die INA ermittelt auch, ob ein Systemobjekt, welches von der SIA erkannt wurde, im späteren Verlauf der Anwendung eventuell wieder aus dem System gelöscht wird. Hierbei handelt es sich gewissermaßen auch um eine Interaktion zwischen zwei Systemobjekten, wobei hier vor allem das Zielobjekt relevant für die Interaktion ist. Initial wird kein Objekt wieder aus dem System gelöscht und erst das Vorhandensein eines entsprechenden Systemaufrufs führt dazu, dass eventuell Objekte der entsprechenden Klasse als möglicherweise gelöscht attribuiert werden. In FreeRTOS

sind die entsprechenden Aufrufe für Fäden beispielsweise `vTaskDelete()` und für Nachrichtenpuffer `vMessagebufferDelete()`.

3.4 Ergebnisse aus SIA und INA

Die Kombination der Ergebnisse der beiden Analysen SIA und INA erzeugt einen Graphen. Dieser Graph enthält als Knoten alle statisch ermittelten (Stellvertreter-) Systemobjekte. Als Resultat der INA sind diese Knoten durch Kanten verbunden, die die jeweiligen Interaktionen repräsentieren. Somit wird der als Knotenliste gestartete Instanzgraph zum IIG vervollständigt Abbildung 3.5 zeigt den gesamten IIG für die Beispielanwendung aus Quellcode 3.1. Dieser IIG dient als Grundlage für die in den folgenden Abschnitten vorgestellten statischen Spezialisierungen.

Neben den bereits genannten Interaktionen der Fäden `fusion_task` und `storage_task` ist keine weitere Verwendung des Puffers `storage_queue` vorhanden. Anders ist die Lage bei den Semaphoren `data_mutex`. Hier kann nicht eindeutig statisch ermittelt werden, welcher Faden auf welchen Semaphor zugreift. Dies wird in den nächsten Abschnitten der Grund dafür sein, dass diese nicht weiter spezialisierbar sind.

Zeichnet man alle Interaktionen in den IIG ein, so bilden sich jeweils zusammenhängende Bereiche von Knoten, die die jeweiligen Interaktionen beschreiben. Es ist zu erkennen, welche Bereiche der Anwendung miteinander stark interagieren und welche weitgehend unabhängig voneinander arbeiten.

3.5 Einschränkungen der Analysen

Die Beispielanwendung ist genau so konstruiert, dass die Analysen die Instanzen `fusion_task`, `storage_queue` und `storage_task` und deren Interaktionen vollständig erkennt. Gleichzeitig wird

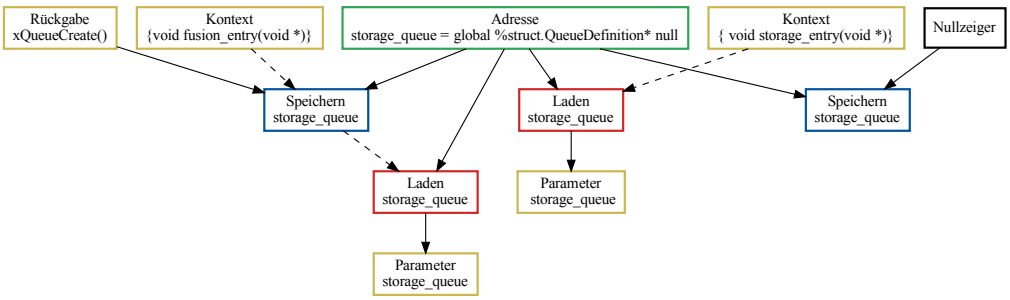

Abbildung 3.4 – Ausschnitt aus dem Werteflussgraph der Beispielanwendung. Gezeigt ist die Zuweisung (blau) der Referenz auf das Queue-Objekt (grün) als Resultat der Erzeugung (gelb) in Zeile 36 und der lesende Zugriff (rot) beim Schreiben über den Aufrufkontext (gelb) der Einstiegsfunktion `storage_entry()` in Zeile 40 der Beispielanwendung. Ebenfalls zu sehen ist die Zuweisung (blau) der initialen Belegung der globalen Variable mit einem Nullwert. Beim Instanziieren wird das Systemobjekt an den Zuweisungsknoten gebunden. Bei einer Verwendung kann ausgehend von den Parameterknoten durch Verfolgen der Kanten so die Menge der möglichen Systemobjekte gefunden werden.

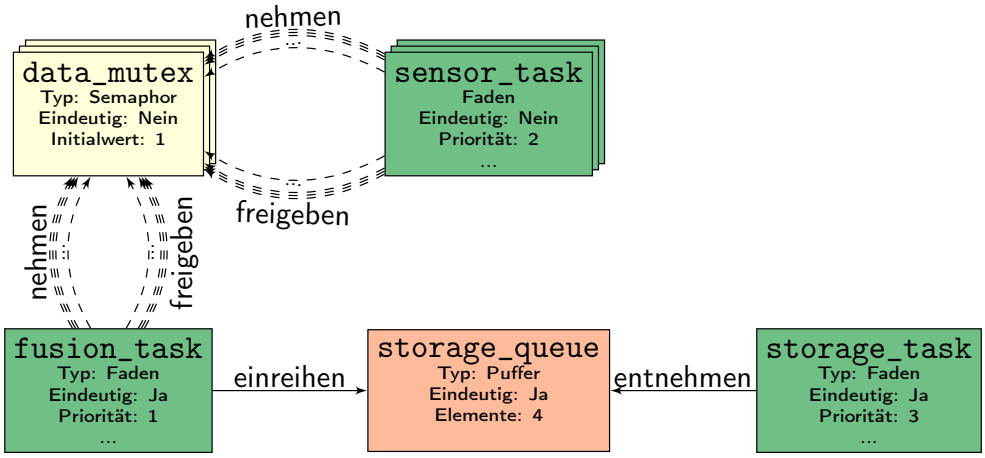

Abbildung 3.5 – Instanz- und Interaktionsgraph der Beispielanwendung aus Quellcode 3.1. Fäden (grün), Puffer (orange) und Semaphore (gelb) bilden das System.

aber bereits bei dieser Demonstrationsanwendung sichtbar, dass die Analyse nicht alle Komponenten der Anwendung vollständig erfassen kann. Insbesondere die statische Wertanalyse kommt an ihre Grenzen, wenn es Werte zu ermitteln gilt, die echt dynamisch zur Laufzeit berechnet werden oder die über zu viele mehrdeutige Indirektionen weitergereicht werden. Mehrdeutig bedeutet dabei, dass die Zeigeranalyse mehrere potenzielle Ziele ermittelt, auf die ein gegebener Zeiger zeigen kann. Die Zeigeranalyse findet sowohl bei über Zeiger referenzierten Datenstrukturen als auch beim Auflösen der virtuellen Methoden polymorpher Objekte eine Vielzahl möglicher Ziele. Letzteres wird in der vorliegenden Implementierung auf Basis von LLVM-Zwischencode zusätzlich erschwert, da das Konzept der virtuellen Methoden nicht nativ abgebildet ist, sondern über Felder von Funktionszeigern ohne besondere Auszeichnung dieser und ohne eine explizite Auszeichnung polymorpher Vererbungshierarchien abgebildet ist. Stattdessen ist Polymorphie als aufrufkontextabhängige Interpretation desselben Speicherobjekts und Zusammenfassung strukturäquivalenter Typdefinitionen umgesetzt. Um dennoch valide Ergebnisse der Analysen zu erhalten, bleibt an der Stelle nur die Lösung der Überapproximation. Die Zeigeranalyse verwendet an allen Stellen, an denen kein exaktes Zielobjekt ermittelt werden kann, die Menge aller Objekte deren Typ zu dem Zeigertypen passt und von denen eine Referenz genommen wird. Dies bedeutet, dass zwar eventuell zu viele Systemobjekte Ziel einer gefundenen Interaktion sind oder zu viele mögliche Interaktionen und dazugehörige Aufrufpfade gefunden werden, jedoch keine Möglichkeit fälschlicherweise ausgelassen wird.

Für die Beispielanwendung bedeutet dies folgende Einschränkungen: Auf der einen Seite ist die genaue Anzahl der für das Erfassen von Sensordaten zuständigen Fäden und der dazugehörigen Daten nicht zur Übersetzungszeit ermittelbar, da diese Information erst zur Laufzeit ermittelt wird. Andererseits wäre aber auch bei bekannter Anzahl der Sensoren die Zuordnung der Semaphoroperationen zu dem jeweils richtigen Semaphor schwierig, da diese über Zeigerindirektionen und instanzlokale Variablen abgebildet sind. Deshalb wird für jede detektierte Semaphor-Interaktion eines Sensor-Fadens, die aus den Zeilen 4, 6, 9 oder 11 resultiert, jeweils eine entsprechende Kante zu jedem in Frage kommenden Semaphor in den IIG eingefügt. Gleiches gilt für die Zugriffe von dem Fusions-Faden auf die Semaphore. Auch für diese werden Kanten von dem Faden zu jedem Semaphor in den IIG eingezogen.

Die Analysen werden dadurch zwar ungenauer, bilden aber stets mindestens das Verhalten der Anwendung oder eine Obermenge ab, sodass spätere Spezialisierungen, die das Systemverhalten auf

die Analyseergebnisse zurechtschneiden, keine Bereiche auslassen, sondern höchstens ungenutzte Flexibilität nicht optimieren können. Die richtige Kante, also die, die die tatsächlich stattfindende Interaktion abbildet, ist stets in der Menge der eingefügten Kanten. Nur ist keine präzise Aussage möglich, welche Kante dies ist, sodass die Interaktion nicht weiter eingeschränkt werden kann. Es muss im Folgenden davon ausgegangen werden, dass alle diese Kanten verwendet werden. Mit dieser Überapproximation sinkt die Möglichkeit, die Interaktion präzise zu spezialisieren, es ist aber in jedem Fall gewährleistet, dass die Anwendung den Anforderungen entsprechend funktioniert. Eine weitere Diskussion dieser Einschränkung findet in Kapitel 6 statt. Dort wird auch diskutiert, wie mit diesen Einschränkungen in der Realität umgegangen werden kann, um die Entwickler solcher Systeme zu sensibilisieren und die Analysen durch gezielte Änderungen der Entwickler zu verbessern.

3.6 Zusammenfassung

Die Ermittlung von Wissen über das zu spezialisierende System ist die Grundlage jeder Spezialisierung. In diesem Kapitel haben wir die notwendigen Werkzeuge kennengelernt, die im Rahmen einer statischen Programmcodeanalyse die von der Anwendung verwendeten Betriebssystemobjekte und deren Interaktionen zu ermitteln. Auf Basis des Anwendungscodes in LLVM-Zwischencode extrahiert die SIA dabei aus dem ICFG die verwendeten instanziierenden Systemaufrufe samt ihrer Aufrufkontexte und ermittelt unter Verwendung des Betriebssystemmodells die Auswirkungen dieser. So entsteht eine Liste von Systemobjekten mit ihren Initialparametern. Als nächster Schritt arbeitet die INA auf derselben Grundlage und analysiert alle Systemaufrufe, die zur Interaktion zwischen den Systemobjekten oder mit dem Betriebssystem dienen. Sie erweitert damit die Liste von Systemobjekten zum IIG, indem alle Interaktionen als Kanten zwischen den als Knoten repräsentierten Systemobjekten eingetragen werden.

Mit diesen Analysen ist der Grundstein für die nachfolgenden Spezialisierungen in den nächsten Kapiteln gelegt. Die Spezialisierungen verwenden jeweils das im IIG gespeicherte Wissen über die Anwendung, um die möglichen und lohnenswerten Spezialisierungsvarianten zu ermitteln und Betriebssystem und Anwendung dementsprechend anzupassen.

4

Systemstart

Wiederholte Abläufe ohne Wiederholung ausführen

Nachdem sich das vorhergehende Kapitel 3 mit der Beschaffung von Wissen über die gegebene Anwendung beschäftigt hat, soll nun dieses Wissen verwendet werden. Grundlegendes Ziel dabei ist, gemäß der Forschungsfrage die Vorteile eines statisch konfigurierten Systems auf dynamisch konfigurierte Systeme anzuwenden.

Der Ablauf des auszuführenden Programms eines eingebetteten Systems besteht grundlegend aus mehreren Phasen, die jeweils unterschiedliche Aufgaben erfüllen. Zu Beginn steht der Start des Systems. Alle für den spezifikationsgemäßen Betrieb notwendigen Initialisierungen werden ausgeführt, bevor das System in den regulären Arbeitsbetrieb und damit in die nächste Phase übergeht. Dieses Kapitel behandelt entsprechend Optimierungsmaßnahmen, die den Systemstart optimieren und die Systemsoftware und insbesondere das Betriebssystem entsprechend der Bedürfnisse der Anwendung spezialisieren.

4.1 Problemfeld und Zielsetzung

Bevor mit der Spezialisierung der Vorgänge des Systemstarts begonnen werden kann, betrachten wir zunächst das eigentliche Problemfeld und die daraus entstehende Zielsetzung. Der Systemstart ist die Grundlage für und der Einstieg in ein arbeitsfähiges System und demnach auch der erste Bereich, dessen Spezialisierung sich diese Arbeit widmet. Er bildet die fundamentale Grundlage für die Anwendung und initialisiert alle benötigten Komponenten des Systems. Er ist dementsprechend essenzielle Voraussetzung für ein arbeitsfähiges Gesamtsystem und bestimmt so den weiteren Verlauf und die Verwendbarkeit des gesamten Systems während der gesamten Laufzeit.

4.1.1 Problemfeld

Im Rahmen dieser Arbeit befinden wir uns, wie bereits beschrieben, im Kontext von eingebetteten Spezialzwecksystemen. Das bedeutet, dass die Aufgabe, die das System übernehmen soll, wohl definiert und mit der Integration in das Gesamtprodukt festgelegt ist. Der Start des Systems ist demnach ein immer wiederkehrender Vorgang, der stets zum selben Ziel, dem Erreichen des arbeitsfähigen Zustandes, führt. Dieser ist durch den in den Programmspeicher zu schreibenden Programmcode vorgegeben und verändert sich nicht von Start zu Start. Einzige Änderungsmöglichkeit ist, diesen Programmcode auszutauschen, jedoch ergibt sich für das neue Programmcodeabbild dasselbe Szenario, dass fortan zwar zu einem anderen, jedoch weiterhin wiederkehrenden arbeitsfähigen Zustand hin gestartet wird.

Konzeptionell ist der Systemstart erweiterter Teil des Ladens des Programms in den Arbeitsspeicher und der Initialisierung von dazugehörigen Variablen. In dieser Initialisierungsphase werden die Systemobjekte erzeugt, die in der späteren Arbeitsphase verwendet werden. Ich gehe davon aus, dass diese Phase konzeptionell schleifenfrei ausrollbar ist. Sie endet mit der Übergabe der Kontrolle an das Betriebssystem und den Eintritt in die Hauptschleife, in der der Planer die lauffähigen Fäden einlastet. Die Spezialisierung der Initialisierungen ist somit analog zu dem Vorgehen eines optimierenden Übersetzers. Dieser propagiert Konstanten und rollt Schleifen aus, sodass dasselbe Endergebnis effektiver erreicht wird [Wul+73].

Systemstartpunkt

Der Punkt in der Laufzeit des Systems, an dem der Startvorgang abgeschlossen ist und der reguläre Arbeitsbetrieb beginnt, nenne ich den *Systemstartpunkt (SSP)*. An diesem Punkt ist die Initialisierungsphase abgeschlossen und das Programm betritt die reguläre Arbeitsphase. Dieser Zeitpunkt muss nicht identisch mit der Übergabe der Kontrolle an das Betriebssystem und dem Start des Planers sein. Je nach Anwendungsimplementierung können auch Fäden noch Initialisierungsaufgaben vornehmen, sodass der genaue Zeitpunkt jeweils anwendungsspezifisch festzulegen ist.

Auf den ersten Blick könnte man nun sagen, dies sei eine nicht kritische Phase der Laufzeit des Systems, es genügt, wenn sie ihr Ziel erreicht und danach das System arbeitsfähig ist. Im Verhältnis zu der Zeit, die das System im arbeitsfähigen Zustand verbringt, ist der Systemstart verschwindend klein und fällt daher kaum in die Gewichtung mit ein. Bei genauer Betrachtung ist der Systemstart aber aus vielerlei Perspektive ein wichtiger Bestandteil der Laufzeit.

Ein grundlegendes Problem ist, wenn nicht nur der Start an sich einer zeitlichen Vorgabe unterliegt, sondern dieser auch das zeitliche Verhalten des gesamten Systems inklusive eventueller kritischer Pfade beeinflusst. Es ist übliches Vorgehen, den Neustart des Systems als Wiederherstellungsstrategie aus Fehlerzuständen zu verwenden[Ake+20; Abd+16; Abd+17]. Daraus resultiert aber, dass der Systemstart auf dem kritischen Pfad zu einer Reaktion mit harten Echtzeitanforderungen liegen kann. Bei einem System, welches diese Wiederherstellungsstrategie verwendet, gehört somit der Systemstart in jeden Pfad einer Echtzeitanalyse und bestimmt mit darüber, ob

zeitliche Vorgaben, die sich eigentlich nur auf den bereits gestarteten Zustand der Anwendung beziehen, eingehalten werden können. Das rechtzeitige Starten des Systems ist also eine funktionale Anforderung eines solchen Systems. Die Spezialisierung der instanziierenden Systemaufrufe mit dem Ziel der Beschleunigung dieser leistet somit einen Beitrag zum Einhalten dieser funktionalen Anforderung.

Auch auf die Benutzbarkeit eines Systems hat die Startzeit als Reaktionszeit auf das Einschalterereignis direkten Einfluss. Die wahrgenommene Benutzbarkeit eines Systems leidet unter übermäßig langen Ausführungszeiten zur Beantwortung einer Anforderung. Ist diese zu hoch, so wird ein solches System als weniger gut benutzbar wahrgenommen und steigert die mentale Belastung für die bedienende Person. Eine Antwort in unter einer Sekunde stellt keine Unterbrechung des gedanklichen Ablaufs dar und bis zu zehn Sekunden sind tolerabel für schwierige Anforderungen [Nie93; Mil68; CRM91]. Dementsprechend gibt es Anwendungsgebiete, die für dort eingesetzte Systeme harte zeitliche Vorgaben erzwingen, wie lange ein System zum Starten benötigen darf. Eine Rückfahrkamera in einem für den US-Amerikanischen Markt zugelassenen Auto muss beispielsweise innerhalb von einer Sekunde nach dem Einlegen des Rückwärtsganges voll funktionsfähig sein [TNHTSA18]. Gleichzeitig wird auch für Systeme ohne harte Anforderungen Wert auf geringe Startzeiten gelegt [Jo+09].

Ebenfalls relevant ist die Startzeit für Systeme, die mit einer schwankenden Energieversorgung betrieben werden. Hier ist es üblich, Berechnungen nur so lange auszuführen, wie sicher Energie vorhanden ist [UM21; JRR14; Bal+15]. Je geringer der Anteil der Rechenzeit ist, die für das Erreichen des arbeitsfähigen Zustandes aufgewendet werden muss, umso größer ist der Anteil, der für die eigentliche Nutzlast des Systems aufgewendet werden kann. Dementsprechend hat eine Startzeitreduktion einen entscheidenden Einfluss auf die durch das System erbringbare Arbeit.

Neben den genannten Herausforderungen aus dem Bereich der Echtzeitfähigkeit hat die dynamische Instanziierung von Systemobjekten auch einen Einfluss auf die möglicherweise auftretenden Fehler. Dynamische Instanziierung hat immer zur direkten Folge, dass Fehler beim Erstellen der Systemobjekte erst zur Laufzeit auftreten und somit auch nur dort behandelt werden können. Findet die Instanziierung hingegen während des Übersetzungsprozesses statt, so treten etwaige Fehler auch bereits während dessen auf und können behoben werden. Dies betrifft nicht nur die Ermittlung von Parametern, sondern auch die gesamte Klasse von Laufzeitfehlern, die mit dynamischer Speicherallokation einhergehen. Eine dynamische Instanziierung erfordert diese zwingend und eröffnet damit immer die Möglichkeit von Fehlern durch Überanforderung. Für statisch instanziierte Objekte ist deren Speicherbedarf bereits zur Übersetzungszeit bekannt und es kann damit sicher entschieden werden, ob dieser Speicher auf dem System verfügbar ist. Zudem entfällt die Notwendigkeit dynamischer Speicheranforderung, wenn die Instanziierung neuer Systemobjekte alleinig diese Anforderung stellt. Wie in Abschnitt 2.3.3 bereits dargestellt, reduziert dies die Ausführungszeit [▷Fie+18], verbessert die Analysierbarkeit des Systems [Die19] und reduziert den Umfang des benötigten Codes [RHL14] und erhöht damit dessen Robustheit gegenüber Angriffen [Zie+19].

Ausgehend von diesen Problemen, die sich aus der dynamischen Instanziierung von Systemobjekten ergeben, lässt sich die Zielsetzung der nun folgenden Spezialisierungen ableiten.

4.1.2 Ziel

Aus dem gerade skizzierten Problemfeld ergibt sich demnach die folgende Zielsetzung: Verringerung der unnötigen Flexibilität in der Implementierung der Systemaufrufe mit dem Ziel der Verbesserung der Laufzeit- und Speichereffizienz, verbesserten Fehlerbehandlung ohne Beeinträchtigung der Verwendungsfreiheiten durch die Anwendungsentwickler. Dieses Ziel folgt dem Ansatz der Beantwortung der eingangs gestellten ersten Forschungsfrage (FF1) nach der Machbarkeit einer solchen Spezialisierung in Bezug auf den Systemstart und die Instanziierung von Systemobjekten.

In anderen Worten heißt dies, die Flexibilität, die die Programmierschnittstelle den Anwendungsentwicklern bietet, diese aber nicht ausnutzen, soll mit statischen Vorausberechnungen ersetzt werden. Dieser Austausch geschieht dabei transparent ohne Einbußen der wahrgenommenen Flexibilität und Freiheit in der Verwendung der gesamten Programmierschnittstellen. Gleichzeitig sollen durch die Spezialisierung die nichtfunktionalen Eigenschaften des Gesamtsystems verbessert werden. Insbesondere Speicherbedarf und Laufzeit sind hierbei die in dieser Arbeit primär betrachteten Größen. Verwandte Arbeiten haben aber bereits gezeigt, dass auch die Fehlertoleranz gegenüber Bitfehlern um den Faktor fünf steigt [Hof+14].

Aus dieser Beobachtung heraus erwächst die Fragestellung, ob nicht auch Systemaufrufe, die Systemobjekte generieren, funktional äquivalent zur Übersetzungszeit vorausberechnet werden können. Die Erwartung ist, dass hierdurch die Laufzeitkosten im Sinn von Ausführungszeit und Speicherbedarf geringer ausfallen und somit die nichtfunktionalen Eigenschaften eines so spezialisierten Systemaufrufs verbessert werden.

Im Folgenden wird dementsprechend die Fragestellung erörtert, ob eine statische Vorausberechnung vormals dynamisch erzeugter Systemobjekte möglich ist. Hierzu werden zuerst die benötigten Voraussetzungen ermittelt und im weiteren Verlauf die algorithmische und technische Umsetzung im Detail betrachtet.

4.2 Benötigtes Wissen

Ausgehend von der Annahme, dass der Systemstart regelhaft ohne große Variation linear verläuft und dabei zu einem immer gleichen arbeitsfähigen Zustand führt, benötigt die Spezialisierung Informationen über eben diesen Zustand, insbesondere über die zu instanziierenden Systemobjekte und deren Eigenschaften. Die hier zu spezialisierenden Systeme gehören dabei der Gattung der dynamisch konfigurierten Systeme an, sodass dieses Wissen mithilfe von statischen Analysen wie in Abschnitt 3.1 aus dem Anwendungscode gewonnen und im IIG gespeichert wird.

4.2.1 Phasen eines instanziierenden Systemaufrufs

Ein instanziierender Systemaufruf besteht aus mehreren Phasen, die zusammen das jeweils gewünschte Ergebnis erzielen. Die Phasen bestehen aus ⓐ Speicherallokation, ⓑ Initialisierung, ⓒ Registrierung im Betriebssystem und ⓔ Seiteneffekte und Rückgabewerte. Jede dieser Phasen benötigt gewisse Informationen über die konkret zu spezialisierenden Systemaufrufstellen und deren Aufrufkontexte. Das Betriebssystemmodell definiert hierbei, welche Informationen für welche Phase zwingend notwendig sind und welche optional bekannt sein können, um die Auswirkungen der jeweiligen Phasen statisch vorauszuberechnen. Gleichzeitig definiert es auch die Effekte der einzelnen Phasen, sodass diese statisch umgesetzt werden können.

ⓐ **Speicherallokation** Die erste Phase alloziert den Speicher, in dem das Systemobjekt leben wird. Für die Spezialisierung dieser Phase muss bekannt sein, dass das Systemobjekt sicher genau einmal existiert, da an einer Speicherstelle zeitgleich nur der Zustand genau eines Systemobjekts gespeichert werden kann und diese somit exklusiv diesem zugeordnet ist. Systemobjekte, die genau auf einem Aufrufpfad sicher erzeugt werden, entsprechen der Semantik eines statisch konfigurierten Systemobjekts. Anders ist es bei den Objekten anderer Häufigkeit. Sie resultieren in einer Überapproximation, falls ihr Aufrufpfad zur Laufzeit nicht getroffen wird oder können nicht statisch spezialisiert werden, wenn ihre Häufigkeit unbekannt ist. Neben der reinen Existenz muss auch die Größe des Speicherbereichs bekannt sein. Diese ist abhängig von

der Klasse des Objekts und eventuell von Parametern des Systemaufrufs. Zuletzt muss sichergestellt werden, dass das Objekt nie wieder aus dem System entfernt wird, da statisch allozierter Speicher nicht dynamisch freigegeben werden kann. Diese Phase ist essenzielle Grundlage für die Durchführbarkeit einer statischen Spezialisierung und Vorbedingung für alle weiteren Phasen. Am Beispiel eines Fadens ist die Größe des Laufzeitstapels notwendiger Parameter, um den Speicherbereich in der richtigen Größe zu allozieren. Für den Faden `fusion_task` ist zu erkennen, dass dieser sicher genau einmal mit einer Laufzeitstapelgröße von 512 B erzeugt wird. Die Speicherallokation der Fäden für die Sensordatenerfassung (`sensor_task`) und der dazu gehörigen Semaphore kann nicht statisch erfolgen, da nicht sicher gesagt werden kann, ob und wie häufig diese stattfindet, da die Ermittlung der angeschlossenen Sensoren nur zur Laufzeit durchgeführt werden kann.

b Initialisierung Während der Initialisierungsphase wird der Speicherbereich des Systemobjekts mit den vorgesehenen initialen Werten gefüllt. Die initialen Werte werden vom Betriebssystemmodell unter Verwendung der Aufrufparameter des Systemaufrufs bestimmt. Die Initialisierung kann abhängig von der Klasse des Systemobjekts auch nur in Teilen durchgeführt werden, sodass nicht alle Initialparameter zwingend erforderlich sind. Am Beispiel eines Fadens sind die auszuführende Einstiegsfunktion und deren Aufrufparameter Teil der Initialparameter. Diese können auch unabhängig voneinander in den initialen Aufrufrahmen geschrieben werden. Für den Faden `storage_task` sind alle Parameter bekannt, sodass diese Phase vollständig statisch vorausberechnet werden kann. Hingegen ist der Parameter der Einstiegsfunktion des Fadens `fusion_task` Laufzeitwissen, sodass diese Auswirkungen dieser Phase nur teilweise statisch vorausberechnet werden können.

c Registrierung im Betriebssystem Das Betriebssystem führt für einige Klassen von Systemobjekten Datenstrukturen, in denen diese verwaltet werden. Ist die designierte Position des Objekts in diesen berechenbar, so kann auch diese Phase statisch vorausberechnet und damit das Objekt im Betriebssystem registriert werden. Ein Faden wird beispielsweise in die Bereitliste des Planers eingehängt. Hierfür notwendig ist die Ausführungspriorität, sodass der Faden an der richtigen Position in dieser Liste eingehängt werden kann. Diese Phase kann nur statisch vorausberechnet werden, solange die Kontrolle der Datenstrukturen noch nicht an das Betriebssystem übergeben wurde, da nach der Übergabe sich der Inhalt verändern kann. Das Betriebssystemmodell definiert dabei den Zeitpunkt, der diese Übergabe vollzieht. Im Fall von FreeRTOS ist dies beispielsweise der Start des Planers, da ab dem Moment die Bereitliste dynamisch verändert wird. Demnach ist für den Faden `fusion_task` die Registrierung statisch möglich und für `storage_task` nicht.

d Seiteneffekte und Rückgabewert Zuletzt verbleiben die Seiteneffekte und der Rückgabewert des Systemaufrufs. Hierzu muss bekannt sein, auf welche Speicherbereiche sich diese auswirken. Je nach Spezifikation des Betriebssystemmodells sind diese eventuell abhängig von der Registrierung des Objekts im Betriebssystem. Am Beispiel der Fäden in FreeRTOS zeigt der Rückgabewert an, ob der Systemaufruf erfolgreich war. Dies erfordert kein Wissen über die Anwendung, sondern kann in jedem Fall gesetzt werden. Zusätzlich wird eine Referenz auf den erzeugten Fadenkontrollblock an eine per Referenz übergebene Speicheradresse geschrieben. Für diesen Seiteneffekt ist es notwendig, die Zieladresse statisch ermitteln zu können.

Die Instanzliste, die im vorherigen Kapitel ermittelt wurde, kann dementsprechend um die Informationen der statisch umsetzbaren Phasen erweitert werden. Tabelle 4.1 zeigt die um die Einordnung der möglichen Phasen ergänzte Tabelle für die Beispielanwendung. Damit kann nun ebenfalls die maximal mögliche Spezialisierungstiefe ermittelt werden.

Systemobjekte	fusion_task	storage_task	storage_queue	sensor_task	data_mutex
Erzeugt vor dem Start des Betriebssystems	✓	×	×	✓	✓
Genau einmal erzeugt	✓	✓	✓	×	×
Betriebssystem alloziert den Speicher	✓	✓	✓	✓	✓
Alle Parameter sind bekannt	×	✓	✓	×	✓
Wird nicht gelöscht	✓	✓	✓	✓	✓
ⓐ Speicherallokation	✓	✓	✓	×	×
ⓑ Initialisierung	(✓)	✓	✓	×	×
ⓒ Registrierung im Betriebssystem	✓	×	–	×	–
ⓓ Seiteneffekte und Rückgabewerte	✓	✓	✓	×	×
Maximal mögliche Spezialisierungstiefe	(✓)	(✓)	✓	×	×

Erzeugungsparameter:
fusion_task = {Typ=Faden, Laufzeitstapel=512, Funktion=fusion_entry, Initialparameter=unbekannt, ...}
storage_task = {Typ=Faden, Laufzeitstapel=512, Funktion=storage_entry, Initialparameter=NULL, ...}
sensor_task = {Typ=Faden, Laufzeitstapel=512, Funktion=sensor_entry, Initialparameter=unbekannt, ...}
storage_queue = {Typ=Puffer, Elemente=4, Elementgröße=sizeof(message_t), ...}
data_mutex = {Typ=Semaphor, Initialwert=1, ...}

Tabelle 4.1 – Statisch spezialisierbare Phasen und maximale Spezialisierungstiefe der Beispielanwendung. Bekannte Informationen über die Systemobjekte der Beispielanwendung, Einordnung der statisch durchführbaren Phasen des jeweiligen instanziierenden Systemaufrufs und daraus resultierende Klassifizierung der maximalen Spezialisierungstiefe. Symbolbedeutung: ✓ wahr/vollständig möglich; (✓) partiell möglich; × falsch/nicht möglich; – nicht zutreffend

4.2.2 Klassifizierung möglicher Spezialisierungen

In Abschnitt 2.2.1.2 wurden die Ebenen der Spezialisierung als Abstraktions-, Instanz- und Interaktionsebene als verschiedene Intensitätsstufen von Spezialisierungen eingeführt. Wir befinden uns hier auf der Ebene der Instanzen, da einzelne Instanzen von Systemobjekten spezialisiert werden. Eine differenziertere Klassifizierung der Spezialisierungen ist anhand zweier weiterer Kriterien vollziehbar: Die *Spezialisierungstiefe* bestimmt, wie viel des ursprünglichen Systemaufrufs an der Aufrufstelle verbleibt. Die *Spezialisierungskompatibilität* beschreibt, ob die nach der Spezialisierung vorhandenen Systemobjekte kompatibel mit den generischen Systemaufrufen der ursprünglichen Betriebssystemimplementierung bleiben.

Tiefe der Spezialisierung

Die Spezialisierung eines dynamisch instanziierenden Systemaufrufs in ein statisch instanziiertes Systemobjekt bedeutet grundlegend die Verlagerung der Auswertung seiner Auswirkungen von der Laufzeit in die Übersetzungszeit. Anhand des gewonnenen Wissens über eine Systemaufrufstelle und das dadurch erzeugte Systemobjekt ist ermittelbar, welche Phasen des Systemaufrufs (ⓐ–ⓓ) in die Übersetzungszeit verlagerbar sind. Dies bedeutet unterschiedlich tiefgreifende Spezialisierungsmöglichkeiten. Anhand der Menge der verlagerbaren Phasen ist nun eine Einordnung in drei Kategorien der *Spezialisierungstiefe* möglich.

vollständig spezialisierbar Die Spezialisierung ist in der Lage, die gesamten Auswirkungen des Systemaufrufs bereits während der Übersetzungszeit abzubilden. Die Auswirkungen aller

Phasen können statisch durchgesetzt werden, sodass zum Zeitpunkt der Ausführung keine Aktion mehr erforderlich ist, um die Auswirkungen des Systemaufrufs zu erzielen. Der entsprechende Systemaufruf kann somit vollständig von seiner ursprünglichen Aufrufstelle eliminiert werden. Die Seiteneffekte und Rückgabewerte werden direkt an den Verwendungsstellen mit den richtigen Werten umgesetzt. Quellcode 4.1 zeigt entsprechende Änderungen auf C-Code Ebene. Dies trifft beispielsweise auf die Instanziierung der Warteschlange `storage_queue` der Beispielanwendung (Zeile 36) zu. Diese wird sicher erzeugt und alle dafür notwendigen Informationen sind statisch bekannt.

Quellcode 4.1 Ausschnitt der Änderungen der vollständig spezialisierten Instanziierung der Warteschlange `storage_queue` im Anwendungscode, dargestellt in äquivalenten C-Code Änderungen.

```
- QueueHandle_t storage_queue;
+ char static_storage_queue_data[sizeof(message_t)*4];
+ Queue_t static_storage_queue = {.head = static_storage_queue_data, .elements=4, /*...*/};
+ QueueHandle_t storage_queue = &static_storage_queue;

- storage_queue = xQueueCreate(sizeof(message_t), 4);
```

partiell spezialisierbar Teile der Auswirkungen des Systemaufrufs sind bereits während der Übersetzungszeit mittels Spezialisierung ausführbar. Mindestens die Phase ⓐ Speicherallokation ist statisch möglich. Es verbleibt jedoch ein Teil der Auswirkungen der anderen Phasen des Systemaufrufs, welcher zwingend während der Laufzeit hervorgerufen werden muss. Der entsprechende Systemaufruf kann dementsprechend modifiziert werden, dass nur die verbleibenden Phasen ausgeführt werden. In der Beispielanwendung zählt der Faden `fusion_task` in diese Kategorie, da der Initialparameter erst zur Laufzeit ermittelt und in den initialen Aufrufrahmen geschrieben werden kann (Zeile 25). Dementsprechend verbleibt diese Aktion an der Aufrufstelle. Die entsprechenden Änderungen sind exemplarisch in Quellcode 4.2 dargestellt.

Quellcode 4.2 Ausschnitt der Änderungen der partiell spezialisierten Instanziierung des Fadens `fusion_task`, dargestellt in äquivalenten C-Code Änderungen.

```
- TaskHandle_t fusion_task;
+ TaskHandle_t fusion_task = &fusion_task_tcb;
+ char fusion_task_stack[512];
+ TCB_t fusion_task_tcb = {.prio=1, .stack=&fusion_task_stack, /*...*/};

- xTaskCreate(fusion_entry, "fusion_task", 512, si, 1, &fusion_task);
+ set_initial_parameter(&fusion_task_stack, si);
```

nicht spezialisierbar Die Auswirkungen des Systemaufrufs sind nicht in die Übersetzungszeit verlagerbar. Der ursprüngliche Systemaufruf muss unverändert während der Laufzeit ausgeführt werden, um die gewünschten Effekte zu erzielen. In diese Kategorie fallen die Fäden zur Sensordatenerhebung (`sensor_task`) in Zeile 27, da statisch aufgrund der umgebenden Schleife nicht bekannt ist, wie viele Instanzen tatsächlich hier erzeugt werden.

Kompatibilität der Spezialisierung

Die zweite grundlegende Frage, die es bei der Auswahl der Spezialisierung zu treffen gilt, ist die nach der Kompatibilität der statisch instanziierten Systemobjekte zu den bisher dynamisch instanziierten. Hierbei unterscheidet man in *brechende*- und *nicht-brechende* Spezialisierungen.

Quellcode 4.3 Ausschnitt der Beispielanwendung der nicht spezialisierenden Fadenerzeugung. Aufgrund der Schleife mit dynamischer Eingabe ist diese Instanziierung nicht spezialisierbar.

```
for (int i=0; i < si->length; i++) {
  xTaskCreate(sensors_entry, "sensor_task", 512, &si->sensors[i], 2,
              &(si->sensors[i]->task));
```

Sind die statisch instanziierten Systemobjekte binärkompatibel zu solchen, die mittels der dynamischen Systemaufrufe instanziiert werden, so heißt dies, die Spezialisierung ist *nicht brechend*. Dies bedeutet, dass alle Systemaufrufe, die zur Kommunikation mit solchen Systemobjekten existieren, unverändert verwendet werden und zur Laufzeit kein Unterschied zwischen statisch und dynamisch instanziierten Systemobjekten vorhanden ist. Einzige Ausnahme ist, dass statisch instanziierte Systemobjekte nicht aus dem System gelöscht werden können. Damit haben nicht-brechende Spezialisierungen den Vorteil, dass nicht unterschieden werden muss, wie ein Systemobjekt instanziiert wurde, noch bedarf es angepasster Implementierung von Systemaufrufen, um mit diesen später zu interagieren.

Im Gegensatz dazu existieren auch *brechende* Spezialisierungen. Diese sind nicht kompatibel zu den bestehenden Systemaufrufimplementierungen, sodass zur Interaktion mit diesen darauf angepasste Implementierungen vom Betriebssystemmodell bereitgestellt werden müssen. Der Vorteil dieser besteht darin, dass auch die zugrunde liegenden Datenstrukturen und Algorithmen an die Verwendung in der konkreten Anwendung angepasst werden können. Somit sind weitere Verbesserungen der nichtfunktionalen Eigenschaften möglich, die Binärkompatibilität wird jedoch dafür aufgegeben.

Aus Sicht der konkreten Anwendung bleiben die funktionalen Eigenschaften aller verwendeten Interaktionen sowohl bei brechenden als auch bei nicht-brechenden Spezialisierungen unverändert. Einzig die nicht genutzten Teile sind von Veränderungen betroffen, aber da diese ja per Definition nicht von dieser Anwendung verwendet werden, ist die Veränderung irrelevant.

In diesem Kapitel verwende ich ausschließlich nicht-brechende Spezialisierungen, da diese genügen, um den prinzipiellen Nutzen der statischen Instanziierung zu demonstrieren. Damit bleibt stets die gesamte Flexibilität der Interaktion zwischen den Systemobjekten vorhanden. Die Spezialisierung des Systemstarts und der Initialisierungen hat damit keinen weiteren Einfluss auf die späteren Interaktionen zwischen den Systemobjekten, insbesondere wird hier keine Flexibilität der Interaktionen verloren. Im nächsten Kapitel führe ich auch brechende Spezialisierungen ein, die die verwendeten Datenstrukturen und Algorithmen verändern, um auch die Interaktionen zu spezialisieren.

4.3 Umsetzung der Spezialisierungen

Die konkrete Umsetzung der Spezialisierung erfolgt den Phasen eines Systemaufrufs entsprechend ebenfalls in mehreren Schritten. Diese setzen jeweils den entsprechenden Teilaspekt, angefangen bei der Selektion der umzusetzenden Spezialisierung bis hin zur Bekanntmachung und Einpassung in das reguläre Betriebssystem um.

Die Implementierung dieser automatischen Spezialisierung habe ich als Komponente in dem Analyse- und Spezialisierungswerkzeug ARA vorgenommen, welches zu diesem Zweck von Gerion Entrup und mir entwickelt wurde. ARA arbeitet als Teil der LLVM-Übersetzerkette [LA04] und gliedert sich dort als Teil der Bindezeitoptimierung mit ein. Die Spezialisierungen werden dabei je nach Art als direkte Änderungen des Zwischencodes oder als Erzeugung neuen C/C++-Codes

vorgenommen. Diejenigen Teilschritte der Spezialisierung, die einer direkten Modifikation des gegebenen Anwendungscodes entsprechen, sind auch als direkte Modifikationen umgesetzt. Allein schon das Vorhandensein der zu ändernden Programmcodezeilen erzwingt diese Vorgehensweise. An dieser Stelle wäre keine andere Möglichkeit der Veränderung vorstellbar, da der Anwendungscode nur in dieser Form vorliegt. Diejenigen Spezialisierungen, die das Erzeugen neuer Objekte, Funktionen oder Implementationen beinhalten, erzeugen neuen C/C++-Code. Diese Wahl ermöglicht es, dass dieser Programmcode ebenso wie der Anwendungscode mit den gleichen Konfigurationsparametern von Präprozessor und Übersetzer in LLVM-Code übersetzt wird. Dies ist vorteilhaft, da, obwohl der Zwischencode unabhängig von der späteren Ausführungsplattform sein soll [LA04], diverse Hardwareabhängigkeiten existieren. Die Größe von zusammengesetzten Datenstrukturen und Datenworten werden so bereits während der Übersetzung von C-Code in Zwischencode zu Konstanten im Programmcode, die keinen Rückschluss auf ihre Bedeutung als hardwareabhängige Konstanten zulassen.

Abbildung 4.1 zeigt einen Überblick des gesamten Spezialisierungsprozesses, wie er in ARA implementiert ist. Der Anwendungscode wird dabei von links in den Prozess eingegeben, durchläuft die Analyse- und Spezialisierungsschritte und verlässt diesen als modifizierter Anwendungscode auf der rechten Seite. Gleichzeitig produziert ARA eine spezialisierte Betriebssystemimplementierung, die gemeinsam mit den für die Anwendung verwendeten Betriebssystemparametern übersetzt wird. Anschließend werden die modifizierte Anwendung und die generierten Komponenten als eine Einheit optimiert. Dies ermöglicht die Ausführung der regulären Bindezeitoptimierungen der LLVM-Werkzeuge und dementsprechend der Berechnung und Ersetzung von Bindezeitkonstanten. Hierzu gehören sowohl Größenberechnungen als auch insbesondere die Ersetzung der Adressen der statisch instanziierten Systemobjekte an den entsprechenden Verwendungsstellen.

4.3.1 Auswahl der Spezialisierung

Ausgehend von der Menge der möglichen Spezialisierungen muss die Wahl getroffen werden, welche konkrete Variante umgesetzt werden soll. Standardmäßig wird hierbei diejenige Variante gewählt, die die größte Spezialisierungstiefe aufweist, jedoch keine brechenden Veränderungen hervorruft.

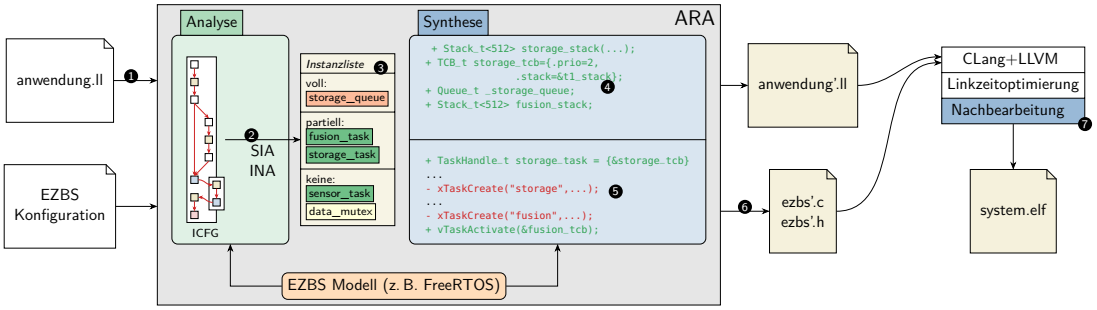

Abbildung 4.1 – Integration der Bestandteile von ARA in den LLVM-Übersetzungsprozess. Als Eingabe werden die Anwendung und die Betriebssystemkonfiguration eingelesen ❶ und mittels SIA und INA analysiert ❷. Die Systemobjekte werden im IIG verzeichnet und anhand der Durchführbarkeit der Spezialisierung klassifiziert ❸. Statische Systemobjekte werden samt Initialisierung generiert ❹ und die Aufrufstellen angepasst ❺, sodass als Ausgabe die modifizierte Anwendung und das spezialisierte Betriebssystem erzeugt werden ❻. Diese werden der regulären LLVM-Überzetzerwerkzeugkette zum Binden und Optimieren zugeführt ❼, sodass das finale Systemabbild entsteht. In Anlehnung an [▷Fie+21]

Dies bedeutet, dass alle Auswirkungen des instanziierenden Systemaufrufs zur Übersetzungszeit umgesetzt werden, falls möglich. Durch Vorgaben von außen kann jedoch eine maximale Spezialisierungstiefe vorgegeben werden, die nicht überschritten werden darf. Die resultierenden Systemobjekte gleichen anschließend den regulär erzeugten und sind, soweit möglich, dem Betriebssystem und der Anwendung an den entsprechenden Stellen bekannt gegeben.

4.3.2 ⓐ Speicherallokation

Der fundamentale Schritt für die statische Instanziierung ist die Allokation eines passenden Speicherbereichs. Für alle Systemobjekte, die sicher erzeugt werden und fortan bis zum Ende der Programmlaufzeit existieren, ist es überflüssig, ihren Speicher dynamisch zu allozieren. Eine statische Speicherallokation entspricht viel eher ihrer Lebenszeit. Hierzu werden globale Variablen passenden Typs und Größe angelegt. Dies führt zu statisch bekannten festen Adressen der Systemobjekte und gleichzeitig entfällt das Bedürfnis nach einer dynamischen Speicherverwaltung für diese Systemobjekte. Damit entfällt der für die Allokation notwendige Laufzeit- und Verwaltungsaufwand und gleichzeitig sind die Objekte und deren Verwendung resilienter gegenüber Fehlern [Hof+14].

4.3.3 ⓑ Initialisierung

Nach dem Vorhandensein des bloßen Speicherbereichs ist dessen Initialisierung der nächste Schritt. Alle Werte, die aufgrund der Aufrufparameter des Systemaufrufs bereits bekannt sind, werden verwendet. Ebenso ergibt die Interpretation der Initialisierungsfunktionen des Betriebssystems Werte, die im Rahmen der statischen Initialisierung an die entsprechenden Speicherstellen geschrieben werden können. Das Betriebssystemmodell bestimmt hierbei, welche Werte zwingend notwendig sind und ob es Abhängigkeitsbeziehungen zwischen diesen gibt. Dadurch ist es auch möglich, diesen Schritt nur teilweise auszuführen und die verbleibenden Teile weiterhin an der Aufrufstelle zu belassen.

Bei Fäden wird der initiale Aufrufrahmen auf dem Laufzeitstapel vorbereitet und mit den initialen Parametern befüllt, falls diese bekannt sind. Ebenso beinhalten die Verwaltungsstrukturen für Synchronisationsobjekte ihren momentanen Zustand. Hierzu zählen beispielsweise die Größe der zu verwaltenden Objekte, Zeiger auf Schreib- und Lesepositionen sowie initiale Zählerstände.

Zur Initialisierung der Systemobjekte verwendet ARA Konstruktoren und Initialisierungslisten für die zuvor genannten globalen Variablen. Dadurch wird sichergestellt, dass die Werte vor der ersten Verwendung an die richtige Stelle im Speicher geschrieben werden.

In Quellcode 4.4 ist ein verkürzter Ausschnitt der für die Beispielanwendung generierten Systemobjekte zu sehen. Für den Faden `storage_task` sind dort Laufzeitstapel und Fadenkontrollblock abgebildet. Die Initialisierung des Fadenkontrollblocks ist hierbei als Initialisierungsliste für die Wertinitialisierung gegeben. Dies hat zur Folge, dass alle Einträge initialisiert werden, auch die ohne nicht explizit angegebenen Wert. Im Gegensatz dazu sind für den Laufzeitstapel nur die Werte für den initialen Aufrufrahmen für den Konstruktor gegeben, sodass nur diese durch den Konstruktor geschrieben werden müssen. Die übrigen Felder des Laufzeitstapels dürfen uninitialisiert bleiben, da deren Initialisierung erst durch Anlegen eines Aufrufrahmens und Initialisierung der dort liegenden Variablen erfolgt.

Im Gegensatz dazu verbleibt für den Faden `fusion_task` das Setzen des Aufrufparameters an der ursprünglichen Stelle des Systemaufrufs, da der Wert erst zur Laufzeit bestimmt wird. Die anderen Werte hingegen werden weiterhin statisch initialisiert.

Quellcode 4.4 Ausschnitt des durch ARA generierten Programmcodes zur statischen Initialisierung des Fadens `storage_task`.

```
1 StackType_t<512> _storage_task_stack(&storage_entry);
2 TCB_t _storage_task_tcb = {
3   .uxPriority = 3,
4   .pxTopOfStack = &_storage_task_stack + 495,
5   .pxStack =      &_storage_task_stack,
6 };
```

4.3.4 Ⓒ Registrierung im Betriebssystem

Neben den Datenstrukturen, die die jeweiligen Systemobjekte repräsentieren, werden auch die betriebssysteminternen Datenstrukturen statisch vorbereitet und mit den richtigen Werten belegt. Dies geschieht, da auch die statisch instanziierten Systemobjekte dem Betriebssystem bekannt sein müssen, als wären sie zur Laufzeit instanziiert. Diese beinhalten beispielsweise die Bereitliste des Planers. Die Fäden werden in diese statisch eingehängt, sodass der Planer diese dort erwartungsgemäß vorfindet. Ebenso trifft dies auch auf alle internen Werte zu, die ausschließlich vom Betriebssystem zur internen Verwaltung verwendet werden. Hierzu zählen beispielsweise Zähler, die die Anzahl vorhandener Systemobjekte eines Typs vermerken.

Das Vorgehen unterscheidet sich hierbei anhand des logischen Ausführungszeitpunkts. Die Startphase besteht dabei aus zwei Abschnitten, die voneinander getrennt werden durch den Zeitpunkt, an dem die Kontrolle über eben diese Datenstrukturen an das Betriebssystem übergeben wird.

Das System startet zunächst mit der Initialisierungsphase, die einem erweiterten Laden von Initialwerten entspricht. Die Registrierungen, die in dieser Zeit stattfinden, werden umgesetzt, indem die Datenstrukturen direkt mit den resultierenden Werten initialisiert werden. Dies geschieht, indem die statisch instanziierten Systemobjekte analog zu dem Verhalten, welches das Betriebssystem zur Laufzeit zeigt, in dessen Datenstrukturen eingehängt werden. Es ist keine weitere Aktion zur Laufzeit mehr notwendig. Quellcode 4.5 zeigt einen verkürzten Ausschnitt aus dem für die Beispielanwendung generierten Code. Zu sehen ist die Bereitlisten des FreeRTOS Planers, in die der Kontrollblock des Fadens `fusion_task` bereits eingehängt ist. Ebenfalls ist das entsprechende Bit im Bitfeld der Prioritäten gesetzt, welches anzeigt, dass ein lauffähiger Faden vorhanden ist.

Im Gegensatz dazu benötigen Registrierungsvorgänge, die nach der Übergabe der Datenstrukturen an das Betriebssystem geschehen, weiterhin Laufzeitaktionen. Dies bedeutet, dass an der Stelle des ursprünglichen erzeugenden Systemaufrufs weiterhin ein Systemaufruf verbleibt. Dieser führt jedoch nur noch die Registrierung und nicht mehr die vorherigen Teilschritte des ursprünglichen Systemaufrufs durch. Das verwendete Betriebssystemmodell muss diese entsprechend zur Verfügung stellen. Für den Faden verbleibt dementsprechend für den Faden `storage_task`, wie in Quellcode 4.6 dargestellt, der Ruf zu dem angepassten Systemaufruf `vTaskActivate()`, der den Faden in die Bereitliste einreiht.

4.3.5 Ⓓ Seiteneffekte und Rückgabewert

Zuletzt verbleibt von den ursprünglichen Systemaufrufen noch die Auswirkungen außerhalb des Aufrufs sichtbar zu machen, also dessen Seiteneffekte und Rückgabewerte nach außen zu propagieren. Die Auswirkungen der instanziierenden Systemaufrufe innerhalb des Betriebssystems sind nun bereits (teilweise) durch die vorherigen Schritte in die Übersetzungszeit verlagert. An den Aufrufstellen stehen jedoch weiterhin die regulären Systemaufrufe, die nun falsche oder überflüssige Aktionen hervorrufen würden. Dementsprechend ist der nächste Schritt, die Aufrufstellen so anzupassen, dass

Quellcode 4.5 Ausschnitt des durch ARA generierten Programmcodes zur statischen Registrierung des Fadens `fusion_task` im Betriebssystem.

```
/* Bereitlisten des Planers */
List_t pxReadyTasksLists[5] = {
  ..., /* Priotität 1 = */ {
    .xListEnd = {
        .pxNext =     (ListItem_t *) &_fusion_task_tcb.xStateListItem,
        .pxPrevious = (ListItem_t *) &_fusion_task_tcb.xStateListItem
    },
    .uxNumberOfItems = 1,
    .pxIndex = (ListItem_t*) &pxReadyTasksLists[2].xListEnd
  }, ...
};

/* Bitfeld der Prioritäten mit lauffähigen Fäden */
volatile unsigned int uxTopReadyPriority = 1 << 0;
```

Quellcode 4.6 Ausschnitt aus dem durch ARA modifizierten Anwendungscode. Der ursprüngliche Aufruf von `xTaskCreate()` ist durch `vTaskActivate()` ersetzt. Die Umsetzung geschieht in IR, zur leichteren Lesbarkeit ist es hier dargestellt in C.

```
- xTaskCreate(storage_entry, "storage_task", 512, NULL, 3, &storage_task);
+ storage_task = &storage_task_tcb;
+ vTaskActivate(&storage_task_tcb, 3);
```

die soeben spezialisierten Systemobjekte verwendet werden und die eventuell noch notwendigen Teile der Systemaufrufe ausgeführt werden. Insbesondere bedeutet dies, die Verwendungsstellen von Rückgabewerten zu ersetzen. Dies betrifft auch per Referenz übergebene Ausgabeparameter mit entsprechenden Werten zu beschreiben. Je nach Zustand des Betriebssystems und Schnittstelle des Systemaufrufs wird dabei unterschiedlich vorgegangen.

An allen Verwendungsstellen des Rückgabewertes wird direkt der entsprechende Wert eingesetzt. Für die Referenzparameter, die als Rückgabekanal verwendet werden, werden entsprechende Zuweisungsoperationen eingefügt.

In Quellcode 4.7 ist ein Ausschnitt aus dem von ARA spezialisierten Programmcode und den darin enthaltenen Änderungen für die Umsetzung der Seiteneffekte abgebildet. Dabei sind die Spezialisierungen zum Zweck des leichteren Verständnisses in äquivalentem C-Code dargestellt. Die tatsächliche Veränderung durch ARA findet innerhalb der als IR-Code gegebenen Anwendung statt und ist dementsprechend als Modifikation von eben diesem IR-Code implementiert. Der Aufruf des Systemaufrufs, der `storage_queue` erzeugt, ist vollständig ersetzt. Einzig die Variable `storage_queue` wird direkt mit der entsprechenden Adresse initialisiert.

Quellcode 4.7 Ausschnitt aus dem durch ARA modifizierten Anwendungscode. Der ursprüngliche Aufruf von `xQueueCreate()` ist nicht mehr vorhanden, es verbleibt lediglich die Zuweisung des Rückgabewerts. Die Umsetzung geschieht in IR, zur leichteren Lesbarkeit ist es hier dargestellt in C.

```
- storage_queue = xQueueCreate(sizeof(message_t), 4);
+ storage_queue = &static_storage_queue;
```

4.3.6 Rekonfiguration des Betriebssystems

Betriebssysteme sind üblicherweise mindestens auf Ebene der Abstraktionen konfigurierbar [▷Fie+18]. Dementsprechend muss eine Anwendung eine für sie notwendige Betriebssystemkonfiguration beschreiben. Je nach Anforderungen der Anwendung wurde durch die Entwickler der gegebenen Anwendung eine Konfiguration des Betriebssystems ausgewählt, mit der diese auf Basis des Betriebssystems arbeitsfähig ist. Diese Konfiguration des Betriebssystems muss ebenfalls den durch die spezialisierte Anwendung veränderten Anforderungen entsprechend angepasst werden. Das Betriebssystemmodell beschreibt hierbei die Abhängigkeiten zwischen den einzelnen Abstraktionen, deren Implementierungsvarianten und deren Beziehungen zu anderen Betriebssystemkomponenten. Anhand dieser Abhängigkeiten führt ARA die Rekonfiguration automatisch durch.

Angenommen, die Anwendung hat bisher alle Systemobjekte vollständig dynamisch erzeugt. Aufgrund der vorangegangenen Spezialisierungsschritte werden einige Fäden statisch alloziert, initialisiert und deren Seiteneffekte umgesetzt. Es verbleibt jedoch die Registrierung im Betriebssystem. Die Konfiguration wird dementsprechend so geändert, dass der dafür notwendige Systemaufruf `vTaskActivate()` verfügbar ist. Ebenfalls werden diejenigen Systemaufrufe in der Konfiguration deaktiviert, die nicht mehr benötigte Funktionalität bereitstellen.

Die dynamische Erzeugung von Systemobjekten setzt voraus, dass eine Freispeicherverwaltung existiert. Werden nun Systemobjekte statisch alloziert, sinkt die maximal dynamisch angeforderte Speichermenge um deren Größe. Die Konfiguration der Freispeicherverwaltung wird daher derart verändert, dass kein Speicher mehr für die nun statisch allozierten Systemobjekte mehr vorgehalten wird. Dies führt zu einer Verringerung des reservierten Speicherbereichs und kann bei vollständiger Abdeckung aller Systemobjekterzeugungen die Notwendigkeit dieser vollständig obsolet werden lassen.

Aller Betriebssystemcode wird so umkonfiguriert, dass die bereits statisch erzeugten Verwaltungsstrukturen verwendet werden, anstatt neue zu erzeugen. Das Betriebssystemmodell beschreibt die jeweils notwendigen Änderungen der konkreten Betriebssystemimplementierung. Am Beispiel des Planers in FreeRTOS bedeutet dies, dass die bereits gefüllte Bereitliste verwendet wird. Ohne Spezialisierung war es Teil der Initialisierung, diese Liste anzulegen, nun aber existiert diese bereits und ist mit den einzuplanenden Fäden gefüllt.

4.3.7 Bindezeitoptimierungen und die richtige Darstellung von initialisierten Datenstrukturen

Zum Abschluss der Spezialisierungen wird das Gesamtsystem gebunden und somit die modifizierte Anwendung mit dem dazu passend generierten Betriebssystem zusammengeführt. In diesem Schritt, durchgeführt durch den regulären Binder, führt dieser Optimierungsvorgänge wie z. B. Konstantenpropagation inklusive Adressberechnung und inzeilige Funktionsersetzung durch. Die Adressberechnung schließt ebenso diejenigen Adressen mit ein, die noch einen konstanten Versatz enthalten und erst beim Platzieren im Speicherabbild vom Lader den finalen Wert erhalten.

Das Initialisieren mit konstanten Werten führt für die meisten Systemobjekte zu einem vorteilhaften Laufzeitverhalten im Zuge der Initialisierung, da die Werte beim initialen Laden des Speicherabbildes direkt richtig sind und es keines weiteren Programmcodes bedarf, diese zu initialisieren. Problematisch ist dies aber für diejenigen Datenstrukturen, die nur dünn mit variierenden Werten besetzt sind und ansonsten zum überwiegenden Teil denselben Wert (zumeist Nullen) enthalten. Dies trifft insbesondere auf Laufzeitstapel und Puffer zu, die durch die statische Instanziierung mit entsprechenden Werten initialisiert werden. Diese enthalten einen kleinen initialen Inhalt, z. B. den initialen Aufrufrahmen oder die Position der Lese- und Schreibzeiger, sind aber ansonsten leer

(keine weiteren Aufrufrahmen und keine mit Daten belegten Pufferplätze). Es sind jedoch auch Datenstrukturen der Anwendung oder ihrer Bibliotheken von diesem Problem betroffen. Grundlegendes Problem dabei ist, dass dem Anwendungsentwickler keine reguläre Möglichkeit gegeben ist, Datenstrukturen als dünn besetzt zu deklarieren und dementsprechend zu initialisieren. Durch das Vorhandensein von Initialwerten abweichend von null werden diese Datenstrukturen automatisch in das Datensegment abgelegt. Dies führt dazu, dass anders als bei der vorherigen Allokation im BSS-Segment oder mittels dynamischer Speicherverwaltung alle Einträge der Datenstruktur aus dem Programmspeicher in den Arbeitsspeicher kopiert werden müssen, anstatt die entsprechende Speicherbereiche nur mit Nullen zu initialisieren. Dies trifft nicht nur auf explizit mit Daten initialisierte Felder und Strukturen zu, sondern betrifft auch solche, deren Initialisierung von Konstruktoren vorgenommen wird. Da diese Konstruktoren nun mit zur Übersetzungszeit konstant berechenbaren Parametern aufgerufen werden, transformieren die in LLVM enthaltenen Optimierungsalgorithmen diese Konstruktoraufrufe in Dateninitialisierungen. Dabei findet keine Abwägung der nichtfunktionalen Eigenschaften statt, sondern es wird grundsätzlich davon ausgegangen, dass ein initialisierter Datenbereich einem Konstruktoraufruf vorzuziehen ist.

Die ersten Schritte des Startvorgangs eines Rechensystems bestehen aus dem Laden des Datensegments und dem Nullen des BSS-Segments [Bac+18]. Erste Evaluationen haben ergeben, dass die Verlagerung vom BSS-Segment in das Datensegment zu einer deutlichen Verschlechterung der Startgeschwindigkeit führt. Dieser Verschlechterung kann entgegengewirkt werden, indem der besonderen Struktur der zu initialisierenden Daten Beachtung geschenkt wird und neben dem Daten- und BSS-Segment ein weiteres Segment für dünn besetzte Datenstrukturen im Programmabbild angelegt wird. Durch geeignete Speicher- und Ladestrategien kann dieses Segment ebenso effizient geladen und initialisiert werden, wie wenn sie im BSS-Segment lägen und anschließend die wenigen von null abweichenden Werte gesetzt würden. Eine mögliche Wahl einer solchen Strategie ist die *Lauflängencodierung (engl. run-length encoding, kurz RLE) (RLE)* [Sal07], welche wiederholende Werte als Paar aus dem zu wiederholenden Wert und der Anzahl der Wiederholungen repräsentiert. Frühe Anwendung fand diese Codierung bereits bei der Übertragung von Televisionsbildern [RC67; Tsu+84] und eignet sich besonders gut, da nur minimale Ressourcen (Zahl der Register und zu lesende Werte je zu schreibendem Wert) zum Dekomprimieren der Daten benötigt werden. Diese Repräsentation verringert deutlich die notwendigen Leseoperationen pro zu schreibendem Speicherwort, sodass eine Annäherung an die für die Initialisierung des BSS-Segments benötigte Zeit erreicht werden kann.

4.3.8 Kosten der Spezialisierung

Das Betriebssystemmodell beschreibt die Anforderungen an das Wissen über die jeweiligen Systemaufrufe und die daraus resultierende mögliche Tiefe der Spezialisierung. Anhand dieses Modells und dem gegebenen Optimierungsziel der nichtfunktionalen Eigenschaften kann die Menge der möglichen Spezialisierungen bestimmt werden. Es ist dabei keineswegs zwingend, jede mögliche Spezialisierung vorzunehmen, da abhängig von den zu optimierenden nichtfunktionalen Eigenschaften nicht alle Spezialisierungen einen Vorteil herbeiführen. So führen manche partiell durchgeführten Spezialisierungen zwar zu einem Laufzeitvorteil, erzwingen aber zeitgleich leicht erhöhten Speicherbedarf. Dementsprechend kann es sinnvoll sein, gewisse partielle Spezialisierungen auszuschließen und eine Instanziierung nicht zu spezialisieren, sollte eine vollständige Spezialisierung nicht möglich sein.

Für die Fäden der Beispielanwendung aus Quellcode 3.1 stellt Tabelle 4.1 die jeweils maximal mögliche Spezialisierungstiefe anhand der vorhandenen Informationen dar. Demnach ist die Instanziierung 1 vollständig spezialisierbar, da alle notwendigen Parameter bekannt sind. Hingegen kann

2 nicht vollständig spezialisiert werden, da er erst zur Laufzeit nach dem Beginn des Mehraufgabenbetriebs instanziiert wird. Dementsprechend muss mindestens das Einreihen in die Bereitliste des Aufgabenplaners an der Aufrufstelle des Systemaufrufs verbleiben. Eine Spezialisierung von 3 ist nicht möglich, da dessen Existenz nicht mit Sicherheit bestimmt werden kann.

Die Selektion einer optimalen Auswahl der möglichen Spezialisierungen ist ein mehrdimensionales Optimierungsproblem, dessen Bewertungsfunktion nicht allgemein gegeben ist, sondern anwendungsspezifisch bestimmt werden muss. Im Folgenden präsentiere ich die Auswirkungen der vorgestellten Spezialisierungen auf Speicher- und Laufzeitbedarf. Weitere Kriterien wie beispielsweise Fehlertoleranz und Angriffssicherheit sind nicht Teil dieser Arbeit. Andere Arbeiten zeigen jedoch, dass sie betrachtenswert sind [Hof+14; DHL17; Zie+19]. Je nach Rahmenbedingungen des jeweiligen Systems sind dementsprechende Abwägungen notwendig, zu denen die im Folgenden präsentierten Ergebnisse als Entscheidungsbasis dienen.

4.3.9 Aktualisierbarkeit der Anwendung

Für das Schließen von Sicherheitslücken oder ergänzen fehlender Funktionalität ist die Aktualisierbarkeit einer Anwendung auch für eingebettete Systeme eine wichtige Funktionalität. Die statische Spezialisierung hat einen Einfluss auf diese Möglichkeiten, da die Betriebssystemschnittstelle auf die Verwendung durch die gegebene Anwendung in ihrer ursprünglichen Version angepasst wird. Dies ist jedoch kein grundsätzliches Veto für eine Aktualisierung, sondern lediglich eine partielle Einschränkung. Für die hier vorgestellten Spezialisierungen resultiert die Einschränkung, dass ohne das Betriebssystem mit zu aktualisieren, die Änderungen nicht die Eigenschaften und Mengen der vorhandenen Systemobjekte beeinflussen dürfen. Jede Änderung, die nur die Anwendungslogik ohne Einfluss auf Systemobjektinstanziierungen zur Folge hat, ist problemlos umsetzbar. Nur diejenigen Änderungen, die Systemobjekte betreffen, sind genauer zu betrachten.

Werden in der neuen Version der Anwendung zusätzliche Systemobjekte benötigt, so können diese problemlos dynamisch in partiellen Aktualisierungen hinzugefügt werden, da hier nur nicht brechende Spezialisierungen verwendet wurden. Wurden die dazu notwendigen Systemaufrufe auskonfiguriert, so können sie mit dem aktualisierten Anwendungscode nachgeliefert werden. Für Systemobjekte, die nicht mehr benötigt werden, kann entsprechender Code der Aktualisierung automatisch hinzugefügt werden, der diese aus dem System deregistriert. Damit belegen sie zwar weiterhin wie in der ursprünglichen Version Speicher, verfälschen aber nicht die funktionalen Eigenschaften des Gesamtsystems.

Einzig Systemobjekte, die in der neuen Version weniger als bisher statisch spezialisierbar sind, stellt eine gravierende Änderung der Instanzliste dar. Nur solche gravierenden Änderungen führen zu der Notwendigkeit, den gesamten bestehenden spezialisierten Programmcode auszutauschen, da eine rückwirkende Generalisierung nicht möglich ist.

4.4 Evaluation

Die soeben beschriebenen Spezialisierungen habe ich in ARA wie in Abbildung 4.1 dargestellt, implementiert. Die Implementation besteht dabei sowohl aus Komponenten, die in Python geschrieben sind, als auch solche, die in C++ geschrieben sind. Die Wahl der Sprache hängt dabei von dem Bereich der Komponente ab. Diejenigen Komponenten, die direkten Zugriff auf den Zwischencode benötigen, also z. B. die Modifikation der Aufrufstellen, sind in C++ geschrieben, da dies die Sprache ist, in der auch LLVM geschrieben ist und somit für die Verwendung dessen Programmierschnittstelle notwendig ist. Diejenigen Komponenten, die abstrakt auf dem Betriebssystemmodell und den Instanz-

und Interaktionsgraphen arbeiten, sowie die Generierung der neuen Betriebssystemkomponenten ist aus Gründen der Flexibilität in Python implementiert. Insbesondere die Einbindung neuer Komponenten in das Betriebssystemmodell und die Generierung initialisierter Datenstrukturen lassen sich gut lesbar und flexibel in Python einarbeiten.

Als Evaluationsobjekte habe ich sowohl Mikrobenchmarks geschrieben, die auf die Messung der Effekte der Spezialisierung der Instanziierung zugeschnitten sind, als auch zwei reale Projekte ausgewählt, die die Anwendbarkeit auf reguläre Anwendungen demonstrieren sollen. Alle diese Anwendungen teilen sich die gemeinsame Ausführungsplattform auf Basis eines STM32F103 Mikrocontrollers (ARM© Cortex©-M3 @72MHz, 128KB Flash, 20KB SRAM) auf einer STM32 Nucleo-F103RB Entwicklungsplatine.

Zwecks der Evaluation der verschiedenen Spezialisierungstiefen bestehen die Evaluationen aus drei Sätzen, bei denen jeweils die maximale Spezialisierungstiefe beschränkt ist. Die spezialisierten Varianten habe ich jeweils mit und ohne Lauflängencodierung untersucht, um den daraus entstehenden Nutzen zu bemessen.

unverändert Es werden ausschließlich die Analysen ausgeführt, sodass Informationen über die Anwendung zur Verfügung stehen. Der Anwendungscode wird jedoch nicht verändert und auch das Betriebssystem wird ohne Veränderungen an die Anwendung gebunden. Dieses entspricht einer regulären Übersetzung von Anwendung und Betriebssystem ohne Eingriffe und dient als Referenz für den Vergleich der anderen Varianten.

partiell Die Ergebnisse von SIA und INA werden dazu verwendet, die bekannten Systemobjekte statisch zu allozieren. Auch wenn anhand der ermittelten Informationen eine tiefere Spezialisierung möglich wäre, wird in dieser Variante ausschließlich die Speicherallokation statisch ausgeführt. Sämtliche Initialisierung der Systemobjekte geschieht weiterhin während der Laufzeit.

vollständig In diesem Szenario gibt es keine Beschränkung der maximalen Spezialisierungstiefe. Alle möglichen Spezialisierungen werden angewendet.

Der Startvorgang des Systems von der ersten Instruktion des Prozessors bis hin zum Erreichen des SSPs besteht aus vier Phasen: (1) kopieren des initialen Datensegments aus dem Programmabbild in den Arbeitsspeicher, (2) initialisieren des BSS-Segments mit Nullen, (3) initialisieren des Segments mit dünn besetzten Daten (RLE decodieren), (4) ausführen des Startcodes der Anwendung beginnend bei der `main()` bis zum anwendungsspezifischen Erreichen des SSPs. Die Dauer dieser Phasen wird mittels der in den Prozessor integrierten *Data Watchpoint and Trace (DWT)* Einheit durchgeführt. Diese enthält einen Zyklenzähler, der in den ersten zwei vom Prozessor nach dem Start ausgeführten Instruktionen aktiviert wird. Am Ende jeder Phase nehme ich einen Zeitstempel dieses Zyklenzählers (Dauer: 10 Zyklen). Jedes Experiment habe ich 100 Wiederholungen durchlaufen lassen, die Standardabweichung lag dabei jeweils unter 30 Zyklen, sodass diese nicht in den Grafiken erkennbar ist. Aufgrund des sehr deterministisch ablaufenden Systemstarts ist auch keine Schwankung zu erwarten gewesen, da keine externen Einflüsse den Systemstart während dieser Experimente beeinflussen.

Neben der Analyse der Laufzeit habe ich auch den Speicherbedarf untersucht. Die verschiedenen Bereiche wie BSS, Daten, RLE und Programmcode sind dabei ebenso aufgeschlüsselt wie die jeweilige Größe im Arbeitsspeicher und im Abbild, aus dem dieser geladen wird.

4.4.1 Mikrobenchmarks

Für die Mikrobenchmarks habe ich zwei exemplarisch Vertreter der verschiedenen Kategorien von Systemobjekten ausgewählt: Für die Kategorie der Synchronisations- und Kommunikationsobjekte wird stellvertretend die in FreeRTOS als Queue bezeichnete Warteschlange evaluiert. Diese eignet sich besonders, weil die Implementierung eines Semaphors auf dieser beruht und somit mehrere Konzepte abgedeckt sind. Aus der Kategorie der Aktivitätsträger wird der Faden gewählt. Dessen Instanziierung wird dabei sowohl vor als auch nach dem Start des Mehraufgabenbetriebs evaluiert, da die Initialisierung und insbesondere die Einreihung in die Bereitlisten der Aufgabenplanung sich deutlich unterscheiden. Die Anwendungen bestehen dabei ausschließlich aus einer Aneinanderreihung von Objektinstanziierungen gleichen Typs, sodass ein störender Einfluss, verursacht durch sonstigen Anwendungscode, weitestgehend ausgeschlossen werden kann.

4.4.1.1 Warteschlangen-Instanziierung

Im Rahmen dieses Benchmarks werden 100 Warteschlangen (FreeRTOS Queues) erzeugt. Wie in Quellcode 4.8 ausschnittweise dargestellt, stehen die entsprechenden Systemaufrufe sequenziell im Programmcode (ohne Schleife). Dies entspricht am ehesten diversen Stellen im Programmfluss, die jeweils eine Warteschlange erzeugen, wenn man sämtlichen dazwischenliegenden Anwendungscode

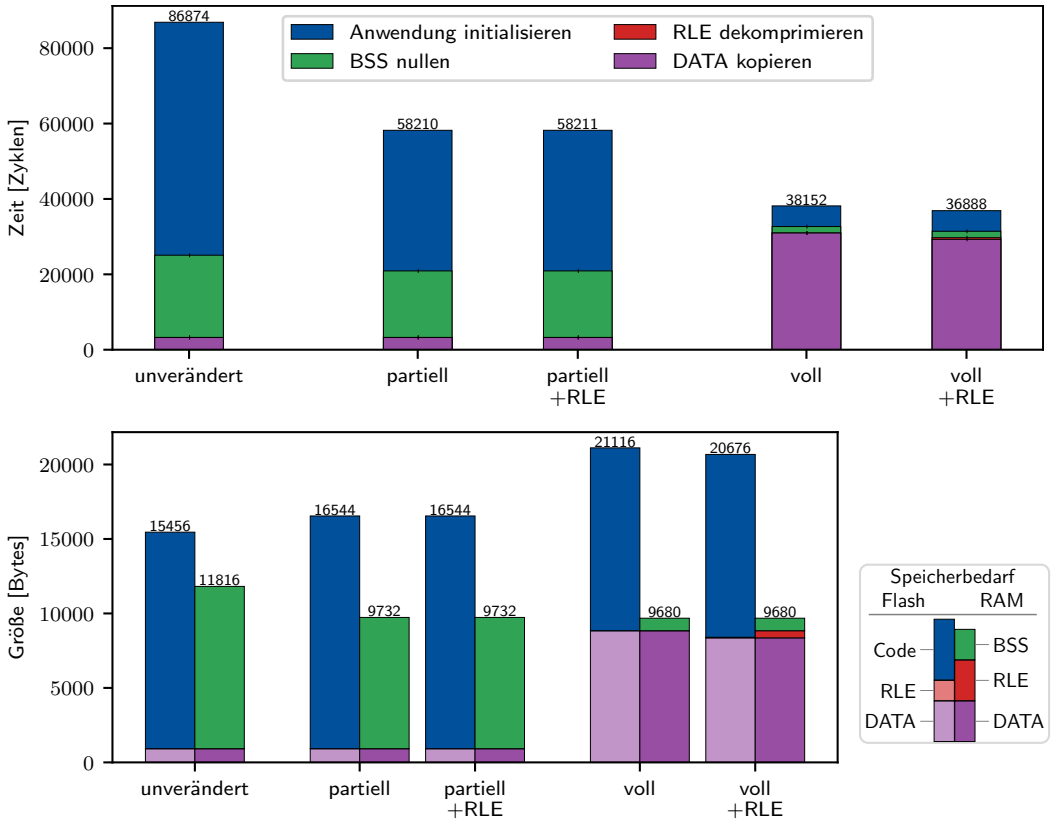

Abbildung 4.2 – Ergebnisse der spezialisierten Instanziierung von Warteschlangen. Zu sehen ist, dass mit steigender maximal erlaubter Spezialisierungstiefe die Laufzeit sinkt, da die Initialisierung in die Übersetzungszeit verlagert ist. Der Speicherbedarf steigt leicht, da die initialisierten Datenstrukturen geladen werden müssen.

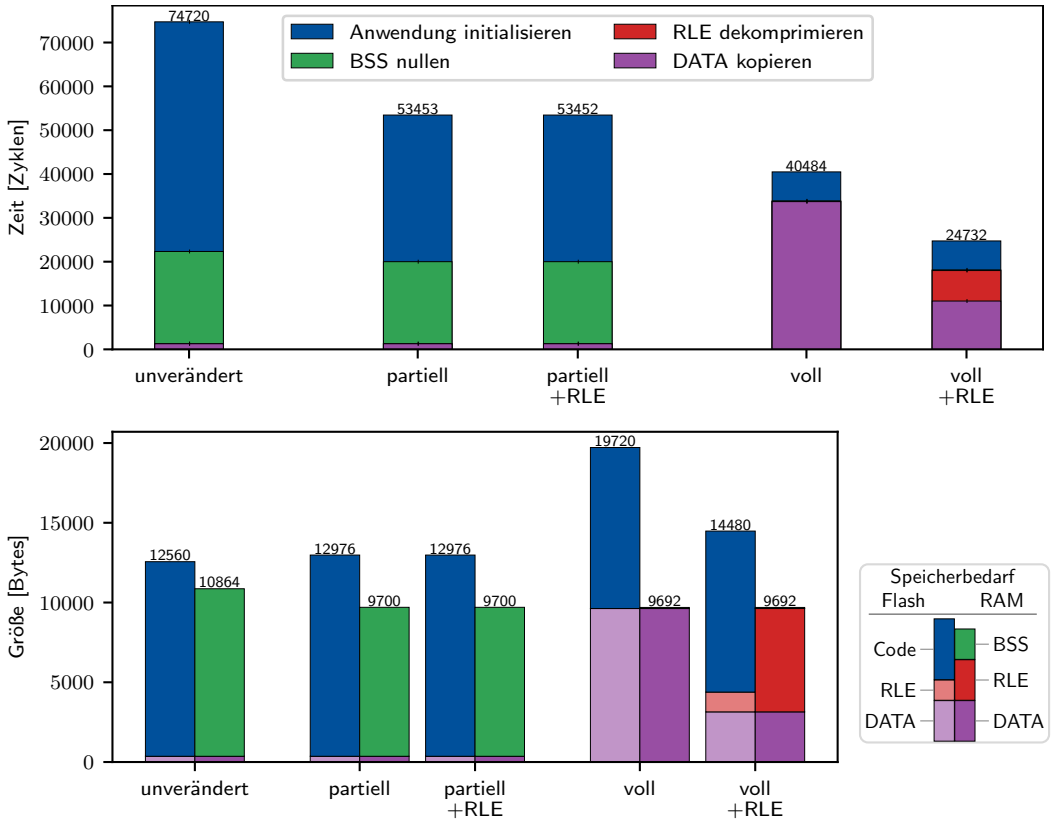

Abbildung 4.3 – Ergebnisse der spezialisierten Fadenerzeugung *vor* dem Start des Planers. Zu sehen ist, dass mit steigender maximal erlaubter Spezialisierungstiefe die Laufzeit sinkt, da die Initialisierung in die Übersetzungszeit verlagert ist. Der Speicherbedarf steigt leicht, da die initialisierten Datenstrukturen geladen werden müssen.

Quellcode 4.8 Programmcode des Benchmarks zur Instanziierung von Warteschlangen in FreeRTOS

```
1  int main() {
2    STORE_TIME_MARKER(main_start);
3    InitBoard();
4    STORE_TIME_MARKER(done_InitBoard);
5    kout.init();
6
7    STORE_TIME_MARKER(begin_queueCreate);
8    queue001 = xQueueCreate(4, sizeof(char));
9    queue002 = xQueueCreate(4, sizeof(char));
10   queue003 = xQueueCreate(4, sizeof(char));
11   /* ... */
12   queue099 = xQueueCreate(4, sizeof(char));
13   queue100 = xQueueCreate(4, sizeof(char));
14   STORE_TIME_MARKER(done_queueCreate);
15
16   vTaskStartScheduler();
17 }
```

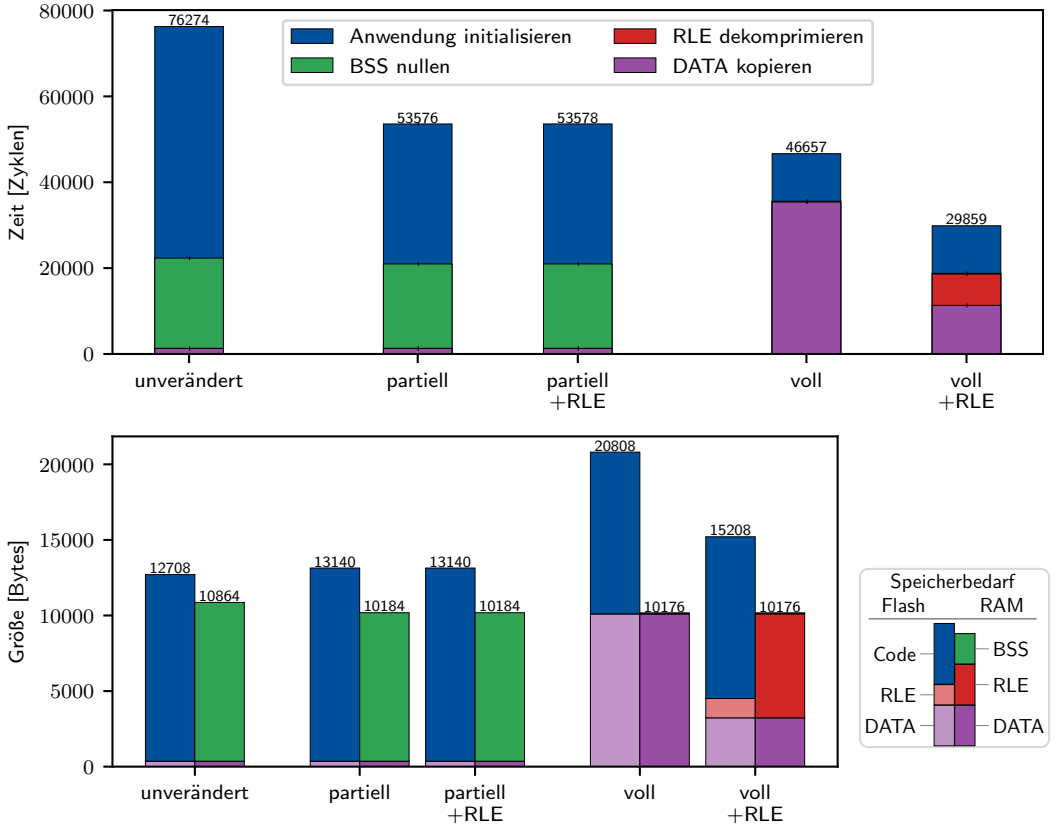

Abbildung 4.4 – Ergebnisse der spezialisierten Fadenerzeugung *nach* dem Start des Planers. Zu sehen ist, dass mit steigender maximal erlaubter Spezialisierungstiefe die Laufzeit sinkt, da die Initialisierung in die Übersetzungszeit verlagert ist. Der Speicherbedarf steigt leicht, da die initialisierten Datenstrukturen geladen werden müssen.

entfernt. Die Wahl der Parameter für Größe und Anzahl der Einträge hat dabei keinen Einfluss auf die Initialisierungsgeschwindigkeit, da nur Zeiger auf den Datenbereich initialisiert werden, der eigentliche Datenbereich aber erst bei Verwendung der Warteschlange beschrieben wird. Der SSP ist für diesen Benchmark auf den Zeitpunkt des Eintritts in den Mehraufgabenbetrieb (Zeile 16) festgelegt, da dieser den Wechsel vom Startvorgang in den regulären Arbeitsbetrieb repräsentiert.

Abbildung 4.2 zeigt die Ergebnisse dieses Benchmarks. Es sind dabei sowohl die Laufzeit der verschiedenen Phasen des Systemstarts als auch der Speicherbedarf der unterschiedlichen Segmente im Flashspeicher und im Arbeitsspeicher dargestellt. Es ist zu erkennen, dass jede Steigerung der Spezialisierungstiefe eine Verringerung der Startzeit hervorruft. Die partielle Spezialisierung in Form der statischen Speicherallokation führt bereits zu einer Reduktion der Startzeit um 28 663 Zyklen (33 %). Die vollständige Spezialisierung maximal erzielbarer Tiefe mit Lauflängencodierung erreicht den SSP bereits 49 986 Zyklen (58 %).

Ein Blick auf den Speicherverbrauch zeigt jedoch, dass die Geschwindigkeitsgewinne im Zusammenhang mit leicht gestiegenem Speicherbedarf im Textsegment stehen. Die Ursache dafür ist, dass nun jeder instanziierende Systemaufruf zusätzlich die Adresse des bereits allozierten Speichers als Argument erhält. Dies resultiert in zusätzlichen 11 Bytes pro Systemaufruf für das Laden der Parameter. In der Variante mit tiefstmöglicher Spezialisierung entfällt dieser Mehraufwand wieder, aber gleichzeitig ist eine Verschiebung vom BSS in das Datensegment zu beobachten. Dieser resultiert

aus den nun initialisierten Datenstrukturen. Damit wächst die Größe des Speicherbedarfs für den Flashspeicher um 52 Bytes pro Warteschlange. Die Lauflängencodierung hat hierbei keinen großen Einfluss, da die Verwaltungsstrukturen der Warteschlangen nicht dünn besetzt sind. Einzig der Leerlauffaden profitiert an dieser Stelle.

4.4.1.2 Faden-Instanziierung vor der Übergabe der Kontrolle an das Betriebssystem

Dieser Benchmark besteht aus einer sequenziellen Erzeugung von 30 Fäden mittels `xTaskCreate()`, die ebenso wie bei dem vorherigen Benchmark ohne Schleife explizit im Programmcode stehen. Jedem Faden wird dabei ein Laufzeitstapel mit 200 Bytes zugeordnet. Da die Parameter der Systemaufrufe alle als Konstanten gegeben sind, sind alle Aufrufe vollständig spezialisierbar. Die Position der Systemaufrufe ist dabei vor der Übergabe der Kontrolle an das Betriebssystem, sodass auch die Registrierung im Betriebssystem statisch durchführbar ist.

Auch bei diesem Benchmark ist zu erkennen, dass die Spezialisierung eine deutliche Verbesserung der Startzeit hervorruft. Abbildung 4.3 zeigt die entsprechenden Ergebnisse. Schon die partiell spezialisierte Variante erreicht den SSP 21 267 Zyklen (28 %) schneller als die generische Variante. Die vollständige Spezialisierung schafft dies bereits nach 34 236 Zyklen (46 %).

Wie auch für die Warteschlangen ist zu erkennen, dass die Spezialisierung mit einem vergrößerten Flash-Speicherabbild einhergeht. Die vollständig spezialisierte Variante benötigt 239 Bytes mehr pro Faden. Diesem Umstand wirkt die Lauflängencodierung jedoch entgegen, sodass bei entsprechender Codierung den Mehraufwand auf 64 Bytes pro Faden reduziert und zeitgleich die Startzeit im Vergleich zu der voll spezialisierten Variante ohne Lauflängencodierung um 15 752 Zyklen (39 %) verbessert wird.

Insgesamt bedeutet dies, dass die Anwendung aller beschriebenen Spezialisierungen in diesem Szenario zu einer Reduktion der Startzeit um 49 988 Zyklen (67 %) ermöglicht. Die Startzeit des Systems wurde somit auf ein Drittel der ursprünglichen Startzeit verkürzt.

4.4.1.3 Faden-Instanziierung nach dem Eintritt in den Mehraufgabenbetrieb

Dieser letzte Benchmark besteht aus einem Faden, der vor Übergabe der Kontrolle an das Betriebssystem instanziiert wird und als sogenannter Einrichtungsfaden fortan für den weiteren Start der Anwendung verantwortlich ist. Der Programmcode dieses weiteren Fadens besteht aus demselben Muster, wie es für den vorherigen Benchmark bereits verwendet wurde. Auch hier werden 30 Fäden mit jeweils 200 Byte Laufzeitstapel instanziiert. Um die Messung nicht durch die Einlastung anderer Fäden zu verfälschen, läuft dieser Einrichtungsfaden mit einer Priorität, die höher ist als die der noch zu erzeugenden.

Abbildung 4.4 zeigt die Ergebnisse dieses Benchmarks, welche auch wie die des vorherigen deutliche Laufzeitgewinne aufweisen. Die partiell spezialisierte Variante erreicht den SSP bereits 22 697 Zyklen (30 %) vor der generischen Variante. Die vollständig spezialisierte Variante erreicht mit Lauflängencodierung eine Reduktion der Startzeit um insgesamt 46 415 Zyklen (61 %). Die Reduktion ist an dieser Stelle geringer als bei dem vorherigen Benchmark, da die Registrierung im Betriebssystem nun zur Laufzeit ausgeführt werden muss.

Wie auch bei den vorherigen Benchmarks ergibt sich eine leichte Vergrößerung des Speicherabbildes, die mit 81 Bytes etwas größer ausfällt als im Vorherigen Benchmark, da der Aufruf des Systemaufrufs zum Einreihen in die Bereitliste nicht vermieden werden kann. Die Verwendung der Lauflängencodierung zeigt auch in diesem Benchmark, dass eine den Daten angemessene Ausdrucksmöglichkeit einen entscheidenden Einfluss auf die Laufzeit eines Systems hat.

4.4.2 Fallbeispiele mit Realweltanwendungen

Nachdem die Spezialisierungen in den Mikrobenchmarks ihren Nutzen demonstrieren konnten, habe ich dieselben Spezialisierungen auch auf zwei echte Anwendungen eingebetteter Systeme angewendet. Diese beiden Fallbeispiele demonstrieren die Anwendbarkeit des Ansatzes auf reale Anwendungen, zeigen aber gleichzeitig auch die Stellen auf, an denen weitere Verbesserungen möglich sind.

4.4.2.1 LibrePilot CopterControl

Die CopterControl-Firmware aus dem LibrePilot-Projekt ist die Fluglageregelungssoftware eines Quadrokopters. Diese verwendet ebenfalls FreeRTOS als Betriebssystem und kann auf einem ARM Cortex M3 Mikroprozessor ausgeführt werden. Dieser ist im realen Einsatz mit externer Peripherie zur Erfassung und Steuerung der Sensoren und Aktoren verbunden. Diese wurden für diese Evaluation

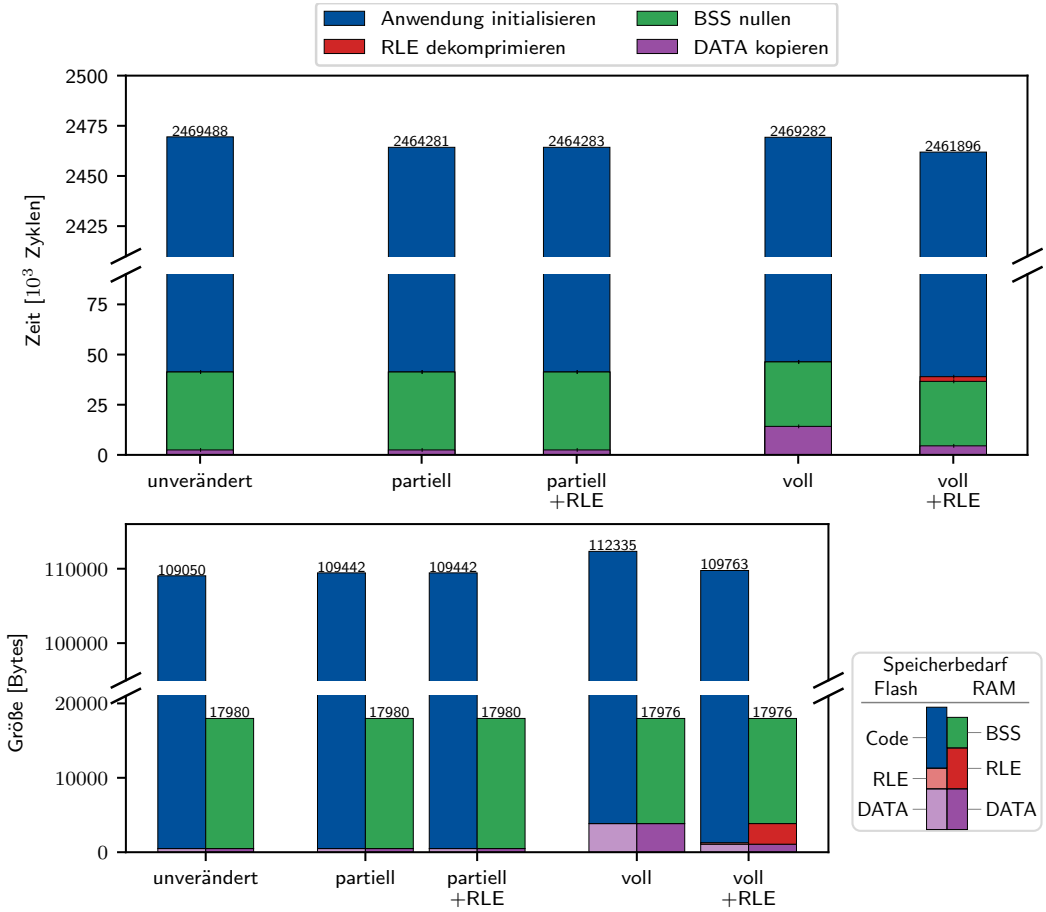

Abbildung 4.5 – LibrePilot CopterControl: Ergebnisse der Systemstart-Spezialisierung. Die Laufzeit bis zum Erreichen des Systemstarts kann durch die Spezialisierung reduziert werden, dies ist jedoch im Verhältnis zu der Gesamtstartzeit relativ gering. Der Speicherbedarf steigt leicht, da der zusätzliche Speicherbedarf für statische Systemobjekte nicht durch den Wegfall der dynamischen Betriebssystemfunktionalitäten ausgeglichen werden kann.

jedoch durch entsprechende Attrappen ausgetauscht, um den Einfluss auf den Startvorgang durch externe Faktoren zu verringern.

Da es in der Anwendung bisher keinen als solchen ausgezeichneten SSP gab, habe ich diesen händisch auf den Zeitpunkt nach der letzten Instanziierung gesetzt. Die Anwendung verfolgt dabei das Muster des Initialisierungstasks, welcher alle weiteren Fäden erzeugt. Dieses Verhalten entspricht dem des dritten Mikrobenchmarks. Verwoben mit der Instanziierung von Systemobjekten ist auch die Initialisierung der restlichen Anwendung.

Die Analyse dieser Anwendung ergibt 2673 Funktionen bestehend aus 17 265 Basisblöcken, die aus 78 787 Zeilen Quellcode übersetzt entstammen. Der Aufrufgraph verzeichnet 2882 normale und 223 Systemaufrufstellen. Von diesen sind 33 Rufe zu instanziierenden Systemaufrufen und dementsprechend potenzielle Spezialisierungskandidaten. Die maximale Länge eines Aufrufpfades zu einer solchen Aufrufstelle beträgt acht Aufrufe. Als Werkzeug zur flexiblen und modularen Implementierung verwendet das LibrePilot Projekt ein speziell angepasstes Skript mit Anweisungen für den Binder, welches die Initialisierungsroutinen der einzelnen Komponenten in einer Sektion von Funktionszeigern zusammensammelt, sodass diese zur Startzeit sequenziell gerufen werden können. Zu diesem Zweck habe ich ARA um die die Fähigkeit der Interpretation dieser speziellen Sektion und deren Aufrufstelle erweitert.

Ergebnis der SIA ist, dass ARA die Instanziierung von 17 Fäden und 24 Warteschlangen erkennt. Dabei fällt auf, dass es sich um mehr Aufrufe als Aufrufstellen handelt. Dies ist darin begründet, dass einige Aufrufstellen mehrfach über unterschiedliche Aufrufpfade erreicht werden können. Von diesen Instanziierungen sind 5 Fäden vollständig und 2 partiell spezialisierbar. Keine der Warteschlangen ist spezialisierbar, weil alle Aufrufstellen als nur bedingt erreichbar klassifiziert werden. Ursache dafür ist, dass diese nach Fehlerbehandlungscode stehen, der zuvor die erfolgreiche Instanziierung von Fäden und Einrichtung anderer Anwendungskomponenten abprüft. Auch wenn es sich hierbei um Programmcode zur Ausnahmebehandlung handelt, so ist dieser Umstand für die statische Analyse nicht ersichtlich, sodass nicht von einer sicheren Instanziierung ausgegangen werden kann und damit die Grundlage für die statische Spezialisierung fehlt. Diese Unsicherheit im Aufrufpfad wird zu allen folgenden Funktionsrufen im Aufrufgraph propagiert, sodass keine weitere Instanziierung in folgenden Knoten als spezialisierbar eingestuft werden kann. In Kapitel 6 gehe ich weiter darauf ein, wie mit solcher Idiomatik, die regulären Programmfluss und Ausnahmebehandlung kennzeichnen, umgegangen werden kann. Für diese Fallstudie verbleiben die Systemobjekte jedoch ohne Spezialisierung.

Nach Anwendung der möglichen Spezialisierung und entsprechender Lauflängencodierung wird die Startzeit des Systems um 7 592 Zyklen verringert. In Abbildung 4.5 ist die genauen Aufteilungen in die verschiedenen Phasen des Systemstarts dargestellt. Aus dieser Darstellung ist erkennbar, dass es sich bei dieser Reduktion nur um einen kleinen Bruchteil der Gesamtlaufzeit handelt. Dies ist darin begründet, dass die allermeiste Zeit in der Ausführung von sonstigem Anwendungscode verbracht wird und dementsprechend die von der Spezialisierung hervorgerufene Reduktion mit 0.31 % der Laufzeit nur minimalen Einfluss hat.

Gleichzeitig ergab eine manuelle Analyse des Programmcodes, dass ein großer Teil der Startzeit mit der Initialisierung von Anwendungsobjekten verbracht wird, die ebenfalls zu Beginn der Laufzeit einmalig instanziiert werden und für den Rest der Laufzeit existieren. Ihre Initialwerte variieren dabei regelhaft nicht zwischen Startvorgängen, sondern sind statisch in Abhängigkeit der eingesetzten Komponenten des Quadrokopters. Schlimmer noch können diese Anwendungsobjekte in der vorliegenden Konfiguration nicht wieder aus dem System entfernt werden, da die Freispeicherverwaltung keine Unterstützung dazu bietet. Sie besteht nur aus einer Implementierung der Allokationsoperation mittels eines monoton inkrementierten Zeigers. Eine Freigabe von Speicher ist bei dieser Implementierung weder vorgesehen noch möglich. An dieser Stelle existiert also das

Problem der eigentlich statischen Objekte, die dennoch erst dynamisch instanziiert werden auf Ebene der Anwendung erneut.

4.4.2.2 GPSLogger

Der GPSLogger ist eine frei zugängliche Implementierung eines tragbaren Geräts zur Erfassung, Visualisierung und Speicherung von Positionsdaten mittels satellitengestützter globaler Positionsbestimmung (GPS). Das System besteht aus dem Mikroprozessor, verbunden mit einer Anzeige (I^2C), einem GPS-Empfänger (UART), einer SD-Speicherkarte (SPI) und zwei Tastern (GPIO). Der Systemstart verläuft dabei bereits im Programmcode zweigeteilt, sodass alle Systemobjekte vor Eintritt in den Mehraufgabenbetrieb instanziiert werden und die Initialisierung der Peripherie von den jeweilig zuständigen Fäden ausgeführt wird. Dementsprechend habe ich den SSP auf den Einstieg in den Mehraufgabenbetrieb festgesetzt. Die gesamte Anwendung besteht aus 1311 Funktionen mit 11 165 Basisblöcken, die aus 79 573 Zeilen Programmcode übersetzt werden. Der Aufrufgraph enthält insgesamt 2303 Aufrufstellen, darunter 36 Aufrufstellen von Systemaufrufen. Von diesen sind 10 Aufrufstellen Rufe von instanziierenden Systemaufrufen und damit Spezialisierungskandidaten. Die maximale Tiefe eines Aufrufpfades einer dieser Aufrufstellen beträgt drei.

Die aus der SIA resultierende Instanzliste (Abbildung 4.6) enthält sechs Fäden (fünf reguläre und der Leerlauffaden), zwei Warteschlangen und zwei Semaphore. Wie in dort angegeben, sind alle Eigenschaften der Instanzen bekannt und die Anforderungen der Spezialisierbarkeit erfüllt, sodass eine statische Instanziierung bei allen vollständig vorgenommen werden kann. Wie in Abbildung 4.7 dargestellt, resultiert bereits eine partielle Spezialisierung in einer Reduktion der Startzeit um 6 790 Zyklen (12 %) in einer deutlichen Reduktion der Startzeit. Wie auch bereits in den Mikrobenchmarks zu beobachten war, führt die statische Instanziierung zuerst zu einer Vergrößerung des Datensegments, woraus eine Verlängerung der Startzeit um 880 Zyklen (2 %) resultiert. Der

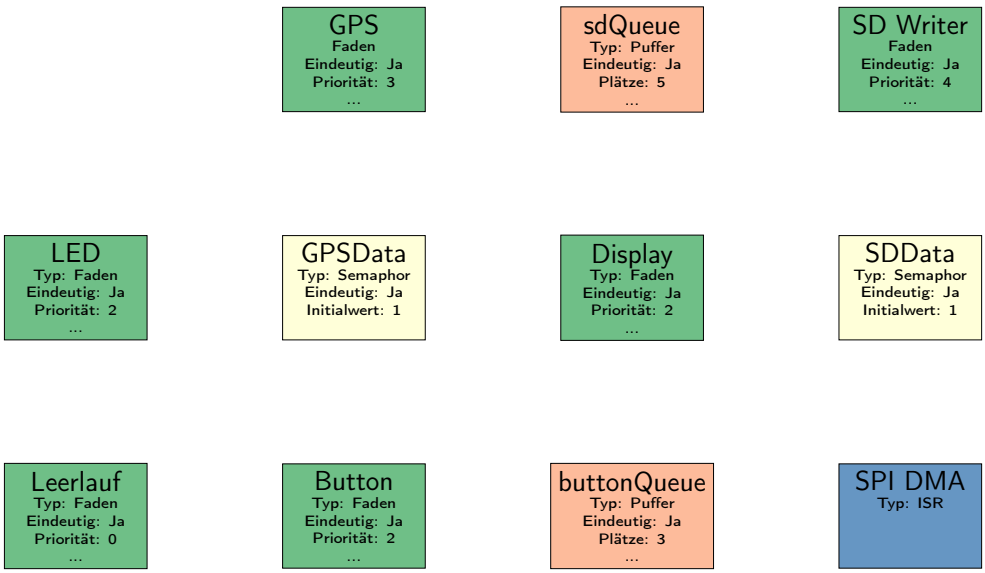

Abbildung 4.6 – Instanzliste der von ARA im GPSLogger detektierten Systemobjekte und ihrer Attribute. Sechs Fäden(grün), davon fünf regulär und der Leerlauffaden, zwei Warteschlangen(orange), zwei Semaphore(gelb) und eine ISR (blau) formen das System.

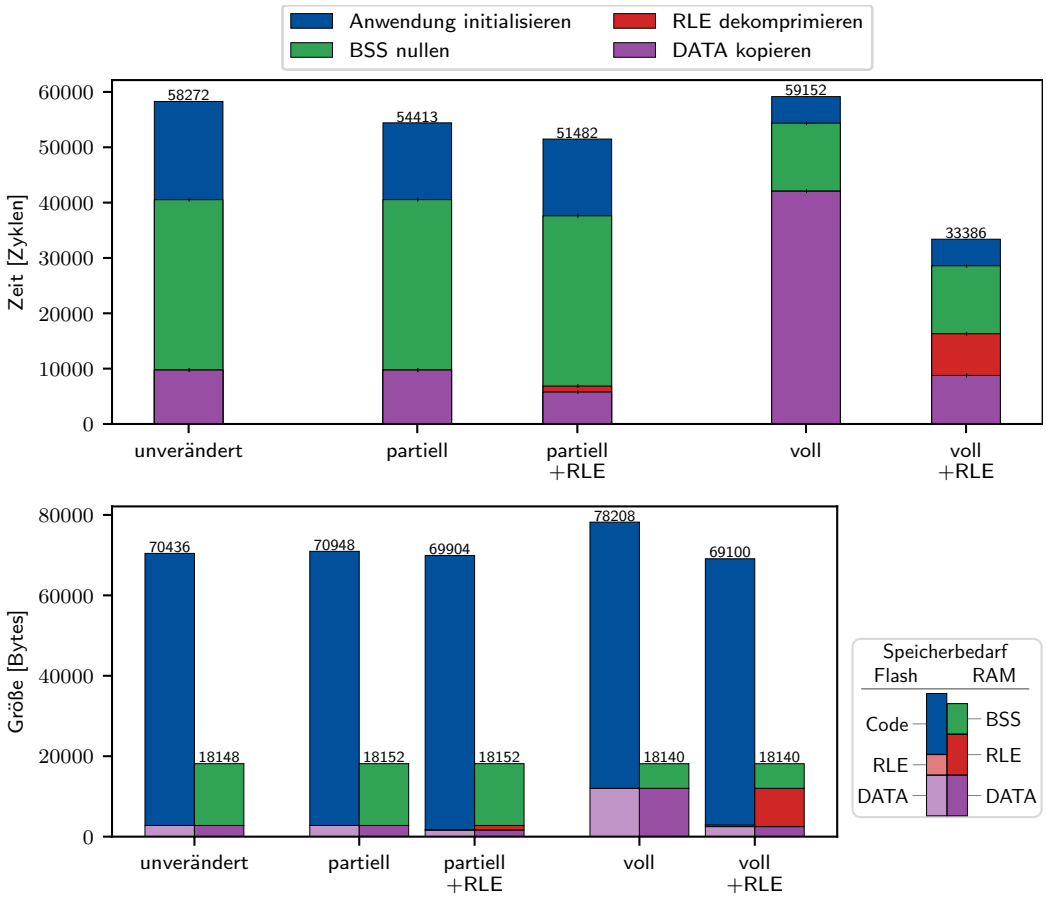

Abbildung 4.7 – GPSLogger: Ergebnisse der Systemstart-Spezialisierung. Sowohl Laufzeit als auch Speicherbedarf sind bei maximaler Ausnutzung der möglichen Spezialisierungen geringer als in der unveränderten Variante.

Einsatz der Lauflängencodierung gleicht diesen Nachteil wieder aus, sodass insgesamt die Startzeit um 24 886 Zyklen (43 %) reduziert wird.

Die Betrachtung des Speicherbedarfs zeigt, dass die Größe des Abbildes der vollständig spezialisierte Variante mit Lauflängencodierung im Flashspeicher insgesamt 1 336 Bytes (2 %) sinkt. Dies resultiert aus zwei gegenseitig laufenden Effekten: Zum einen führt die statische Instanziierung zu einem vergrößerten Datensegment, die Lauflängencodierung hält dies jedoch insbesondere für die Laufzeitstapel ausreichend niedrig. Insgesamt wird so 124 Bytes für diese Systemobjekte benötigt. Die vollständige Spezialisierbarkeit aller Systemobjekte führt jedoch gleichzeitig dazu, dass die gesamte dynamische Speicherverwaltung nicht mehr benötigt wird. Somit entfällt sämtlicher dazugehöriger Programmtext aus dem Textsegment und resultiert in einer Reduktion um 1 460 Bytes. Zusätzlich bedeutet der Wegfall der Speicherverwaltung, dass der Gesamtspeicherbedarf sicher zur Übersetzungszeit bestimmbar ist und somit kein überdimensionierter Speicherbereich vorgehalten werden muss, um eventuell variable Anforderungen zur Laufzeit erfüllen zu können. Dies resultiert in einer zusätzlichen Planungssicherheit für die Dimensionierung des benötigten Arbeitsspeichers.

Dieses Fallbeispiel demonstriert neben der Anwendbarkeit der statischen Spezialisierung auch, dass die notwendigen Ausdrucksmittel für die Deklaration von Speicherbereichen und Konstruktoren auch in anderen Bereichen der Anwendung einen Gewinn bringen. So ist der Startzeitgewinn

zwischen den partiell spezialisierten Varianten des GPSLoggers mit und ohne Lauflängencodierung darauf zurückzuführen, dass auch Objekte der Anwendung von der Lauflängencodierung profitieren. Konkret handelt es sich hierbei um einen Pufferspeicher der verwendeten Dateisystembibliothek, die bei der Verwendung der Speicherkarte verwendet wird. Dieser Pufferbereich besteht vorwiegend aus Nullen, die später mit gelesenen oder zu schreibenden Daten eines Blocks aus dem Dateisystem gefüllt werden sollen, attribuiert mit dazugehörigen konstant initialisierten Verwaltungsinformationen (1136 Bytes). Dementsprechend sorgt auch bei solchen Datenstrukturen die Lauflängencodierung für einen Laufzeitvorteil von hier 2 931 Zyklen (5 %). Ursache ist auch hier wieder, dass Optimierungsalgorithmen des Übersetzers ermitteln, dass alle Werte des Konstruktoraufrufes und somit alle daraus resultierenden Schreibzugriffe statisch berechenbar sind und somit dieser Konstruktoraufruf in eine Initialisierungsliste für den entsprechenden Speicherbereich umgewandelt wird.

4.5 Zusammenfassung

Der Systemstart ist das Fundament für das Erreichen eines arbeitsfähigen Zustandes und je nach Fehlerbehebungsstrategie auch zwangsläufig Teil der kritischen Pfade während der Laufzeit. Dynamisch konfigurierte Systemschnittstellen geben zwar mehr Flexibilität, es bedarf aber zusätzlichem Laufzeitaufwand dieser Flexibilität gerecht zu werden. Dementsprechend scheint eine Vermeidung dieser ungenutzten Flexibilität als lohnenswerte Maßnahme. In diesem Kapitel haben wir dazu die Umformung von dynamisch konfigurierten Systemobjektinstanziierung in statische äquivalente eingeführt. Auf diesem Weg ist es möglich, all jenen Systemobjekte die Vorteile statischer Konfiguration zukommen zu lassen, ohne für die Flexibilität zahlen zu müssen. Gleichzeitig verbleibt die Freiheit, auch den vollen Umfang der möglichen Flexibilität auszunutzen, falls benötigt, ohne eine Interoperabilität zu verhindern.

Mit den vorgestellten Mikrobenchmarks und der Anwendung der Spezialisierungen auf Realweltprojekte konnte ich demonstrieren, dass dieser Ansatz funktioniert und auch in realistischen Szenarien lohnenswerte Verbesserungen erzielt. Insgesamt handelt es sich dabei in der Regel um eine Abwägung zwischen Speicherbedarf und Laufzeit. Für einige Anwendungen ist jedoch auch eine Verbesserung beider Eigenschaften möglich.

Zusätzlich konnte ich auch zeigen, dass das Fehlen von Ausdrucksmöglichkeit nicht nur bei der Schnittstelle zwischen Anwendung und Betriebssystem zu einem unnötigen Mehraufwand führt. Auch fehlt es an der Möglichkeit, anwendungsspezifische oder an Datenstrukturen angepasste Ladestrategien zu selektieren.

Forschungsfrage

Mit den hier vorgestellten Spezialisierungen sind für die ersten zwei Forschungsfragen Antworten in Bezug auf den Systemstart gefunden. Bezug nehmend auf die erste Forschungsfrage (FF1) kann ich nun sagen: Es ist möglich, die Vorteile statischer Konfiguration und Instanziierung auf eigentlich dynamisch instanziierte Systemobjekte anzuwenden, ohne dabei die Flexibilität, die das Betriebssystemmodell bietet, an den Stellen einzuschränken, an denen sie benötigt wird. Es sind keine negativen Auswirkungen auf nicht spezialisierte Systemobjekte beobachtbar. Insbesondere ergibt sich für die Anwendungsentwickler kein Nachteil, da sie weiterhin die dynamischen Instanziierungsaufrufe verwenden und diese transparent vollautomatisch umgeformt werden. Auch die Aktualisierbarkeit ist nicht grundsätzlich eingeschränkt, sondern benötigt lediglich die Beachtung der momentan verwendeten Systemversion, um den zu aktualisierenden Programmcode entsprechend zusammenzusetzen.

Für die zweite Forschungsfrage (FF2) ergibt sich folgenden Bild: Die statische Spezialisierung verringert die benötigte Laufzeit der Instanziierungen, sodass insgesamt der Startvorgang so spezialisierter Systeme beschleunigt wird um bis zu 67 %. Die Prognose des benötigten Speicherbedarfs ist

nicht ganz eindeutig, sondern hängt von der konkreten Anwendung ab. Es zeichnet sich jedoch das Bild, das bei einem höheren Grad der Spezialisierung tendenziell von einem reduzierten Speicherbedarf ausgegangen werden kann. Im besten Fall konnten Reduktionen des Flashspeicherbedarfs um 2 % erreicht werden. Im schlimmsten Fall resultiert der Speicherbedarf jedoch in einer Erhöhung von 64 Bytes pro statisch instanziiertem Systemobjekt.

Nachdem dieser Abschnitt der Lebenszeit einer Anwendung in Ausführung beleuchtet wurde, widme ich mich im folgenden Kapitel der Arbeitsphase der Anwendung und betrachte dort vorhandenes Spezialisierungspotenzial.

5

Interaktionen

Bei bekannten Interaktionen weniger interagieren

Das System ist gestartet und befindet sich nun im arbeitsfähigen Regelbetrieb. Die bisher getätigten Spezialisierungen haben nun keine weiter Auswirkung auf diesen Betriebsmodus, da sie die Systemobjekte exakt so instanziiert haben, wie die dynamischen Systemaufrufe dies auch getan hätten. In diesem Kapitel wollen wir uns daher der Frage widmen, ob nicht auch eine Verbesserung der nichtfunktionalen Eigenschaften für den regulären Arbeitsbetrieb anhand der gewonnenen Informationen möglich ist.

Dieses Kapitel nähert sich dazu schrittweise der Beantwortung der Fragestellung.

Zunächst betrachte ich in Abschnitt 5.1 in welchen Bereichen die generischen Implementierungen der dynamischen Systemobjekte Spezialisierungspotenzial aufweisen und stelle exemplarische Interaktionsspezialisierungen vor. In Abschnitt 5.2 liegt der Fokus auf der Auswahl spezialisierbarer Systemobjektinstanzen anhand ihrer Verwendung und der Spezialisierbarkeit ihrer Verwendungsstellen. Anschließend betrachtet Abschnitt 5.3 die Anwendung der vorgestellten Interaktionsspezialisierungen auf reale Anwendungen und evaluiert die Anwendbarkeit und das Kosten-Nutzen-Verhältnis.

5.1 Spezialisierungspotenzial und nicht genutzte Flexibilität generischer dynamischer Systemobjekte

Ein Kommunikationsobjekt (z. B. eine Nachrichtenwarteschlange), welches ohne Einschränkungen in einem dynamisch konfigurierten System verwendbar sein soll, muss alle möglichen Einsatzszenarien abdecken können. Dies bedeutet, dass die Implementierung keine Optimierungen hinsichtlich des später angewendeten Verwendungsmusters enthalten kann. Dies führt zu pessimistischen Annahmen und daraus resultierenden übervorsichtigen Schutzmaßnahmen, um die korrekte Funktionsweise in jedem Fall garantieren zu können. Dies bedeutet beispielsweise, dass zum Schutz von Verwaltungsdaten eine Unterbrechungssperre verwendet wird, die zu grobgranular jede Unterbrechung unterbindet, anstatt nur diejenigen, die tatsächlich im Konflikt mit der momentan anstehenden Änderung stehen. Dies führt unter Umständen zu einer Prioritätsumkehr, da eine niederpriore Aufgabe während der Unterbrechung abgearbeitet wird, obwohl eine höherpriore Aufgabe auf Abarbeitung wartet. Daraus entstehen nicht nur vergrößerte maximale Ausführungszeiten, sondern auch eine zwangsläufig schlechtere Analysierbarkeit dieser und dementsprechend pessimistischere Abschätzung. Die Spezialisierung von Interaktionen statisch konfigurierter Echtzeitsysteme konnte bereits zeigen, dass sowohl die eigentliche Ausführungszeit [Hof+09; DL17] als auch deren Analysierbarkeit [Die+17; Die19] deutlich verbessert werden.

Das Optimierungspotenzial ergibt sich aus der wegfallenden Notwendigkeit, beliebige Zugriffsmuster zu gestatten. Dies resultiert zum einen in einer weniger pessimistischen Unterbrechungssperre und zum anderen aus geringerem Aufwand für Verwaltungsinformationen. Am Beispiel einer Nachrichtenwarteschlange ist dies gut demonstrierbar: Die Implementierung einer Warteschlange, die zeitgleiche Zugriffe aus beliebigen Kontrollflüssen koordiniert abarbeiten soll, ist gezwungen, zu diesem Zweck global die Unterbrechungen zu sperren, um die Konsistenz der internen Datenstrukturen sicherzustellen. Dieses Verhalten führt in jedem Fall zu einem funktional richtigen Ergebnis, hat aber Nebenwirkungen auf die eigentlich nicht an dem Zugriff beteiligten Komponenten des Systems und auf die allgemeine Analysierbarkeit des Systems. Die globale Unterbrechungssperre führt, wie wir in Abschnitt 2.3.3 gesehen haben, zu Prioritätsumkehr, Verschlechterung der Analysierbarkeit von Antwortzeiten und damit pessimistischeren Analyseergebnissen und zu Beeinflussung von eigentlich unabhängigen Aktivitäten, da insgesamt die Planung und Einlastung von Aktivitätsträgern (Fäden und Unterbrechungshandhaber) ausgesetzt wird. Ist nun aber für eine gegebene Warteschlange die Menge der darauf zugreifenden Aktivitätsträger bekannt, so kann dieses Wissen eingesetzt werden, um die Beeinflussung der nicht beteiligten Komponenten des Systems zu verringern. Beispielsweise kann die Anhebung der Ausführungspriorität auf ein geringeres Niveau begrenzt oder eventuell sogar vollständig ausgelassen werden. Ebenfalls kann es möglich sein, weniger komplexe Zuteilungsverfahren zu verwenden und so die benötigte Rechenzeit und den Speicherbedarf für dazugehörige Verwaltungsdaten zu reduzieren. Konkret bedeutet dies, dass wenn es z. B. genau zwei zugreifende Fäden gibt, keine dynamisch wachsende verkette Liste zur Verwaltung der Wartenden benötigt wird, sondern nur ein einziges Bit, welches den Wartezustand des bekannten anderen Fadens repräsentiert.

Insbesondere bei Kommunikationsobjekten besteht die zweite große Komponente des Verwaltungsaufwandes neben der Synchronisation darin, zu verwalten, welche eventuell blockierten Handlungsstränge als Nächstes weiterlaufen sollen. Im einfachsten Fall wird allen signalisiert, dass dies eventuell möglich ist und die Konkurrenzsituation wird erneut aufgebaut. Größere Planungssicherheit im Sinne einer Echtzeitfähigkeit und einer Fortschrittsgarantie für alle bieten jedoch Verfahren, die die Freigabe geordnet erteilen. Das Erstellen und umsetzen einer solchen Ordnung bedarf ebenfalls Rechenzeit zum Abspeichern und Verwalten eben dieser Verwaltungsinformationen. Ist nun aber für ein gegebenes Kommunikationsobjekt bekannt wer wie interagiert, so ließe sich die

Zuteilungsverwaltung bestenfalls vollständig in die Übersetzungszeit verlagern und somit keine weiteren Laufzeitkosten verursachen. Zumindest aber kann eine Aussage darüber getroffen werden, wie viel der im generischen Fall notwendigen Koordination auch in einem gegebenen konkreten Fall notwendig ist.

5.1.1 Spezialisierungspotenzial von Synchronisations- und Kommunikationsobjekten in FreeRTOS

Quellcode 5.1 FreeRTOS Queue Implementierung mit globaler Unterbrechungssperre

```
1 BaseType_t xQueueGenericSend( QueueHandle_t xQueue, const void * const pvItemToQueue, ↘
      TickType_t xTicksToWait, const BaseType_t xCopyPosition )
2 {
3   BaseType_t xEntryTimeSet = pdFALSE, xYieldRequired;
4   TimeOut_t xTimeOut;
5
6   for( ;; )
7     {
8       taskENTER_CRITICAL();
```

FreeRTOS bietet als Objekte zur Synchronisation und Kommunikation mehrere Abstraktionen an (Queue, zählende und binärer Semaphore, Mutex, StreamBuffer). Die ersten sind jeweils so implementiert, dass sie grobgranular die Unterbrechungen sperren und die Aufgabenplanung pausieren während einer Interaktion. Dies führt sicher immer zu einem konsistenten Zustand und einfacher Implementierungslogik. Gleichzeitig resultiert daraus aber auch potenzielle Prioritätsumkehr, während hochpriore Unterbrechungen nicht aktiv werden können, da eine niederpriore Interaktion in Abarbeitung ist. Abbildung 5.1 zeigt den entsprechenden Ausschnitt aus dem Quellcode der Implementierung von FreeRTOS. Es ist zu erkennen, dass in Zeile 8 ein kritischer Abschnitt betreten wird. Dieser sperrt je nach verwendeter FreeRTOS-Konfiguration und Hardwareunterstützung mindestens die Aufgabenplanung, meist aber global alle Unterbrechungen durch Erhöhen der momentanen Ausführungspriorität. Diese Sperrung geschieht unabhängig von einer Notwendigkeitsprüfung und resultiert aus der pessimistischen Annahme, dass eine Sperre immer notwendig ist.

Unter der Annahme, dass im Regelfall diese grobgranulare Sperre nicht notwendig ist, weil das zu verwaltende Betriebsmittel nicht belegt ist, resultiert hier ein Spezialisierungspotenzial. In FreeRTOS ist die Implementierung der Queue das zentrale Grundgerüst, auf dem ebenfalls Mutex, zählende und binäre Semaphore aufbauen. Aus diesem Grund betrachte ich hier stellvertretend nur für die Queue mehrere spezialisierte Varianten zur Demonstration des resultierenden Spezialisierungsvorteils.

Zugriffe auf Warteschlangen im allgemeinen Fall bedürfen einer grobgranularen Unterbrechungssperre wie in der gegebenen Implementierung vorhanden. Ist aber aus der Interaktionsanalyse erkennbar, dass die Zugriffsmuster nicht beliebig sind, so gibt es verschiedene Verfahren, die Unterbrechungssperre feingranular oder mittels weicher Synchronisation ganz ohne Sperre zu gestalten. Lamport betrachtet bereits früh dieses Forschungsfeld und beschreibt einige der auftretenden Probleme wie gleichzeitiges Schreiben desselben Datums und die Notwendigkeit der Sequenzialisierung von Instruktionen durch die Hardware. Daraus resultiert die Notwendigkeit, die Lese- und Schreibreihenfolge mehrteiliger Daten einzuhalten, um konsistente Ergebnisse zu erzielen[Lam77b]. Allgemein ist es wichtig, dass die Korrektheit der verwendeten Algorithmen zum sperrfreien Zugriff gewährleistet ist. Herlihy und Wing stellt hierzu die Kriterien der Linearisierbarkeit als eben solche Korrektheitskriterien auf [HW90].

Verschiedene Arbeiten [MS96; Her+05; Doh+04b] beschreiben Optimierungen für weich synchronisierte Pufferstrukturen variierender Größe. Ebenso gibt es auch weich synchronisierte Ringpuffer [RT12; HPS02]. Dennoch ist umstritten, ob diese Implementierungen lohnenswert gegenüber globalen Sperren sind und ob der zu führende Mehraufwand gerechtfertigt ist [Doh+04a]. Ebenso ist die Freispeicherverwaltung von diesen Problemen betroffen, sodass auch dort derartige Probleme gelöst werden müssen [Her+05]. Die konkrete Implementierung der verwendeten Spezialisierung soll an dieser Stelle aber weniger das Thema sein als die automatische Selektion anhand der Verwendungsmuster und gebotenen Eigenschaften. Aus diesem Grund beschränke ich mich ausschließlich auf das Szenario mit genau einem Leser und einem Schreiber, für die bekannte und bewährte Lösungen existieren [Lam77a].

5.1.2 Laufzeitgewinne durch Einschränkung der zugelassenen Aktivitätsträger

Zur Demonstration des prinzipiellen Nutzens werden nun im Folgenden die reguläre Implementierung der Queue in FreeRTOS mit zwei Varianten der auf einen Leser und einen Schreiber spezialisierten Varianten verglichen. Diese unterscheiden sich in der Unterstützung für blockierendes Warten. FreeRTOS ermöglicht in der regulären Variante die Angabe einer Wartezeit, die der Systemaufruf blockiert, bis er fehlschlägt. Die beiden Extremwerte dieser Wartezeit sind unendliches Warten und kein Warten. Genau diese beiden Spezialfälle entsprechen den zwei unterstützten Varianten der spezialisierten Implementierungen.

Zusätzlich vergleiche ich die alternative Implementierung StreamBuffer, die in FreeRTOS vorhanden ist und ebenfalls keine Wartezeiten gestattet. Diese ist wie die spezialisierten Varianten als Ringpuffer implementiert, arbeitet jedoch nicht nachrichtenorientiert, sondern stromorientiert.

Für die Messungen habe ich dieselbe Versuchsanordnung verwendet, die bereits in Abschnitt 4.4 verwende. Jede Messung wurde in 100 Wiederholungen auf dem STM32F103 Mikrocontroller (ARM© Cortex©-M3 @72MHz, 128KB Flash, 20KB SRAM) ausgeführt und die Zeiten wurden jeweils mit prozessoreigenen Zählern ermittelt. Die die Plattform zeigte ein deterministisches Verhalten, sodass alle Wiederholungen dieselbe Laufzeit ergaben.

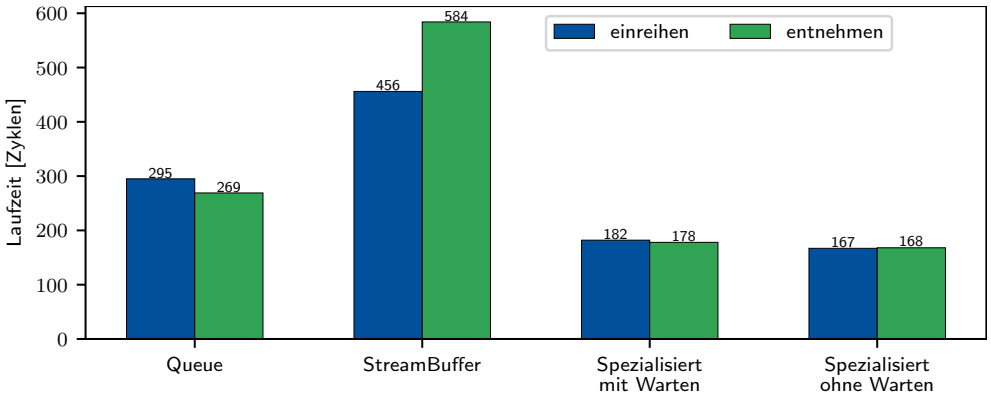

Abbildung 5.1 – Laufzeiten der verschiedenen Puffer-Implementierungen im nicht blockierenden Fall. Die spezialisierte Variante hat eine geringere Laufzeit als die generische Queue-Implementierung. Die StreamBuffer-Implementierung ist keine lohnenswerte Alternative, da sie erhöhte Laufzeit aufweist.

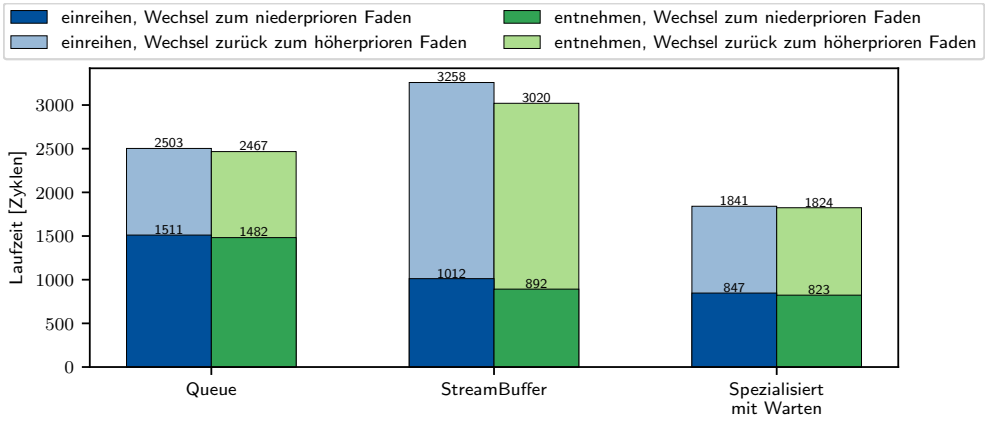

Abbildung 5.2 – Laufzeiten der verschiedenen Puffer-Implementierungen im blockierenden Fall. Die spezialisierten Varianten haben eine geringere Laufzeit als die generische Queue-Implementierung. Die StreamBuffer-Implementierung ist keine lohnenswerte Alternative, da sie erhöhte Laufzeit aufweist.

Abbildung 5.1 zeigt den Zeitbedarf von Schreibe- und Leseoperationen der drei Implementierungen. Gezeigt ist jeweils der Erfolgsfall, dass gerade nicht blockiert werden muss und zu demselben Kontrollfluss zurückgekehrt wird, der den Systemaufruf getätigt hat. Es ist zu erkennen, dass die spezialisierten Varianten jeweils weniger Zeit benötigen als die generische Variante. Dies resultiert daraus, dass Aufgabenplanung und globale Unterbrechungssperre nicht getätigt werden, da diese nicht notwendig sind. Die StreamBuffer-Implementierung benötigt eine erhöhte Laufzeit, ist dementsprechend keine lohnenswerte Alternative für dieses Szenario. Ursache dafür ist der vermehrte Aufwand für die stromorientierte Verwaltung und die ähnlich grobgranulare Unterbrechungssperre.

In Abbildung 5.2 ist der Zeitbedarf für den eventuell blockierenden Fall dargestellt, bei dem im Anschluss ein anderer Kontrollfluss die Ausführung fortsetzt. Dieser blockiert sich seinerseits direkt wieder, sodass direkt zum ursprünglichen Faden zurückgewechselt wird und somit die minimalen Wechselkosten gemessen werden. Gemessen wurde jeweils die Zeit über die Aufgabenplanung hinweg bis zur Rückkehr des anderen Kontrollflusses, der auf das jeweilige Ereignis gewartet hat. Auch hier ist zu erkennen, dass der Zeitbedarf der spezialisierten Varianten geringer ist. Dies liegt ebenso wie bei dem nicht blockierenden Erfolgsfall an der vereinfachten Unterbrechungssperre. Auch in diesem Szenario ist die StreamBuffer-Implementierung konkurrenzfähig, sodass sie in den folgenden Überlegungen nicht als Alternative selektiert wird.

5.2 Selektion spezialisierbarer Systemobjekte und geeigneter Spezialisierungen

Die Erkennung der spezialisierbaren Systemobjekte ist der nächste Schritt hin zu dem Ziel der Interaktionsspezialisierung. Diese geschieht anhand der durch die INA ermittelten Informationen im IIG in mehreren Schritten. Hierbei werden die jeweils zu den verschieden spezialisierten Implementierungen relevanten Eigenschaften und Einschränkungen der Systemaufrufstellen mit beachtet.

Grundlegende Voraussetzung ist dabei die Eindeutigkeit des zu spezialisierenden Systemobjekts. Dies bedeutet, dass nur Interaktionen spezialisierbar sind, deren beteiligte Systemobjekte mindestens teilweise statisch instanziiert werden können. Andernfalls ist eine weitere Zuordnung der Interaktio-

nen in den weiteren Auswahlschritten schlichtweg nicht möglich. Nach den Erkenntnissen aus dem vorherigen Kapitel trifft dies jedoch auf viele Systemobjekte zu, sodass dies nicht als problematische Einschränkung aufgefasst werden muss.

Im Folgenden bestimmt das Betriebssystemmodell, welche Interaktionsmuster spezialisierbar sind. Das Betriebssystemmodell weiß um die Bedeutung der verschiedenen Systemrufe zur Interaktion und ist somit in der Lage, spezialisierbare Objekte zu klassifizieren. Dies geschieht anhand einer Analyse des IIG. Für jedes Systemobjekt wird dabei die Menge der ein- und ausgehenden Kanten und die damit verbundenen anderen Systemobjekte analysiert. Jede vom Betriebssystemmodell angebotene Spezialisierung hat dabei eine Menge von Bedingungen, die erfüllt sein müssen, damit die jeweilige Spezialisierung anwendbar ist. Aus der Auswertung dieser Bedingungen in Bezug auf die Kanten eines Systemobjekts im IIG entsteht eine Liste aller möglichen Spezialisierungen.

Im nächsten Schritt wird die Interoperabilität der möglichen Spezialisierungen der verschiedenen Systemobjekte geprüft. Dies ist notwendig, da dieselbe Aufrufstelle je nach Aufrufpfad auf unterschiedlichen Systemobjekten zugreifen kann. Dabei bestimmt die konkrete Umsetzung der Spezialisierungen, ob zwei verschiedene Systemobjekte über ein und dieselbe Aufrufstelle angesprochen werden können. Hat die Spezialisierung eines Systemobjekts zur Folge, dass die zu rufende Funktion ausgetauscht werden muss und ist diese Funktion nicht als virtuelle Methode implementiert, so besteht ein Konflikt, der der Spezialisierung entgegensteht. Demnach müssen alle Aufrufstellen auf eine solche Doppelung geprüft werden.

Besteht ein Konflikt an einer Aufrufstelle, so sind mehrere verschiedene Verfahren möglich, diesen zu lösen. Die technisch einfachste Möglichkeit besteht darin, auf eine weniger stark spezialisierte Variante zurückzugreifen (bis hin zu keiner Spezialisierung), bis auf alle beteiligten Systemobjekte die gleiche Implementation der gerufenen Funktion angewendet werden kann. Dies führt in jedem Fall zu einer validen Selektion der Implementierung, aber gleichzeitig schränkt es die möglichen anwendbaren Spezialisierungen ein. Alternativ können Verfahren eingesetzt werden, die die auszuführende Implementierung erst spät, also zur Ausführungszeit anhand des verwendeten Systemobjekts binden. Ist das Betriebssystem bereits in einer objektorientierten Sprache implementiert, so sind virtuell überladene Methoden eine Implementierungsmöglichkeit. Alternativ kann an eben diesen Aufrufstellen zunächst eine Trampolinfunktion gerufen werden, die die richtige Implementierung selektiert. Dies vermeidet die Laufzeitkosten an denjenigen Aufrufstellen, die keine dynamische Selektion benötigen. Mit diesen Techniken ist es möglich, verschieden spezialisierte Systemobjekte an derselben Aufrufstelle zu verwenden.

Abschließend bleibt noch die Wahl der konkreten Spezialisierung aus den Mengen der ermittelten jeweils möglichen Spezialisierungen. Zu diesem Zweck muss zwischen Speicherbedarf der einzelnen Systemobjekte, Laufzeitaufwand der Systemaufrufe und Speicherbedarf für den Programmcode der verschiedenen Implementierungen abgewogen werden.

5.2.1 Auswahl einer validen Spezialisierung

Die beschriebenen Auswahlkriterien formulieren Bedingungen, die für die jeweiligen Aufrufstellen erfüllt sein müssen. Gleichzeitig beeinflussen die Bedingungen, die an eine Aufrufstelle gestellt sind, aber auch alle anderen mit einem Systemobjekt in Verbindung stehenden Aufrufstellen. Dementsprechend müssen auch diese Wechselwirkungen mit in Betracht gezogen werden. Das Gesamtproblem lässt sich als Bedingungserfüllungsproblem in einem Bedingungsgraphen darstellen. In dieser Darstellung ist ein algorithmisches Lösen möglich [BPS99].

Zur Lösung dieser Zusammenhänge lässt sich das Bedingungserfüllungsproblem wie folgt beschreiben. Die Aufrufstellen sind jeweils die Knoten (V) in einem Bedingungsgraphen ($G = (V, E)$). Diese Knoten werden mit Hyperkanten verbunden, die jeweils ein Systemobjekt repräsentieren (E).

Eine Hyperkante (e_i) verbindet dabei alle Aufrufstellen, die eine konkrete Instanz eines Systemobjekts betreffen.

$$V = \{v_i | i \in \text{Aufrufstellen}\}$$
$$E = \{e_s | s \in \text{Systemobjekte}, e_i \subset V, e_i \neq \emptyset\}$$
$$e_i = \{v_i | v_i \text{betrifft } s\}$$

Die Funktion $S(e)$ stellt dabei die wählbaren validen Spezialisierung des zu e zugehörigen Systemobjekts dar. Eine valide Lösung dieses Bedingungserfüllungsproblems ist eine derartige Belegung dieser Hyperkanten, dass alle Kanten, die einen Knoten betreffen, jeweils dieselbe Spezialisierungsstufe der jeweiligen Systemobjekte zugewiesen haben.

$$\forall v_i \in V, e_s, e_t \in E | (v_i \in e_s \wedge v_j \in e_s) \implies S(e_s) \cap S(e_t) \neq \emptyset$$

Je nach Aufbau des Graphen und der zur Verfügung stehenden Spezialisierungen können sich so mehrere valide Belegungen ergeben. Im allgemeinen Fall ist die Lösung von Bedingungserfüllungsproblemen NP-schwer [Kum92]. In jedem Fall ist die triviale Lösung gegeben, die keine Spezialisierungen, sondern ausschließlich die Verwendung der generischen Implementierung vorsieht. Somit ist sichergestellt, dass das Problem stets lösbar ist und die Suche nach weiteren Lösungen nur die Menge der bekannten Lösungen erweitert. Mittels Rückverfolgungsalgorithmen sind so weitere valide Belegungen bestimmbar [DF99].

Sind neben der trivialen Belegung weitere valide Belegungen möglich, so muss entschieden werden, welche verwendet werden soll. Zu diesem Zweck kann auf den Spezialisierungen eine Ordnung definiert werden, die diese Selektion beschreibt. Alternativ kann auch anhand eines Entscheidungsbaumes die zu wählende Belegung ermittelt werden.

Zum Erstellen eben dieser Entscheidungskriterien ist eine genauere Betrachtung der Auswirkungen der möglichen Spezialisierungen notwendig. Diese dient zur Ermittlung der nichtfunktionalen Implikationen aus der Verwendung einer konkreten Spezialisierung im Zusammenspiel mit den anderen verwendeten Spezialisierungen, die der Entscheidungsmetrik zugrunde liegen.

Ein mögliches Kriterium kann hierbei neben dem Laufzeitaufwand auch der Speicherbedarf der jeweiligen Spezialisierungen sein. Der Speicherbedarf setzt sich dabei aus dem für die konkrete Instanz und dem für alle Instanzen benötigten Speicher zusammen. Pro Instanz wird Speicher sowohl zum Verwalten des internen Zustandes des Objekts benötigt als auch zum Verwalten der eventuell vorhandenen Nutzdaten. Dieser befindet sich je nach Implementierung der Spezialisierung und Initialisierungsart im BSS und eventuell auch im Datensegment des resultierenden Speicherabbildes. Pro Spezialisierung wir der entsprechende Programmtext benötigt, welcher im Textsegment lokalisiert ist.

Für die hier beispielhaft betrachteten Varianten des spezialisierten Puffers stellt Abbildung 5.3 die Unterschiede im Speicherbedarf der verschiedenen Spezialisierungen in Kombination mit anderen verwendeten Implementierungen dar. Der instanzbezogene Speicherbedarf der Spezialisierungen ist geringer als der der generischen Implementierung. Ebenso ist auch der Speicherbedarf für den Programmtext der jeweiligen Spezialisierungen geringer. Während jedoch der Bedarf einer spezialisierten Instanz alternativ zu dem einer generischen Instanz in die Gesamtsumme eingeht, muss der Bedarf des Programmtextspeichers zusätzlich für jede verwendete Spezialisierung mit einbezogen werden. Dies resultiert in einer Abwägung zwischen Einsparungen pro Instanz und Mehraufwand pro verwendeter Implementierung. Ein möglicher resultierender Entscheidungsbaum könnte also sich für Spezialisierungen entscheiden, so sie denn möglich sind, aber nur diejenige Spezialisierung mit eventuell geringerer Spezialisierungstiefe verwenden, die dafür aber mehr Verwendungen abdeckt, sodass weniger Varianten benötigt werden. Konkret bedeutet dies für die

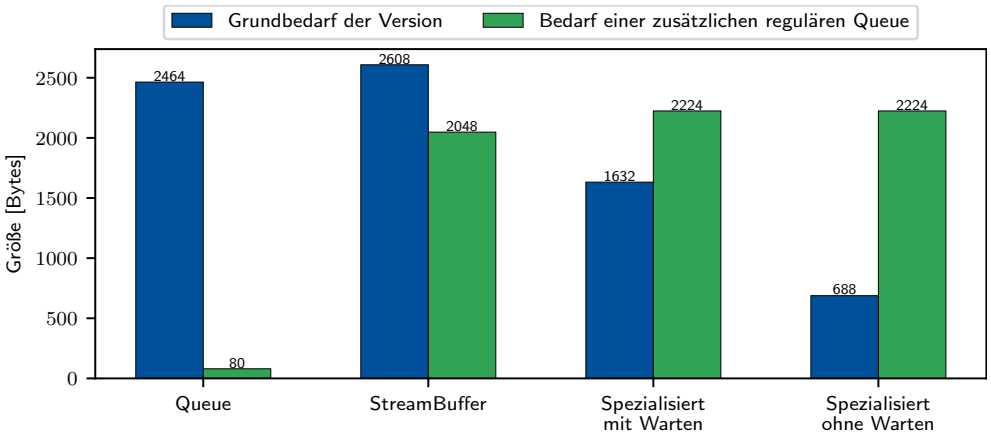

Abbildung 5.3 – Speicherbedarf für die unterschiedlichen Puffer-Implementierungen. Die spezialisierten Varianten benötigen weniger Speicher. Da sie sich weniger Code mit der regulären Implementierung teilen, wird bei Parallelbenutzung mehr zusätzlicher Speicher für den Code benötigt.

hier vorliegenden Beispiele, dass sich gegen die spezialisierte Variante ohne Blockieren und für die mit der Möglichkeit zum Blockieren entschieden wird. Deren Funktionsumfang ist eine Obermenge der Spezialisierung ohne Blockieren. Eine solche Wahl liefert also alle Verwendungsstellen eine valide Belegung im Bedingungserfüllungsgraphen.

Für die bereits im vorherigen Abschnitt verwendete Beispielanwendung (Quellcode 3.1) stellt Abbildung 5.4 den IIG dar. In diesem ist dargestellt, dass die Aufrufe, die den Puffer `storage_queue` verwenden, eindeutig zugeordnet werden können. Diejenigen Aufrufe, die die in `GuardedData` eingebetteten Semaphore `sensor_mutex` verwenden, können nicht sicher zugeordnet werden, da die Zeigeranalyse hier Raum für Unsicherheiten lässt. Da sich die Interaktionen jedoch auf unterschiedliche Klassen von Systemobjekten beziehen, beeinflusst diese Unsicherheit die Spezialisierung

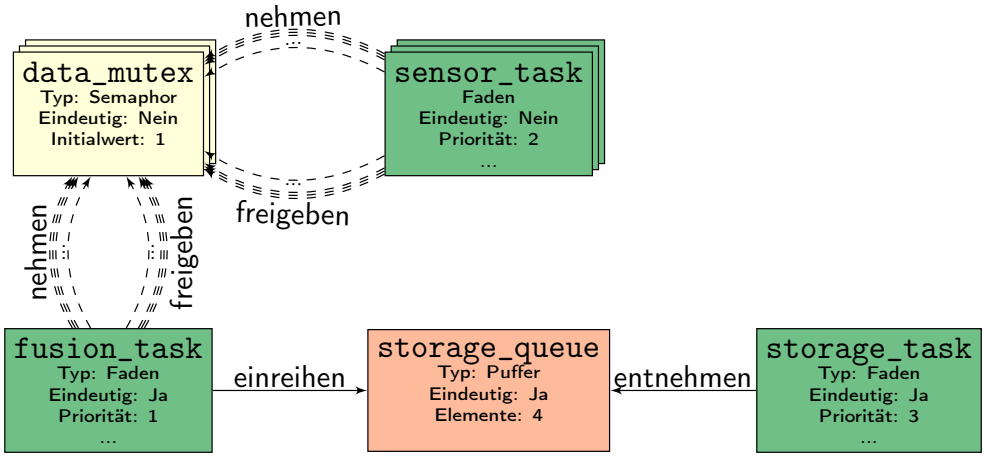

Abbildung 5.4 – Instanz- und Interaktionsgraph der Beispielanwendung aus Quellcode 3.1. Fäden (grün), Puffer (orange) und Semaphore (gelb) bilden das System. Die möglichen Interaktionen zwischen den Systemobjekten sind als Kanten dargestellt.

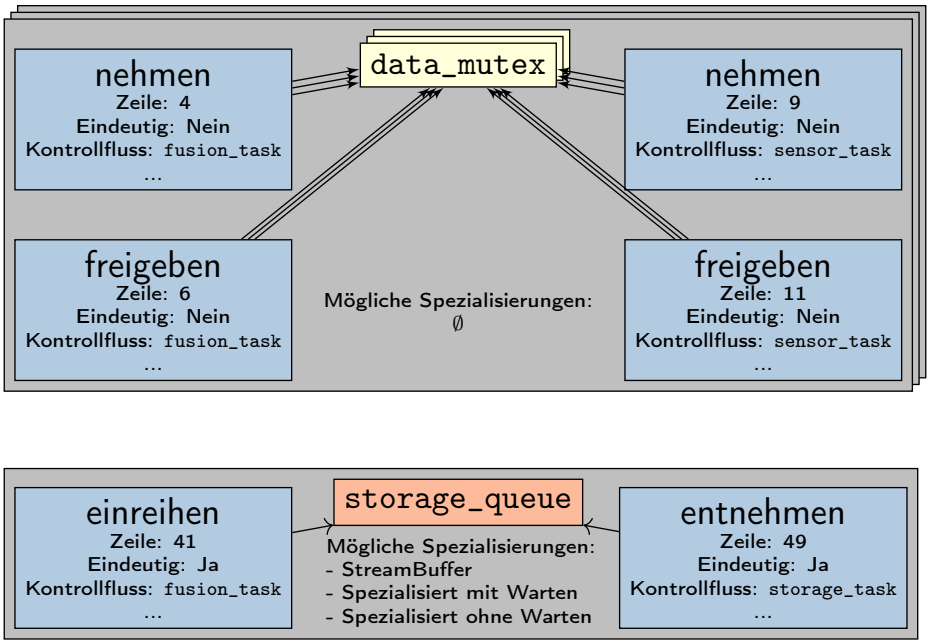

Abbildung 5.5 – Bedingungsgraph der Beispielanwendung hinsichtlich der Auswahl möglicher Spezialisierungen. Aufrufstellen (blau) sind mittels Hyperkanten (grau) an die Systemobjekte (gelb und orange) gebunden. Der Puffer `storage_queue` ist in allen Varianten spezialisierbar. Für den Semaphor `sensor_mutex` ist hingegen keine eindeutige Zuordnung der Interaktionen und damit auch keine Spezialisierung möglich.

des Puffers nicht negativ. Ferner ist erkennbar, dass die Zugriffe auf den Puffer `storage_queue` dem Muster eines einzigen Lesers und Schreibers entsprechen. Damit ist dieser ein potenzieller Kandidat für die zuvor eingeführte Spezialisierung.

In Abbildung 5.5 ist der resultierende Bedingungsgraph dargestellt. In diesem Beispiel sind die Hyperkanten schnittfrei, sodass die Entscheidung für eine Spezialisierung für die Objekte unabhängig voneinander getroffen werden kann. Für den Puffer `storage_queue` ist eine Spezialisierung mit allen vorgestellten Varianten möglich. Für die Semaphore `sensor_mutex` sind keine der vorgestellten Spezialisierungen möglich.

5.3 Evaluation

Aus den vorherigen Abschnitten wissen wir: Die Spezialisierung der Interaktion mit Systemobjekten kann einen Vorteil für nichtfunktionalen Eigenschaften eines Gesamtsystems erzielen. Es gibt ein Schema, nach dem die automatische Selektion geeigneter Spezialisierungen unter Beachtung der Randbedingungen, die das Verwendungsmuster und die Eigenschaften der Spezialisierungen berücksichtigen. Mit diesen beiden Grundlagen an der Hand wenden wir diese Spezialisierungen nun auf ein reales Problem in Form einer echten Anwendung eines eingebetteten Systems an. Ich habe dazu die bereits aus dem vorherigen Kapitel bekannten Anwendungen gewählt.

5.3.1 GPSLogger

Von dem GPSLogger wissen wir bereits, dass er aus 5 Fäden besteht, die zwei Puffer und zwei Semaphore für ihre Synchronisation und ihren Datenaustausch verwenden. In Abbildung 5.6 ist der aus der INA hervorgehende Interaktionsgraph dargestellt. Dieser zeigt, dass der Puffer `buttonQueue` nur von dem Faden für die Ansteuerung des Displays (`Display`) gelesen wird und ausschließlich der Faden, der die Eingabe der Tasten überwacht (`Buttons`) in diesen Puffer schreibt. Damit ist genau dieser Puffer sicher Ziel der angesprochenen Spezialisierung. Dies trifft ebenso auf den zwischen dem `SD`-Faden und dem `GPS`-Faden verwendeten Puffer `sdQueue` zu, die damit ebenfalls potenzielles Ziel der Spezialisierungen wird. Für die anderen Kommunikationsobjekte ist zu erkennen, dass nicht alle Interaktionen sicher einem Ziel zugeordnet werden können. Daraus ergibt sich hier keine Möglichkeit für die gegebene Spezialisierungsform. Die Anordnung der Aufrufstellen erlaubt für diese beiden ebenfalls die beschriebene Spezialisierung. Problematisch ist hingegen, dass jeweils Zeitbeschränkungen für die maximale Wartezeit der Systemaufrufe verwendet werden. Die beiden vorgestellten Implementierungen beherrschen diese Funktionalität jedoch nicht. Bei genauerer manueller Betrachtung fällt jedoch auf, dass die Aufrufstellen, die die `sdQueue` verwenden, jeweils in Endlosschleifen liegen, die die Zugriffe wiederholen, bis diese erfolgreich sind. Da die Schleifen, wie in Abbildung 5.2 dargestellt, keinen anderen Code enthalten als den Systemaufruf, ist dieses Konstrukt äquivalent zu einem Aufruf mit unendlicher Wartezeit, welcher von der spezialisierten Variante unterstützt wird.

Für die Semaphore `GPSData` und `SDData` zeigt der IIG keine eindeutige Zuordnung der Interaktionen des Fadens `Display`. Dies liegt in der nicht erfolgreichen statischen Zeigeranalyse begründet, die die Referenzen der Objekte nicht eindeutig nachverfolgen kann. Daher ist hier keine Spezialisierung möglich. Manuell wäre dies jedoch durchführbar, da auf C++-Sprachebene diese Zuordnung eindeutig ist, jedoch bei der Übersetzung zum LLVM-Zwischencode verloren geht.

Insgesamt kann somit von den vier Synchronisationsobjekten mindestens die `sdQueue` spezialisiert werden. Dies resultiert für die Zugriffe auf diese zu den in Abschnitt 5.1.2 beschriebenen

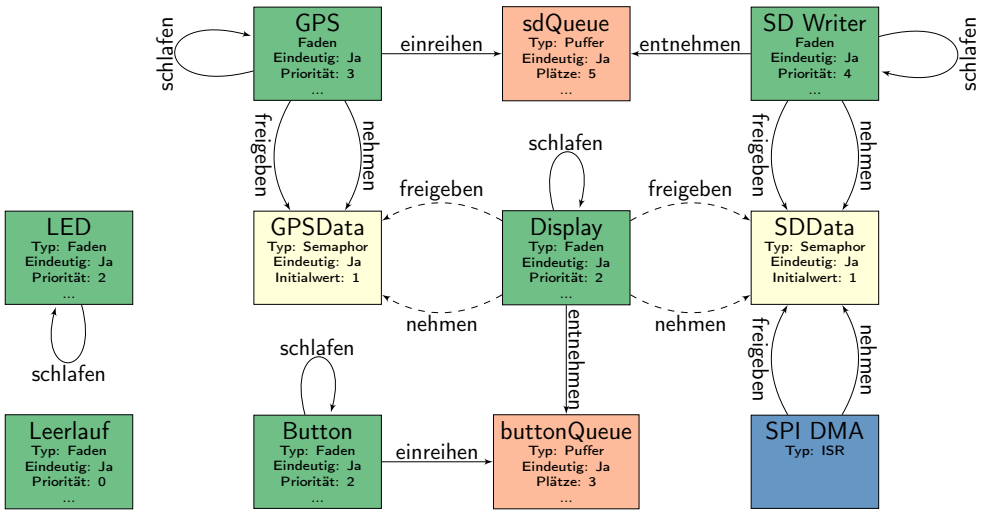

Abbildung 5.6 – Interaktionsgraph des GPSLoggers. Nur die Interaktion des Display-Fadens kann nicht eindeutig dem richtigen Semaphor (GPSData) zugeordnet werden, die anderen Interaktionen sind eindeutig zuordenbar.

Verbesserungen der Zugriffszeiten. Für das Gesamtsystem resultiert daraus ein Mehrbedarf an Speicher von 1 088 Bytes im Textsegment und 5 Bytes im Datensegment.

Quellcode 5.2 Widersprüchliche Verwendung von Systemaufrufen mit Wartezeit. Der Aufruf von `xQueueReceive()` blockiert nur 50 Zeiteinheiten. Schlägt dieser fehl, so wird jedoch kein alternativer Programmcode ausgeführt, sondern direkt der Systemaufruf in der Endlosschleife erneut aufgerufen. Damit entspricht dies einer unendlich blockierenden Semantik ohne Zeitbeschränkung.

```
1  void runSDMessageLoop()
2  {
3    /* ... */
4    while(true)
5      {
6        SDMessage msg;
7        if(xQueueReceive(sdQueue, &msg, 50))
8          {
9            switch(msg.messageType)
10             {
11             case RAW_GPS_DATA:
12               saveRawData(msg);
13               usbDebugWrite("Receive   GPS Data: %s\r\n", msg.rawData.rawDataBuf);
14               break;
15             default:
16               break;
17             }
18         }
19
20     }
21   /* ... */
22 }
```

5.3.2 LibrePilot

Die Analyse der Fluglagesteuerung aus dem LibrePilot führt zu der Erkenntnis, dass es in der vorliegenden Form keine spezialisierbaren Interaktionen gibt. Dies ist auf mehrere Ursachen zurückzuführen.

eindeutige Aktivitätsträger Wie bereits in Abschnitt 4.4.2.1 beschrieben, sind die meisten Instanzen der Systemobjekte momentan nicht eindeutig identifizierbar. Dies liegt primär in der Verwendung von Programmiermustern begründet, die die Instanziierung als bedingt erscheinen lassen.

eindeutige Ziele Die zweite große Herausforderung ist die Zuordnung der Systemaufrufe zu den verwendeten Systemobjekten. Die verwendeten Systemobjekte werden in diesem Projekt an größere Kontextobjekte gebunden und anhand dieser indirekt ausgewählt. Diese Indirektion über Zeiger führt zu dem Problem, dass die Zeigeranalyse diese auflösen muss. Dies gelingt nur teilweise und die daraus resultierende pessimistische Überabschätzung führt dazu, dass vermehrt zu viele Fäden scheinbar auf ein Synchronisationsobjekt zugreifen und so die Spezialisierung verhindern.

Eine genauere Betrachtung dieser grundlegenden Probleme findet im folgenden Kapitel 6 querschneidend über alle analysierten Anwendungen hinweg statt.

5.4 Zusammenfassung

In diesem Kapitel habe ich die Spezialisierung von Interaktionen in dynamisch konfigurierten eingebetteten Systemen betrachtet. Zunächst wurden mögliche Kandidaten für eine solche Spezialisierung vorgestellt und deren Mehrwert im Blick auf veränderte nichtfunktionale Anforderungen betrachtet. Im Folgenden habe ich die systematische Analyse der Verwendungsstellen betrachtet und die daraus hervorgehende automatische Detektion spezialisierbarer Verwendungsstellen erläutert. Der letzte Abschnitt demonstriert die praktische Umsetzbarkeit dieser Spezialisierungen am Beispiel realer Anwendungen.

Forschungsfrage

Im Hinblick auf die zu Beginn gestellten Forschungsfragen kann ich nun weitere Antworten geben. Ergänzend zu den Erkenntnissen aus dem vorherigen Kapitel ergibt sich zur ersten Forschungsfrage (FF1) die Erkenntnis, dass auch die Interaktionen der Systemobjekte mittels dieser Spezialisierungen verbessert werden können. Dies führt an der hier vorgestellten Spezialisierung zu einer Beschleunigung der spezialisierten Systemaufrufe. Wie auch bereits bei der Instanziierung bleiben die nicht spezialisierten Systemaufrufe dabei unbeeinflusst. Deren Verhalten ändert sich nicht und der Programmcode dieser bleibt unverändert.

Die in der zweiten Forschungsfrage (FF2) betrachtete Veränderung von Laufzeitaufwand und Speicherbedarf hängt von der konkreten Spezialisierung an den jeweiligen Verwendungsstellen im Einzelnen und an der globalen Verwendung einer Spezialisierung im Gesamtsystem ab. An den Verwendungsstellen führt die hier vorgestellte Spezialisierung zu einer Verringerung der Ausführungszeiten. Der Speicherbedarf der Gesamtanwendung steigt, sobald verschiedene Implementierungen des gleichen funktionalen Verhaltens vorgehalten werden müssen. Dies bedeutet, verwenden alle Verwendungsstellen dieselbe Spezialisierung, so führt dies zu einer Speicherbedarfsreduktion. Werden jedoch verschiedene Implementierungen aufgrund unterschiedlicher Anforderungen benötigt, so steigt der Speicherbedarf für den Programmtext insgesamt leicht an.

Damit sind die ersten beiden Forschungsfragen für die Startphase des Lebenszyklus der Anwendung in Ausführung sowie für die Arbeitsphase behandelt. Verbleibend ist noch die dritte Forschungsfrage (FF3) nach den hinderlichen Programmiermustern. Im nun folgenden Kapitel 6 wird diese Fragestellung entsprechend beleuchtet und beantwortet.

6

Idiomatik und Semantik

Hindernisse und Einflussnahme auf die Spezialisierbarkeit

In den vorherigen Kapiteln habe ich mögliche anwendungsgewahre Betriebssystemspezialisierungen für dynamisch konfigurierten Systeme vorgestellt. Dabei sind bereits Probleme, Hindernisse und deren Ursachen am Rande aufgetreten, die ich in diesem Kapitel systematisch behandle. Dabei geht es zum einen um die Einordnung ihrer Herkunft und zum anderen um die Frage, ob diese ein grundsätzliches Hindernis für die Spezialisierung sind oder ob mit weiteren Arbeiten Lösungswege möglich sind. Zu diesem Zweck stelle ich zunächst zwei exemplarische Ausschnitte aus den Fallbeispielen GPSLogger und LibrePilot vor. Im Folgenden beschreibe ich die verschiedenen Ebenen der Problemursache und ordne die in den Fallbeispielen auftretenden Probleme entsprechend ein. Dabei gebe ich Information darüber, ob die Probleme grundsätzlicher Art sind oder wie, mit welchem Nutzen und zu welchem Preis hier welche Verbesserungen möglich wären.

6.1 Fallbeispiele

Zum besseren Verständnis der Probleme und Hindernisse verwende ich im Folgenden zwei demonstrative Beispiele aus den Fallstudien GPSLogger und LibrePilot. Diese beinhalten eine exemplarische Zusammenstellung der aufgetretenen Probleme und grundlegenden Hindernisse.

6.1.1 LibrePilot

Die Anwendung wurde bereits in Abschnitt 4.4.2.1 vorgestellt. Sie ist in der Sprache C implementiert, besteht aus mehreren Modulen, die für die einzelnen Hardware- und Softwarekomponenten zuständig sind und ist über zur Laufzeit ausgewertete Konfigurationsoptionen bereits auf Ebene der Abstraktionen konfigurierbar.

In Quellcode 6.1 ist ein Ausschnitt aus dem LibrePilot dargestellt. Diese zeigt einen Teil des Codes auf dem Weg zum SSP aus den verschiedenen Modulen der Anwendung und den dazugehörigen Anweisungen für den Binder. Ab Zeile 3 ist dargestellt, wie manuell ein Konstruktormechanismus implementiert wurde. Diesen verwenden die Module für die Höhenregelung und GPS-Positionsbestimmung zum Registrieren ihrer Initialisierungs- und Startfunktionen (Zeilen 33 und 44). Die in Sektionen mit definierten Namen abgelegten Funktionszeiger (Zeile 11) werden vom Binder zu einem Feld von Zeigern zusammengetragen (Zeile 67) und zur Laufzeit aufgerufen (Zeile 16).

Während des Starts werden jeweils dynamisch Systemobjekte instanziiert (Zeilen 25, 41 und 31). Dies geschieht teils bedingt anhand der Laufzeitkonfiguration (Zeile 40). Diese beinhalten das Auslesen von Konfigurationsoptionen aus einem Konfigurationsspeicher und dem Prüfen auf vorhandene Peripherie. Außerdem werden Zusicherungen geprüft und verhindern eventuell den Fortschritt der Anwendung (Zeile 30).

6.1.2 GPSLogger

Als zweite Fallstudie verwende ich einen Ausschnitt aus dem bereits aus Abschnitt 4.4.2.2 bekannten GPSLogger. Dieser ist mittels der Sprache C++ implementiert, verwendet Objektorientierung als Paradigma und besteht aus mehreren Klassen, die die Funktionsbausteine analog zu der verwendeten Hardware realisieren.

In Quellcode 6.2a ist der verwendete Ausschnitt aus dem Programmcode abgebildet. Dieser zeigt Teile der Klassen zur Verwaltung von GPS-Positionsinformationen (Zeile 30) und deren Verwendung zur Ansteuerung einer Anzeige (Zeile 22). Das GPS-Datenmodell folgt dabei dem Entwurfsmuster Einzelstück (en. Singleton), sodass der Zugriff nur über die entsprechende Klassenmethode (Zeile 32) möglich ist. Konkurrierende Zugriffe auf die internen Daten werden dabei von einem Semaphor geschützt (Zeile 47). Dabei wird das Entwurfsmuster *Ressourcenallokation ist Initialisierung (RAII)* verwendet, mittels der Klasse `MutexLocker` implementiert (Zeile 1) und im Datenmodell verwendet (Zeile 37).

Da die Anwendung aus mehreren Übersetzungseinheiten besteht, müssen diese für die Verarbeitung durch ARA miteinander gebunden werden. Quellcode 6.2b zeigt Ausschnitte aus dem Zwischencode und deren Änderungen durch das Binden. Zu sehen ist, dass Typen verloren gehen (Zeile 60) und sich Signaturen von Funktionen ändern (Zeile 74).

Quellcode 6.1 Fallbeispiel aus dem LibrePilot. Zu sehen ist ein Zusammenschnitt des manuell implementierten Konstruktormechanismus und bedingte Instanziierung von Systemobjekten. Die Module (GPS und Altitude) registrieren jeweils ihre Initialisierungs- und Startfunktionen (Zeilen 33 und 44). Die Funktionszeiger werden als strukturierte Daten in einer ausgewiesenen Sektion abgelegt (Zeile 11) und vom Binder(Skript Zeile 67) zusammengetragen. Im Hauptprogramm werden diese über dieses Feld von Funktionszeigerpaaren (Zeile 16) aufgerufen(Zeilen 50 und 52).

```
1  #include <FreeRTOS.h>
2
3  /* Nachbau der Konstruktorfunktionalität */
4  typedef void (*initcall_t)(void);
5  typedef struct {
6      initcall_t init;
7      initcall_t start;
8  } initmodule_t;
9
10 #define MODULE_INIT(ifn, sfn) static initmodule_t __initcall_##ifn __attribute__((__used__)) \
11        __attribute__((__section__(".initcallmodule.init"))) = { .init = ifn, .start = sfn }
12
13 extern initmodule_t __module_initcall_start[], __module_initcall_end[];
14
15 #define MODULE_DO_ALL(fun)                                                          \
16   { for (initmodule_t *im = __module_initcall_start; im < __module_initcall_end; im++) {  \
17       if (im->fun) {                                                                \
18         (im->fun)(); }                                                              \
19     }                                                                               \
20   }
21
22
23 /* Altitude-Modul */
24 void altitude_init(void) {
25   altitude_queue = xQueueCreate(3, 4);
26   /* ... */
27 };
28 void altitude_task(void *param) {/* Arbeitscode */};
29 void altitude_start(void) {
30   ASSERT(altitude_queue);
31   xTaskCreate(altitude_task, "altitude", 100, NULL, 3, NULL);
32 }
33 MODULE_INIT(altitude_init, altitude_start);
34
35
36 /* GPS-Modul */
37 void gps_init(void) {/* Initialisierungscode */};
38 void gps_task(void *param) {/* Arbeitscode */};
39 void gps_start(void) {
40   if (config_is_gps_enabled() && gps_is_present()) {
41     xTaskCreate(gps_task, "gps", 100, NULL, 1, NULL);
42   }
43 }
44 MODULE_INIT(gps_init, gps_start);
45
46
47 /* Haupt-Modul */
48 int main() {
49   /* ... */
50   MODULE_DO_ALL(init);
51   /* ... */
52   MODULE_DO_ALL(start);
53   /* ... */
54   vTaskStartScheduler();
55 }
```

Skript für den Binder

```
60 SECTIONS {
61   /* ... */
62   .text:
63   {
64     /* ... */
65     . = ALIGN(4);
66     __module_initcall_start = .;
67         KEEP(*(.initcallmodule.init))
68     . = ALIGN(4);
69     __module_initcall_end   = .;
70     /* ... */
71   }
72     /* ... */
73
74 }
```

Konfigurationsspeicher

```
80 telemetry: no
81 gps: yes
82 ...
83 camera: no
```

Quellcode 6.2 Fallbeispiel aus dem GPSLogger. Zu sehen ist (a) ein gekürzter Ausschnitt der Anwendung und (b) Ausschnitte aus dem Zwischencode mit den Auswirkungen des Bindens vor dem Durchführen der Analysen. Verwendet werden die Entwurfsmustern RAII (Zeile 1) und Singleton (Zeile 32), das Paradigma Objektorientierung (Zeile 11) sowie aus menschlicher Sicht unbedingte eindeutige Instanziierung von Systemobjekten (Zeile 47).

```
1  class MutexLocker {
2  public:
3    MutexLocker(SemaphoreHandle_t mtx) {
4      xSemaphoreTake(mutex, portMAX_DELAY);
5    }
6    ~MutexLocker() { xSemaphoreGive(mutex); }
7  private:
8    SemaphoreHandle_t mutex;
9  };
10
11 class GPSSatellitesData {
12   int *data;
13   /* ... */
14 };
15
16 class DerivedGPSSatellitesData:
17   public GPSSatellitesData {
18   OtherType_t additional;
19   /* ... */
20 };
21
22 void vDisplayTask(void *pvParameters) {
23   /* ... */
24   GPSSatellitesData satellites = \
25        GPSDataModel::instance().getSatData();
26   /* ... */
27   DerivedGPSSatellitesData dd;
28 }
```

```
30 class GPSDataModel {
31 public:
32   static GPSDataModel & instance(){
33     return GPSDataModel::_the_instance;
34   };
35
36   GPSSatellitesData getSatData() const {
37       MutexLocker lock(mutex);
38       return satellitesData;
39     };
40   /* ... */
41 private:
42   GPSSatellitesData satellitesData;
43   SemaphoreHandle_t mutex;
44   /* ... */
45
46   GPSDataModel() {
47     mutex = xSemaphoreCreateMutex();
48     /* ... */
49   };
50   GPSDataModel( const GPSDataModel &c );
51   GPSDataModel& operator=( const GPSDataModel &c );
52   static GPSDataModel _the_instance;
53 };
```

(a) Quellcode der Anwendung

```
60 - %class.GPSDataModel = type { %class.GPSSatellitesData, i32* }
61   %class.DerivedGPSSatellitesData = type { %class.GPSSatellitesData, i32* }
62   %class.GPSSatellitesData = type { i32* }
63 - %class.MutexLocker = type { i32* }
64
65     /* ... */
66
67   define void @_Z12vDisplayTaskPv(i8* %0) {
68     ...
69 -   %5 = call %class.GPSDataModel* @_ZN12GPSDataModel8instanceEv()
70 +   %5 = call %class.DerivedGPSSatellitesData* @_ZN12GPSDataModel8instanceEv()
71
72     /* ... */
73
74 - define i32* @_ZNK12GPSDataModel10getSatDataEv(%class.GPSDataModel* %0) {
75 + define i32* @_ZNK12GPSDataModel10getSatDataEv(%class.DerivedGPSSatellitesData* %0) {
76     ...
77 -    call void @_ZN11MutexLockerD2Ev(%class.MutexLocker* %4)
78 +    call void @_ZN11MutexLockerD2Ev(%class.GPSSatellitesData* %4)
```

(b) Ausschnitt aus dem LLVM Zwischencode. Farblich hervorgehoben sind die Änderungen aufgrund des Bindens vor dem Durchführen der Analysen.

6.2 Die Ebenen der Problemursachen

Die Ursachen für die auftretenden Hindernisse und Probleme sind auf verschiedenen Ebenen verortet. Diese beginnen beim Design und der Verwendung der Betriebssystemschnittstelle und gehen über Übersetzer, Werkzeuge und Idiomatik bis hin zu der verwendeten Programmiersprache. Je nach Ebene sind dementsprechend andere Lösungswege notwendig.

6.3 Design der Betriebssystemschnittstelle

Die charakterisierende Designentscheidung dynamisch konfigurierter Betriebssysteme ist die, dass die Konfiguration und insbesondere die Definition der im zur Laufzeit vorhandenen Systemobjekte zur Laufzeit geschieht.

6.3.1 Beliebigkeit der Instanziierung

Dies bedeutet, jederzeit dürfen neue Systemobjekte erzeugt werden. Die Folge davon für die reguläre Betriebssystemimplementierung ist, dass zwanghaft dynamisch wachsende Datenstrukturen zur Verwaltung verwendet werden müssen. Zudem ist zwingend eine Freispeicherverwaltung zur Laufzeit notwendig, um den benötigten Speicher bereitzustellen. Mit diesem Modell wird die Illusion aufrechterhalten, dass eine beliebige Menge von Systemobjekten gleichzeitig existieren kann. Faktisch ist dies aber nicht möglich, da die Ressourcen eines physischen Systems immer endlich sind.

Für die statische Spezialisierung bedeutet dies, dass nicht alle Systemobjekte statisch spezialisierbar sein können. Diejenigen, deren Instanziierung auch konzeptionell tatsächlich erst zur Laufzeit entschieden wird, können keinesfalls statisch behandelt werden.

Beim GPSLogger ist dieses Problem nicht aufgetreten. Alle Systemobjekte wurden konzeptionell statisch erzeugt und konnten demnach auch statisch instanziiert werden. Auch bei den Interaktionen wären konzeptionell alle daran beteiligten Objekte statisch bestimmbar. Beim LibrePilot hingegen ist dies nicht der Fall. Wie in Quellcode 6.1 Zeile 40 zu sehen, werden hier Systemobjekte sowohl in Abhängigkeit von Einträgen eines Konfigurationsspeichers als auch abhängig von dem tatsächlichen Vorhandensein von Hardwarekomponenten instanziiert. Dazu werden weitere Systemobjekte in Schleifen instanziiert, deren Grenzen ebenfalls von diesen Laufzeitzuständen abhängt. Dementsprechend ist statische Spezialisierung für diese Systemobjekte so nicht möglich.

Ein möglicher Lösungsansatz wäre hier eine pessimistische Überabschätzung zu liefern, indem jedes optionale Systemobjekt wie eine partielle Instanziierung ohne die Phasen der Registrierung im Betriebssystem behandelt würde. Damit könnte eine obere Schranke des Speicherbedarfs und der Anzahl dieser Objekte gegeben werden. Dies führt jedoch auch zu einem im Gegensatz zu der dynamischen Variante erhöhten Speicherbedarf, wenn nicht alle optionalen Systemobjekte zeitgleich existieren können. Ist die Häufigkeit jedoch unbekannt, wie die der Sensordatenverarbeitungsfäden in der Beispielanwendung aus Quellcode 3.1 Zeile 27, so ist keine statische Lösung möglich.

6.3.2 Beliebigkeit der Interaktionspartner

Folge der Beliebigkeit der Instanziierung ist, dass auch die Interaktionspartner beliebig sind. Während bei statisch konfigurierten Systemen die vorhandenen Systemobjekte eindeutig benannt sind, ist dies entsprechend bei dynamisch konfigurierten Systemen nicht der Fall. Hier werden die Systemobjekte zwangsläufig anhand von zur Laufzeit erzeugten Referenzen identifiziert, die von der Erzeugungsstelle zur Verwendungsstelle weitergegeben werden müssen. Während im mentalen

Modelle eines Entwicklers die Interaktionspartner feststehen, hindert die dynamische Schnittstelle dabei, diese Informationen vollständig auszudrücken [Kla+14]. Vielmehr werden Abhängigkeiten implizit verwendet, ohne diese explizit formulieren zu können, da die Betriebssystemschnittstelle keine Möglichkeiten vorsieht, diese auszudrücken. Gleichzeitig ist es aufgrund der Beliebigkeit der Instanziierung auch nicht immer möglich, die Gesamtmenge der möglichen Interaktionspartner zu bestimmen.

Für die statische Spezialisierung bedeutet dies, dass davon betroffene Interaktionen nicht spezialisierbar sind. Diese müssen stets den schlechtmöglichsten Fall annehmen, dass beliebige Systemobjekte beliebig interagieren.

Beim GPSLogger ist dieses Problem konzeptionell nicht aufgetreten. Eine manuelle Spezialisierung könnte hier alle Interaktionen entsprechend genau den richtigen Systemobjekten zuordnen, sodass eine vollständige statische Spezialisierung möglich wäre. Momentan hindert zwar beispielsweise die Zuordnung des Semaphors zum Schutz der GPS-Positionsdaten (Zeile 4), jedoch wäre dies grundsätzlich bei besserer Sprachunterstützung lösbar (siehe Abschnitt 6.7). Anders ist die Situation beim LibrePilot. Da hier nicht alle Systemobjekte eindeutig sind, ist eine Spezialisierung der Interaktionen so statisch nicht vollständig möglich. Aufgrund der Modularität des Systems wäre jedoch grundsätzlich eine auf die statisch bekannten Instanzen eingeschränkte Spezialisierung möglich. Für die Beispielanwendung wäre eine bei besserer Sprachunterstützung erkennbar, dass die uneindeutig instanziierten Systemobjekte für die Sensordatenverarbeitung immer paarweise vorkommen, sodass hier die vorgestellten Interaktionsspezialisierungen möglich wären.

Eine mögliche Lösung wäre es, die Betriebssystemschnittstelle dahingehend zu erweitern, dass eine Beschreibungsmöglichkeit für die bisher nur impliziten Zusammenhänge ergänzt wird. Die Entwickler wären so in der Lage, die in ihrem mentalen Modell vorhandenen Beziehungen auszudrücken, sodass die statische Spezialisierung diese verwenden kann. Dies hat jedoch einen erhöhten Aufwand bei der Entwicklung zur Folge, da diese Informationen niedergeschrieben und stets aktuell gehalten werden müssen.

Im allgemeinen Fall ist dieses Problem nicht lösbar, da gerade diese Freiheit Kern der dynamischen Betriebssystemschnittstelle ist. In den realen Anwendungen ist jedoch zu erkennen, dass die meisten Verwendungen gutmütig sind, sodass die Spezialisierung anwendbar ist. Die Hinderungsgründe liegen einer statischen Spezialisierung liegen zumeist in den anderen Ebenen verortet. Die verbleibenden Stellen beschränken sich auf den Teil der Anwendung, die die Flexibilität der dynamischen Schnittstelle explizit verwendet und dementsprechend gewollt dynamisch arbeitet.

6.3.3 Pessimistische Grundannahmen

Die Freiheit des dynamischen Betriebssystemmodells hat zur Folge, dass an allen Stellen der Betriebssystemimplementierung mit einer pessimistischen Grundhaltung vorgegangen werden muss. Dies bedeutet, dass Verwaltungsdatenstrukturen beliebige Systemobjektkonfigurationen abbilden können müssen. Zugriffe auf diese müssen beliebige Zugriffsmuster unterstützen, sodass die Synchronisation dementsprechend ausgelegt werden muss. Auch kann der Erfolg von Systemaufrufen nicht garantiert werden, da beispielsweise notwendige Betriebsmittel nicht zur Verfügung stehen.

Dies führt dazu, dass die Anwendungen zusätzlichen Programmcode enthalten, der diese Fehlerfälle behandelt (Quellcode 6.1 Zeile 30). Dieser Code wiederum ist ein Hindernis für die statische Spezialisierung, wenn er nicht mit entsprechender Sprachunterstützung gekennzeichnet ist.

Die statisch spezialisierten Systemaufrufstellen haben dieses Problem nicht mehr. Ihr Erfolg kann für instanziierende Systemaufrufe statisch berechnet werden, sodass eine Fehlerbehandlung überflüssig ist. Damit könnte der Anwendungscode vereinfacht werden und damit weitere Kom-

ponenten des Systems spezialisierbar machen. Dies hätte auch wieder positiven Einfluss auf die statische Spezialisierbarkeit, da die Analysierbarkeit dadurch verbessert wird.

Sichtbar ist dieses Problem im LibrePilot in Zeile 30. Hier behindert die Fehlerbehandlung die statische Spezialisierung, da der Faden in der Folgezeile nur bedingt instanziiert wird. Geschieht diese Fehlerbehandlung nicht mit Sprachunterstützung, so ist die Existenz des Fadens nicht sicher statisch berechenbar.

Als Lösungsansatz könnten die Ergebnisse der Analysen an die Programmierer zurückgegeben werden, sodass sie den Code entsprechend umbauen und den Fehlerbehandlungscode als solchen auszeichnen. Dies hätte jedoch zusätzlichen Mehraufwand für die Programmierer zur Folge und ist nur möglich, wenn es entsprechende Sprachunterstützung gibt. Besser wäre eine automatisierte wiederholte Anwendung der Spezialisierung, sodass die jeweils gewonnenen Informationen mit als Eingabe für die erneute Analyse dienen. Damit wäre erkennbar, dass hier nie der Fehlerfall eintreten kann.

Insgesamt treffen die pessimistischen Annahmen bei meinem Ansatz nur auf diejenigen Systemobjekte und deren Interaktionen zu, die nicht statisch spezialisierbar sind. Alle spezialisierbaren Objekte und Interaktionen haben analog zu den statisch konfigurierten Systemen wohl definierte Eigenschaften und Verwendungen, sodass weniger pessimistische Algorithmen und Datenstrukturen verwendet werden können, solange die unbekannte Interaktion mit unbekannten Systemobjekten ausgeschlossen werden kann.

6.3.4 Einfachheit der Schnittstelle

Nach Raskin sind ein geringes mentales Modell und ein eindeutiger Lösungsweg wichtige Kriterien einer guten Schnittstelle zwischen Mensch und Maschine [Ras00]. Die Schnittstelle muss jedoch ausdrucksstark genug für die auszudrückenden Intentionen sein, um zu einem guten Ergebnis zu gelangen.

Im Fall von FreeRTOS ist es beispielsweise nicht möglich, das paarweise zusammengehören zweier Objekte auszudrücken. Das resultiert in dem Problem, dass solch ein Zusammenhang nur implizit im Programmcode als Interaktion niedergeschrieben sein kann. Daraus resultierend kann diese Information auch nicht direkt für die Spezialisierung verwendet werden.

Dieses Problem tritt beispielsweise in der Beispielanwendung bei der Sensordatenverarbeitung auf. Dort werden Faden und Semaphor (Mutex) jeweils paarweise instanziiert (Zeilen 27, 13). Es führt dazu, dass die Interaktionen des Fadens nicht direkt dem zugehörigen Semaphor zugeordnet werden. Stattdessen muss zunächst von einer Verwendung eines beliebigen Semaphors ausgegangen werden. Gleiches trifft auch auf das gleichartige Muster im GPSLogger und LibrePilot zu.

Dieses ist ein grundsätzliches Problem des einfachen Systemmodells der unabhängigen Systemobjekte. Damit steht es der statischen Spezialisierung entgegen, da dem Modell und der dazugehörigen Schnittstelle die Ausdrucksmittel für derartige Information fehlen.

Als Lösung wäre hier eine Erweiterung der Betriebssystemschnittstelle denkbar, die die entsprechende Information explizit abbildet. Dies könnte beispielsweise ein zusätzlicher Parameter sein, der die Menge der Fäden angibt, die auf einen Semaphor verwenden dürfen. Die Kosten dieser Erweiterung wären vergleichsweise gering, da beim Programmieren die entsprechenden Mengen konzeptionell bekannt sein müssen. Passend zu dem dynamischen Modell wäre somit ein Systemaufruf zum Modifizieren dieser Mengen notwendig. Eine statische Analyse wäre damit in der Lage, eben diese Mengen auch statisch vorauszuberechnen, solange diese Information statisch berechenbar ist.

6.4 Verwendung des Betriebssystems

Eine weitere Gruppe von Problemen resultiert aus der Verwendung des Betriebssystems. Die Probleme stehen teilweise im Zusammenhang mit dem Design der Betriebssystemschnittstelle und den daraus hervorgehenden Verwendungsmöglichkeiten, teilweise sind sie jedoch auch organisatorischen Ursprungs.

6.4.1 Wahl des Betriebssystems

Eine Umfrage unter Softwareentwicklern für eingebettete Systeme hat ergeben, dass die Entscheidung, das eine oder andere Betriebssystem zu verwenden, nicht unbedingt aufgrund funktionaler oder nichtfunktionaler Eigenschaften dieser getroffen werden. Vielmehr gaben 69% der Befragten an, dass sie schlicht dasselbe Betriebssystem verwenden wie in einem vorherigen Projekt, da es bereits bekannt ist. Auch gaben 24 % an, dass die Entscheidung für ein anderes Betriebssystem nicht in ihrer Entscheidungsgewalt lag [EET19]. Diese Aussagen lassen darauf deuten, dass die Flexibilität nicht notwendig ist, sondern unangefordert Teil des Betriebssystems ist. Daraus ergibt sich die Annahme, dass ebenso ein statisch konfiguriertes Betriebssystem zur Implementierung einer gleichwertigen Lösung verwendet werden könnte.

Eine Lösung des ursprünglichen Problems wäre nur, die Entscheidung problemspezifisch umzusetzen. Für die hier vorgestellten automatischen Spezialisierungen ist dies jedoch kein Problem. Vielmehr ist es so, dass die automatische Spezialisierung genau dieses Problem löst, indem sie nicht verwendete Flexibilität in der Verwendung des Betriebssystems erkennen und durch Umformung in ein statisches System entfernen. Es verbleiben nur die echten dynamischen Entscheidungen, bei denen ein dynamisch konfiguriertes System einen Mehrwert gegenüber einem statisch konfigurierten System bietet.

6.4.2 Pseudodynamische Entscheidungen

Das dynamische Programmiermodell leitet dazu an, diese dynamische Entscheidungsfreiheit auch für Entscheidungen zu verwenden, die eigentlich bereits zur Implementierungszeit, mindestens aber zur Übersetzungszeit entschieden werden sollten und entschieden werden können. Stattdessen werden sie jedoch in die Ausführungszeit verlagert.

Beispiele dafür sind: Systemobjekte, die indirekt per dynamisch allokierter Referenz anstatt statisch feststehender Bezeichner angesprochen werden (Beispielanwendung Zeilen 28 und 57). Besonders problematisch ist dies, wenn die Speicherbereiche dynamisch allokiert werden, da dann mehrere Stufen der Referenzen nachverfolgt werden müssen. Kontrollfluss, der abhängig vom Erfolg anderer Systemaufrufe ist, die bei statischer Konfiguration keinen Misserfolg haben könnten (LibrePilot Zeile 30). Listen von aufzurufenden Funktionen, die eigentlich zur Übersetzungszeit bekannt sind, werden scheinbar dynamisch zusammengesammelt (LibrePilot Zeilen 11 und 67).

Bei einem statisch konfigurierten System würden alle diese Entscheidungen, deren Ergebnis spätestens zur Übersetzungszeit (Bindezeit) feststeht, auch statisch ausgedrückt werden. Für die automatische Spezialisierung bedeutet dies, dass das Ergebnis erst anhand von Analysen berechnet werden muss. Dies ist kein grundsätzliches Hindernis, steigert aber den Aufwand.

Insbesondere anwendungsspezifische Implementierungen von Entscheidungsmechanismen, wie der Zusammenstellung von Initialisierungsfunktionen im LibrePilot unter Verwendung des Binders sind dabei Hindernisse für die Analyse, da es jeweils anwendungsspezifische Lösungen benötigt.

Eine mögliche Lösung wäre dazu die Erweiterung der Betriebssystemschnittstelle, sodass derartige Modularität ausdrückbar wird. Besser wäre jedoch eine Unterstützung der Programmiersprache, sodass die Intentionen in dieser direkt umgesetzt werden können.

Die Implementierung einer anwendungsspezifischen Lösung für die Initialisierungsfunktionen des LibrePilots habe es ermöglicht, dass die genannten 7 Fäden erkannt werden. Ohne diese Lösung wäre nur ein Faden erkannt worden.

6.4.3 Echte dynamische Entscheidungen

Neben den pseudodynamischen Entscheidungen existieren auch die echten dynamischen Entscheidungen, die auf Umweltzustände und Ereignisse reagieren. Diese erlauben es, dass sich eine Anwendung an ihre Umwelt anpasst.

Das bedeutet jedoch, dass hier grundsätzlich keine statische Spezialisierung möglich ist, da diese Laufzeitinformationen definitionsgemäß nicht zur Übersetzungszeit vorliegen. Das Problem daran ist, dass diese Entscheidungen Einfluss auf die Qualität der statischen Spezialisierung haben. Je nach Art und Umfang führen diese zu einer Verschlechterung der Möglichkeiten der statischen Spezialisierung, da pessimistische Annahmen getroffen werden müssen. Im schlechtesten Fall sind keine statischen Spezialisierungen möglich, sodass die verwendete Implementierung auf die vollständig dynamische Variante zurückgreift.

Sichtbar ist eine solche Entscheidung beispielsweise im LibrePilot (Zeile 40. Hier wird vor dem Erzeugen des GPS-Fadens auf das Vorhandensein der entsprechenden Hardware geprüft.

Wäre diese Prüfung nicht vorhanden, so wäre der Faden vollständig statisch instanziierbar und damit der Systemstart den Ergebnissen aus den Benchmarks (Abschnitt 4.4.1) entsprechend schneller. Bei Anwendung auf alle derartigen Entscheidungen würde insgesamt auch Speicher gespart werden. Gleichzeitig fällt auch die Unsicherheit weg, ob das System auch mit allen optionalen Komponenten funktioniert, da der Speicherbedarf dann statisch bestimmt würde. Im Istzustand wird dieses Problem erst zur Laufzeit festgestellt und erfordert dann entsprechende Behandlung.

Als Lösungsvorschlag könnten, wie in Abschnitt 6.3.1 beschrieben, alle Systemobjekte reserviert werden. Da die Freispeicherverwaltung mindestens dieselbe Menge Speicher vorhalten müsste, um alle Peripherie zeitgleich zuzulassen, wäre mindestens die Phase der Speicherallokation (Abschnitt 4.2.1) ohne erhöhten Speicherbedarf und andere Kosten möglich.

6.5 Übersetzer und Werkzeuge

Die für die Umsetzung der statischen Spezialisierung verwendeten Werkzeuge und die Ebene der Programmiersprache im Modell der hierarchischen virtuellen Maschinen haben ebenfalls einen Einfluss auf die Möglichkeiten der Spezialisierung und führen ihrerseits Hindernisse ein.

6.5.1 Wahl der richtigen Sprachebene

Die Implementierung einer Anwendung zur Lösung eines Problems wird regelhaft in einer Hochsprache durchgeführt. Die Hardware, die diese Anwendung ausführt, kann jedoch nur Maschineninstruktionen ausführen. Dementsprechend besteht hier eine semantische Lücke, die der Übersetzungsprozess schließen muss. Übersetzer wie die aus der LLVM-Werkzeugkette [LA04] führen diesen Prozess durch.

Das Problem ist die Wahl der richtigen Ebene für die statischen Analysen und Durchführung der Spezialisierung. Hierbei muss eine Abwägung geschehen zwischen dem vorhandenen Informationsgehalt und der Kompatibilität zu mehreren Hochsprachen.

Die Implementierung in ARA verwendet die LLVM-Zwischensprache als Grundlage für Analyse und Spezialisierung. Damit wird eine Kompatibilität mit allen Hochsprachen ermöglicht, für die es einen entsprechenden Übersetzer hin zu dieser Zwischensprache gibt. Während diese Zwischensprache zwar über ein Typsystem verfügt, so ist sie doch als maschinennahe Sprache konzipiert [LA04]. Dies hat zur Folge, dass das Typsystem weniger aussagekräftig ist und auch andere Auszeichnungsmöglichkeiten verschiedener Hochsprachen fehlen.

Im GPSLogger existieren unter anderem die Klassen `MutexLocker`, `DerivedGPSSatellitesData`, `GPSDataModel` und `GPSSatellitesData` (siehe Quellcode 6.2). Die Klasse `GPSDataModel` enthält ein Element vom Typ `GPSSatellitesData` (Zeile 42). Die Klasse `DerivedGPSSatellitesData` hingegen erbt von der Klasse `GPSSatellitesData` (Zeile 16). Im Zwischencode ist dies aber nicht mehr unterscheidbar (Zeilen 60 und 61).

Der Vorteil von LLVM ist, dass es eine Vielzahl von Programmiersprachen gibt, die auf dieses Werkzeug aufbauen. Damit bedeutet eine Arbeit auf LLVM-Code automatisch eine breite Sprachunterstützung. Um diese bei zu behalten, müsste die LLVM Zwischensprache derart erweitert werden, dass sie die Informationen, die in der Hochsprache vorhanden sind, nicht verwirft, sondern auch in der Zwischensprache zugänglich lässt. Dies würde jedoch für jede Sprache einen gesonderten Aufwand bedeuten, da genau dieser Teil der Übersetzerkette sprachspezifisch ist. Zusätzlich ist die Zwischensprache momentan nicht dafür ausgelegt, sprachspezifische Informationen dieser Art zu beinhalten, sodass auch eine Erweiterung der Sprache notwendig wäre. Tatsächlich wird im LLVM-Projekt gerade der gegenteilige Ansatz verfolgt, weniger Informationen zu speichern. Zeiger sind aktuell in den Versionen ab 14 nicht mehr typbehaftet [Llv].

Alternativ könnte die Spezialisierung so implementiert werden, dass sie direkt die Hochsprache als Eingabesprache verwendet. Bei diesem Ansatz wären keine Übersetzungsverluste vorhanden und die Analysen hätten das gesamte in der Hochsprache ausgedrückte Wissen über die Anwendung zur Verfügung. Jedoch müsste für diesen Ansatz für jede zu unterstützende Sprache entsprechende Unterstützung implementiert werden.

Grundsätzlich ist dieses Problem jedoch wie beschrieben lösbar und steht nicht im Widerspruch zu der prinzipiellen Anwendbarkeit der hier vorgestellten Spezialisierungen.

6.5.2 Typen und Auszeichnungen in LLVM

Für die Verarbeitung mittels ARA müssen alle Programmteile zusammengeführt werden. Die Änderungen durch das Zusammenführen sind in der Abbildung farblich markiert. LLVM verschmilzt dabei alle strukturäquivalenten Typen und ignoriert die in der Hochsprache zuvor geltende Namensäquivalenz. Dies zeigt sich darin, dass Typen nicht mehr existieren (Zeile 60) und Parameter und Rückgabewerte ihre Typen ändern (Zeilen 74 und 70). In der konkreten Anwendung sind so beispielsweise die Klassen zur Verwaltung der GPS-Daten und der Ansteuerung der SD Speicherkarte im GPSLogger nicht mehr unterscheidbar.

Dieser Informationsverlust führt dazu, dass beispielsweise die Wertanalyse und die Zeigeranalyse weniger präzisere Informationen liefern. Zeiger auf verschiedene Typen können so beispielsweise nicht mehr anhand des Zieltyps unterschieden werden.

Die Lösung für diese Probleme gleicht den des vorherigen Abschnitts. Entweder muss LLVM dahingehend erweitert werden, dass es Namensäquivalenz beachtet oder es muss direkt auf der Hochsprachenebene gearbeitet werden.

Für die konkrete Implementierung bedeutet dieses Problem eine Degradation des Ergebnisses der Spezialisierung. Als minimale Lösung wurde die verwendete LLVM-Implementierung so angepasst, dass sie auch Namensäquivalenz beachtet Dies funktioniert jedoch nur für LLVM bis Version 13 wegen der Typenlosen Zeiger ab den folgenden Versionen. Ohne diese Erweiterung wären beim GPSLogger die Fäden für das GPS-Modul und die Speicherkarte nicht spezialisierbar. Dementsprechend ist auch dieses Problem kein grundlegendes Hindernis für den Ansatz, sondern nur ein Problem der momentanen Implementierung.

6.6 Idiomatik

Die Kombination aus der verwendeten Programmiersprache und dem Betriebssystemmodell führt zu bestimmten Idiomen bei der Implementierung von Anwendungen. Aus diesen Idiomen heraus entstehen Probleme für die Analysierbarkeit der Anwendung. Sind diese regelhafter Natur, lassen sie sich teilweise mittels der automatischen Spezialisierung beheben, dies gelingt jedoch nicht in allen Fällen.

6.6.1 Manueller Nachbau von fehlender Sprachsemantik

Nicht jede Sprache verfügt über alle Mittel des Ausdrucks. Gleiches gilt für die Unterstützung der Betriebssystemschnittstelle für die Idiomatik der Sprache. Programmierer verwenden jedoch gerne gewohnte und effektive Mittel zur Implementierung ihrer Anwendungen. Dies führt dazu, dass fehlende Mittel mit den vorhandenen nachgebaut werden.

FreeRTOS ist in C implementiert, der GPSLogger hingegen in C++. In C++ ist RAII das Mittel der Wahl für eine Verwendung eines Betriebsmittels [Str97, Kapitel 14.4], die FreeRTOS Schnittstelle unterstützt dies jedoch nicht. Daher wurde eine entsprechende Funktionalität nachgebaut (Zeile 1). Dies hat zur Folge, dass die zu spezialisierende Aufrufstelle nicht mehr direkt in dem Anwendungscode an der Verwendungsstelle der Ressource ist, sondern in der umgebenden RAII-Implementierung.

Die Sprache C unterstützt das objektorientierte Programmierparadigma nicht nativ. Der LibrePilot ist jedoch in C implementiert und verwendet zur Initialisierung der einzelnen Module das Konzept des Konstruktors. Zu diesem Zweck ist dieses mittels speziell benannter Sektionen mit Funktionszeigern (Zeile 11) die beim Binden zusammengetragen werden (Zeile 67) implementiert. Für die Analyse ist hieraus jedoch nicht die Semantik dieser Liste von Funktionszeigern ersichtlich, sodass aus der Implementierung nicht hervorgeht, dass jede Funktion genau einmal im Rahmen der Schleife in Zeile 50 beziehungsweise Zeile 50 aufgerufen wird. Vielmehr muss die Analyse davon ausgehen, dass an diesen Stellen beliebige Funktionen beliebig aufgerufen werden, sodass kein handhabbarer und aussagekräftiger Aufrufgraph erstellt werden kann und somit die gesamte Analyse unmöglich wird.

Außerdem verfügt C über keine Auszeichnungsmöglichkeit für regulären und ausnahmebehandelnden Kontrollfluss. Der LibrePilot verwendet jedoch Zusicherungen (Zeile 30), die zur Laufzeit eine Ausnahmebehandelnden Kontrollfluss hervorrufen können. Für die Analyse ist dieser jedoch nicht unterscheidbar, sodass für die Instanziierung in der folgenden Zeile davon ausgegangen werden muss, dass diese nur bedingt erreichbar ist. Unter der Annahme, dass die Spezialisierung den regulären Fall abdecken soll und zum anderen sie garantieren kann, dass das abgeprüfte Systemobjekt von der Spezialisierung sicher instanziiert wird, ist jedoch auch die Instanziierung des Fadens nach der Zusicherung sicher.

Da bei allen Problemen zu spezialisierende Systemaufrufe mit betroffen sind, ist dies ein Problem für die Spezialisierung. Dieses ist jedoch ein technisches Problem, zu dem es mehrere Lösungswege

gibt, sodass die Spezialisierung umsetzbar wird. Für diese Arbeit wurde der Weg gegangen, dass anwendungsangepasste Zusatzinformationen hinzugefügt wurden, die der Analyse in genau diesen Situationen die richtige Semantik beschreiben. Diese Lösung ist jedoch nicht generisch anwendbar, sondern auf das konkrete Problem maßgeschneidert, sodass für eine breitere Anwendung von ARA eine verbesserte Lösung notwendig sein wird.

Alternativ kann die Analyse um einen Katalog bekannter Idiomatiken erweitert werden, sodass diese automatisch mit erkannt werden. Die im LibrePilot verwendete Behandlung der Zusicherung besteht beispielsweise aus einer leeren Endlosschleife. Eine solche kann nicht zum Fortschritt der Anwendung beitragen, eine Markierung einer solchen als irregulären Kontrollfluss wäre folglich systematisch möglich. Jedoch bleiben diese Verfahren sehr problem- und implementierungsspezifisch.

Grundsätzliche Abhilfe schafft nur der Wechsel der verwendeten Programmiersprache für die Anwendung und die Betriebssystemschnittstelle, sodass die direkte Unterstützung der Sprache für alle gewünschten Idiome vorhanden ist.

Die Anwendung der anwendungsspezifischen Lösungen hat es ermöglicht, dass beim GPSLogger die Interaktionen den jeweiligen Systemobjekten zugeordnet werden konnten. Ohne diese wäre das RAII Muster nicht erkennbar gewesen und somit wäre nicht eindeutig gewesen, welcher Semaphor Ziel der Operation ist. Beim LibrePilot haben die Ergänzungen dazu geführt, dass überhaupt statisch spezialisierbare Systemobjekte erkannt wurden, da ohne diese kein einziges auf einem sicheren Pfad gelegen hätte. Somit haben diese Maßnahmen wesentlich zur Anwendbarkeit der Spezialisierungen beigetragen.

6.6.2 Vorauseilende Änderungsvorsorge

Wiederverwendbarkeit, Lesbarkeit und Ordnung sind wichtige Faktoren für das Gelingen von Softwareentwicklung [NR69]. Ein systematisches Herangehen nach Ingenieursprinzipien ist dabei essenziell[Sch12]. Entwurfsmuster sind die Grundlage für systematische Softwareentwicklung [Gam+95]. Deren weite Verbreitung sorgt sowohl für eine Strukturierung des Programmcodes als auch für eine geringere kognitive Last beim Verstehen des Codes, sobald ein solches Muster wiedererkannt wird.

Sowohl die Verwendung von Entwurfsmustern wie beispielsweise dem des Einzelstücks als auch allgemein die Verwendung von virtuellen Methoden im objektorientierten Programmierparadigma führen zu dem Zustand, dass vermehrt Indirektionen über Zeiger im Programmcode vorhanden sind. Virtuelle Methoden werden stets zur Laufzeit mit dazugehöriger Tabelle aufgelöst und ermöglichen so Laufzeitpolymorphismus. Dadurch ist jedoch nicht automatisch und unmittelbar ein Mehrwert gegeben, wenn für den momentan vorliegenden Anwendungsfall auch mit globalen Variablen und ohne Laufzeitpolymorphismus dasselbe Ergebnis erzielbar wäre. Sie werden jedoch trotzdem verwendet, um vorbeugend auf bisher nicht absehbare zukünftige Änderungen in den Anforderungen besser vorbereitet zu sein.

Bei Programmiersprachen wie C++, die keine Aussagen über das Aliasproblem von Zeigern geben, sind sie ein Hindernis für die statische Analyse und damit die Betriebssystemspezialisierung, da nur mit Heuristiken oder dynamischen Analysen das Erkennen von Entwurfsmustern möglich ist [Pet05; Liu21]. Auch die Durchführbarkeit und die Qualität der Ergebnisse anderer statischer Analysen wie Antwortzeitanalyse [Wä+18] und Speicherbedarfsanalyse [DL18] leiden unter der Anwesenheit von Zeigern und Indirektionen. Daher empfehlen die einschlägigen Standards insbesondere für sicherheitskritische Systeme diese nicht zu verwenden [ISO11; Mis].

Für die hier vorgestellten Spezialisierungen bedeutet dies eine graduelle Verschlechterung der Spezialisierbarkeit, jedoch kein grundsätzliches Hindernis. Sind die Indirektionen grundsätzlich statisch berechenbar, so stellen sie kein Hindernis da.

Als Lösungsmöglichkeit wäre hier die Wahl einer Sprache möglich, die entsprechende Zusagen über Zeiger treffen kann. Rust [KN18] setzt strenge Regeln eines Eigentumsmodells durch, sodass Entwurfsmuster wie das Einzelstück explizit mit der Sprache ausgedrückt werden und dadurch analysierbar sind. C# [Ecm] bietet ebenfalls Sprachunterstützung für Entwurfsmuster, sodass diese analysierbar werden.

Ein Wechsel der Sprache führt zwar im ersten Moment zu erhöhten Kosten, da die Anwendung und eventuell benötigte Bibliotheken erneut implementiert werden müssen, zugleich profitiert jedoch auch die restliche Anwendungslogik von den besseren Sprachunterstützungen. Die Verwendung von Rust als Programmiersprache verhindert beispielsweise vollständig die Fehlerklasse der Speicherüberläufe, sodass die Implementierungen robuster gegenüber Angriffen werden.

6.7 Programmiersprache

Das Problem, welches von einem eingebetteten System gelöst werden soll, ist in einer domänenspezifischen Sprache definiert. Die Implementierung einer Anwendung zur Lösung eines Problems wird regelhaft in einer Hochsprache durchgeführt. Dementsprechend besteht hier eine semantische Lücke, die der Übersetzungsprozess schließen muss.

Die Übersetzung von der domänenspezifischen Problemdefinition wird von Menschen vorgenommen und läuft somit nicht deterministisch ab [Die19]. Bereits auf diesem Weg gehen Informationen verloren. Daher ist es wichtig, die richtige Sprache zu wählen, um das vorhandene Wissen entsprechend abbilden zu können.

6.7.1 Garantien und Auszeichnungskraft

Die wichtigste Grundlage für die statische Spezialisierung ist die Möglichkeit, die Kontroll- und Werteflüsse der Anwendung statisch analysieren zu können. Dies ist nur möglich, wenn die verwendete Programmiersprache diese genügend auszeichnet. Für reguläre Kontrollflüsse trifft dies auf alle Hochsprachen zu, sie verfügen über Sprachmittel für Schleifen, Verzweigungen und Unterprogrammaufrufe. Anders ist die Situation für ausnahmebehandelnde Kontrollflüsse, diese werden nicht von allen Hochsprachen unterstützt. Gleiches gilt für Werteflüsse, auch hier sind die Garantien der verschiedenen Sprachen sehr unterschiedlich.

Dies führt aber bei der statischen Analyse zu dem Problem, dass Ausnahmebehandlungen nicht vom regulären Programmablauf unterscheidbar sind. Pfade zu Funktionsaufrufen und hier besonders die zu Systemaufrufen sind dann im ICFG nicht mehr eindeutig. Die Alias-Analyse für Zeiger ist eine Herausforderung für die statische Analyse zum Nachverfolgen der Interaktionen.

Die hier vorhandenen Anwendungen und auch mehr als 77 % der momentanen Echtweltsysteme verwenden C/C++ als Implementierungssprache [EET19]. Diesen Sprachen fehlt es jedoch an Ausdruckskraft und Garantien. C verfügt über keinerlei Unterscheidung von Kontrollflüssen, während C++ zumindest das Konzept der Ausnahme kennt. Beide Sprachen geben jedoch keine Garantien bezüglich der Alias-Analyse von Zeigern.

Die in ARA verwendete SVF [SX16] implementiert für die Alias-Analyse Andersens Algorithmus [And94] und ist damit in der Lage, für einige Fälle Ergebnisse zu liefern. Andersens Algorithmus arbeitet dabei mit Teilmengenbedingungen. Im Gegensatz dazu arbeitet Steensgards Algorithmus mit Gleichheitsbedingungen. Dies ermöglicht eine besonders effiziente Ausführung mit annähernd linearem Zeit- und Speicherbedarf bei Darstellung als Union-Find-Datenstruktur [Ste96]. Im Allgemeinen ist die Alias-Analyse aber ein schwieriges Problem, welches bereits in anderen Arbeiten untersucht wurde. Für Aufrufgraphen in C-Programmen mit Funktionszeigern gibt es sowohl Untersuchungen

der Präzision dieser Analysen [MRR04], deren Ausführungsgeschwindigkeit [SH97], als auch über den Einfluss von Funktionszeigern auf die Erstellung der Aufrufgraphen [ACT99] und den Vergleich verschiedener Extraktionsmethoden [Mur+98]. Bei Sprachen wie C/C++, die keine Garantien für die Zeiger geben, ist die Alias-Analyse jedoch ungeachtet dessen ein schwieriges Problem, welches nie statisch berechenbar sein wird [Hin01].

Dieses Problem zeigt sich symptomatisch bei der Sprache, die für die Anwendung verwendet wird. Ihre Ursache liegt aber bereits in der Wahl der Sprache, die die Betriebssystemschnittstelle bereitstellt. Bereits hier sind Ausnahme und Regelfall nicht sprachlich unterscheidbar, sondern nur idiomatisch anhand eines Rückgabewertes des Systemaufrufs. Es gibt auch keine Garantien über Zeiger und deren Ziele oder Besitz.

Dieses Problem ist nur vollständig lösbar, indem bereits die Betriebssystemschnittstelle in einer Sprache implementiert wird, deren Semantik ausdrucksstark genug ist, Kontrollflüsse und Zeiger entsprechend zu beschreiben. Eine mögliche Lösung dafür wäre die Sprache Rust, welche sowohl Sprachsemantik für Ausnahmekontrollflüsse bereitstellt als auch starke Garantien über Zeiger sicherstellt.

Für meine Arbeit bedeutet dies, dass ich nur anhand von Mustererkennung Fehlerbehandlungsidiome zu identifizieren, um diese entsprechend bei der Kontrollflussanalyse behandeln zu können. Ohne diese Mustererkennung hätte die Analyse des LibrePilots nicht eine statisch spezialisierbare Instanz ergeben. Wäre der LibrePilot in Rust implementiert, so wären alle Systemobjekte bekannt, wenn auch nicht alle statisch instanziierbar, da die Abhängigkeiten von vorhandener Hardware und Konfiguration weiterhin verbleiben. Zudem wären die Interaktionen zumindest teilweise zuordenbar, sodass auch hier statische Spezialisierungen möglich wären.

Die fehlenden Garantien und semantischen Mittel führen dazu, dass die Qualität der statischen Spezialisierung nicht ihr Optimum erreicht. Sie führen dazu, dass nur ein Teil des theoretisch spezialisierbaren Umfangs umsetzbar ist. Werden diese jedoch ergänzt, so ist mein hier vorgestelltes Verfahren auch auf die übrigen Teile anwendbar.

6.7.2 Unterspezifizierte Semantik

Eine der wichtigsten Eigenschaften der hier vorgestellten Spezialisierungen ist die Erhaltung der Semantik. Das bedeutet, wie in Abschnitt 2.2 definiert, das Systemverhalten durch die Spezialisierung nicht ändert.

Um dies sicherzustellen, muss während der Analyse das erwartete Systemverhalten ermittelbar sein. Dies ist aber problematisch, wenn die Semantik der verwendeten Programmiersprache hier nicht ausreichend spezifiziert ist. In C++ ist beispielsweise der Zeitpunkt, zu dem ein Konstruktor einer statischen globalen Variable ausgeführt wird, nicht genau festgelegt, sondern implementierungsabhängig. Entsprechend der Spezifikation der Sprache [Com+11] wird nur sichergestellt, dass dieser vor der Verwendung einer anderen Funktion oder einer anderen globalen Variable aus derselben Übersetzungseinheit aufgerufen wird. Auch ist der aufrufende Faden nicht festgelegt.

Für die Umsetzung der Spezialisierung bedeutet dies, dass neben der eigentlichen Anwendung und dem Betriebssystemmodell zusätzlich auch implizit die konkrete Implementierung der Laufzeitumgebung der verwendeten Programmiersprache Teil der Eingaben der Anwendung wird, um dasselbe beobachtbare Verhalten sicherzustellen.

Beim GPSLogger trifft dies auf den instanziierenden Systemaufruf in Zeile 47 zu. Für diese Arbeit war dies jedoch kein Problem, da ausschließlich Systemobjekte in Konstruktoren instanziiert wurden, die keine Registrierung im Betriebssystem (Abschnitt 4.3.4) benötigen. Die Auswirkungen der anderen Phasen könnten ungeachtet des Ausführungszeitpunktes unverändert umgesetzt werden.

Problematisch wäre dies, wenn ein Systemobjekt, wie z. B. ein Faden, der im Betriebssystem registriert werden müsste, in einem Konstruktor instanziiert würde. Um auch diesen Fall zu lösen, müsste auch das Verhalten der im Original verwendete Laufzeitbibliothek mit als Eingabe verwendet werden.

6.7.3 Entscheidung der Programmiersprache

Aus den vorherigen Abschnitten geht hervor, dass die Wahl der Programmiersprache einen wesentlichen Einfluss auf die Umsetzbarkeit der statischen Spezialisierung hat. Die Programmiersprache ist implizit auf allen Ebenen der Problemursachen mit einbezogen. Dementsprechend sollte diese Wahl nach technischen Gesichtspunkten eigentlich nicht C und C++ die auszuwählenden Sprachen für eingebettete Systeme sein sollte. Eine besser geeignete Sprache wäre Rust, da diese wie beschrieben Probleme auf mehreren Ebenen löst.

Eine Untersuchung zeigt jedoch, dass C und C++ die vorherrschenden Sprachen für die Entwicklung derartiger Systeme sind [EET19]. Gleichwohl ist aber nach selber Studie das Beherrschen der Komplexität und Menge des Codes die zweitgrößte Herausforderung direkt nach der Integration neuer Technologien. Die Verwendung von anderen Programmierparadigmen und mächtigeren Sprachkonstrukten helfen dabei, diesen Problemen zu begegnen. Da der Sprachwechsel aber aufgrund einer hohen Wiederverwendungsrate von 81 % aus vorherigen Projekten gescheut wird, resultiert dies in der vorgefundenen Situation, dass fehlende Sprachmittel manuell ohne Sprachunterstützung ergänzt werden. Dies führt zu den in den vorherigen Abschnitten aufgeführten Problemen.

Als grundsätzliche Lösung ist hier nur der Wechsel zu einer geeigneteren Programmiersprache vorhanden. Für die Umsetzung in dieser Arbeit habe ich problemspezifische Heuristiken in die Implementierung eingefügt, sodass die Spezialisierungen wie beschrieben durchführbar sind, dies ist jedoch kein allgemein funktionierender Ansatz, sodass ein Sprachwechsel zu präferieren ist.

6.8 Zusammenfassung

In diesem Kapitel habe ich die Probleme aufgeführt und eingeordnet, die der momentanen Implementierung sowie der grundsätzlichen Anwendbarkeit des Ansatzes entgegenstehen. Diese entspringen verschiedenen Ebenen, angefangen beim Design der Betriebssystemschnittstelle und der daraus resultierenden Verwendung über die verwendeten Werkzeuge bis hin zu der verwendeten Programmiersprache und der entstehenden Idiomatik. Dabei konnte ich zeigen, dass die meisten Hindernisse nur die momentane Umsetzung technisch hindern. Nur wenige Probleme sind fundamentale Probleme, die dem Ansatz entgegenstehen.

Forschungsfrage

Damit kann ich nun auch die dritte Forschungsfrage (FF3) nach den grundsätzlichen Hindernissen, die statischer Spezialisierung, wie sie in den vorherigen Kapiteln beschrieben ist, entgegenstehen, beantworten. Grundlegend sind alle Hindernisse zurückzuführen auf ein Defizit in den zur Analysezeit vorliegenden Informationen. Sie unterscheiden sich jedoch hinsichtlich der Ursache dieses Defizits und in den damit verbundenen Lösungsmöglichkeiten. Zudem unterscheiden sie sich in technisch lösbare Probleme, die die momentane Implementierung des Ansatzes an der vollen Ausschöpfung seines Potenzials hindern und solchen, die nicht technisch lösbar sind.

Tabelle 6.1 zeigt eine Zusammenfassung der zuvor vorgestellten Hindernisse. Sie sind dabei nach ihrer technischen Lösbarkeit gruppiert. Dabei ist zu erkennen, dass nicht technisch lösbaren Probleme genau diejenigen sind, die echte dynamische Laufzeitentscheidungen beinhalten. Die technisch lösbaren Probleme sind diejenigen, die aus den verwendeten Werkzeugen, Sprachen und

Nicht allgemein technisch lösbar
OS Design: **Beliebigkeit der Instanziierung und Interaktionspartner**
OS Design: **Pessimistische Grundannahmen**
OS Design: **Einfachheit der Schnittstelle**
OS Verwendung: **Wahl des Betriebssystems**
OS Verwendung: **Echte dynamische Entscheidungen**
Sprache: **Auswahl der Programmiersprache**
Technisch lösbar
OS Verwendung: **Pseudodynamische Entscheidungen**
Werkzeuge: **Wahl der Sprachebene**
Werkzeuge: **Typen und Auszeichnungen in LLVM**
Idiomatik: **Manueller Nachbau von Sprachsemantik**
Idiomatik: **Vorauseilende Änderungsvorsorge**
Sprache: **Garantien und Ausdruckskraft**
Sprache: **Unterspezifizierte Semantik**

Tabelle 6.1 – Hindernisse der statischen Spezialisierung und deren Ursachen.

Implementierungsentscheidungen der Anwendung und der Spezialisierung hervorgehen. Aufgrund dieser Lösbarkeit stehen sie dem von mir vorgestellten Spezialisierungsverfahren nicht grundsätzlich nicht entgegen. Damit kann ich zeigen, dass der Ansatz nicht nur, wie in den beiden vorherigen Fragen beantwortet, lohnenswert und durchführbar ist, sondern auch wenige nicht lösbare Hindernisse hat.

Dabei ist festzuhalten, dass jedes auftretende Problem den Ansatz nicht direkt unanwendbar macht, sondern lediglich das Optimierungspotenzial etwas verringert. Anhand der Evaluationen der Fallbeispiele zeigt sich jedoch, dass die Hindernisse nur teilweise die Spezialisierung beeinflussen, sodass diese in den anderen Bereichen der Anwendung technisch machbar ist. Dies steht im Kontrast zu bestehenden Arbeiten statischer Spezialisierung, die stets eine vollständige Analysierbarkeit vorausgesetzt haben, um anwendbar zu sein [Die19; Hof16; Hof14; Sch11].

Das größte Potenzial für eine bessere Anwendbarkeit liegt in der Empfehlung an Anwendungsentwickler, eine andere Programmiersprache als C/C++ zu wählen. Bei der Wahl von Rust als Sprache verbessert sich nicht nur die Spezialisierbarkeit, sondern auch die Anfälligkeit für andere Fehlerklassen wie Speicherüberläufe [Uc17; Get16; Rad+22].

Die nicht lösbaren Hindernisse sind direkt mit der charakterisierenden Eigenschaft der dynamisch konfigurierten Systeme verbunden: Es können zur Laufzeit beliebige Systemobjekte instanziiert werden, die beliebig interagieren dürfen. Aufgrund des Anwendungsfeldes der eingebetteten Systeme, bei denen jedoch die auszuführende Aufgabe durch das umliegende Gesamtsystem bestimmt ist, ist auch davon auszugehen, dass die Mehrzahl der Systemobjekte und Interaktionen konzeptionell statisch festgelegt ist. Dies bedeutet eine konzeptionelle Anwendbarkeit des Ansatzes, dessen Qualität durch technische Verbesserungen noch steigerbar ist. Die Fallstudien bestätigen dieses Bild, indem bereits mit der bestehenden Implementierung und manueller Betrachtung festgestellt werden kann, dass ein Großteil der Systeme spezialisierbar ist.

7

Schluss

Zusammenfassung, Ergebnisse und Ausblick

In dieser Arbeit habe ich das Themenfeld der automatischen Spezialisierung von vormals dynamischen Systemaufrufen eingebetteter Echtzeitsysteme betrachtet. Das Gesamtziel dieser Arbeit war es, die Möglichkeiten der anwendungsspezifischen Spezialisierung zu ermitteln. Ziel war es dabei, die nichtfunktionalen Eigenschaften des resultierenden Gesamtsystems positiv zu beeinflussen. Außerdem habe ich untersucht, welche Hindernisse eben dieser statischen Spezialisierung grundlegend entgegenstehen. Insgesamt komme ich dabei zu dem Ergebnis, dass eine solche automatische anwendungsgewahre Spezialisierung nutzbringend eingesetzt werden kann.

7.1 Zusammenfassung der Arbeit

Die Motivation dieser Arbeit entspringt aus dem Problem, dass Betriebssysteme im Regelfall einen deutlich größeren Funktionsumfang liefern, als die konkret darauf auszuführende Anwendung benötigt. Während dieser Umstand bei Gemeinzwecksystemen wünschenswert ist, um auf sich ändernde Anforderungen und neu auszuführende Anwendungen vorbereitet zu sein, ist dies gleichzeitig ein Problem für eingebettete Systeme. Diese sind im Regelfall ressourcenbeschränkt, sodass unnötig verwendete Ressourcen diese verteuern. Gleichzeitig ist die auszuführende Anwendung jedoch im Voraus durch das umgebende Gesamtsystem festgelegt. Da aber eine manuelle Anpassung der Betriebssystemimplementierung zu aufwendig und fehleranfällig ist, eröffnet sich hier das Feld für eine automatische anwendungsgewahre Spezialisierung des Betriebssystems, welches in dieser Arbeit betrachtet wurde.

Grundlage war die automatische Gewinnung von Wissen über die Verwendung des Betriebssystems durch die Anwendung. Bestehende Arbeiten betrachten dabei statisch konfigurierte Systeme [DL17; Wä+18] und dringen dabei wie diese Arbeit bis auf Interaktionsebene vor. Für dynamische Systeme existieren Arbeiten, die entweder statisch, aber nur auf Abstraktionsebene arbeiten [Zie+19; RHL14] oder sie arbeiten zur Laufzeit des Systems [EKO95; LP09; Mur+19]. In dieser Arbeit hingegen analysiere und spezialisiere ich bereits zur Übersetzungszeit bis auf Ebene der Interaktionen und erweitere damit den Kenntnisstand der Wissenschaft. Zu diesem Zweck habe ich die statische Instanz- und Interaktionsanalyse vorgestellt. Diese ermittelt auf Basis des Anwendungsprogrammcodes die Systemobjektinstanzen und deren Verwendung durch die Anwendungslogik. Als Ergebnis entsteht der Instanz- und Interaktionsgraph. In diesem sind alle Instanzen als Knoten dargestellt und anhand ihrer Eigenschaften attribuiert. Die Kanten zwischen diesen Knoten repräsentieren die möglichen Interaktionen, die diese Systemobjekte im Rahmen der gegebenen Anwendung ausführen können.

Als erste Spezialisierung habe ich die statische Instanziierung der Systemobjekte vorgestellt. Während im Normalfall die Systemobjekte dynamisch zur Laufzeit instanziiert und initialisiert werden, verlagert diese Spezialisierung genau diesen Prozess in die Übersetzungszeit. Als Resultat entstehen im Voraus bekannte Systemobjekte. Dies führt zur Reduktion des benötigten Programmspeichers von bis zu 2 % unter anderem durch wegfallende Initialisierungsfunktionen. Zeitgleich reduziert sich auch die Startzeit des Gesamtsystems um bis zu 67 %. Insbesondere bei sicherheitskritischen Anwendungen, deren Startzeit auf dem kritischen Pfad liegt, verbessert sich so die Antwortzeit und deren Schwankungsbreite. Aber auch für die allgemeine Benutzbarkeit ist diese Startzeitverringerung ein positiver Beitrag zum Nutzungserlebnis.

Die zweite Klasse von Spezialisierungen zielt auf die Arbeitsphase der Anwendung ab. Hier habe ich die Spezialisierung von Systemaufrufen zur Kommunikation zwischen Komponenten der Anwendung vorgestellt. Am Beispiel von Nachrichtenwarteschlangen konnte ich zeigen, dass eine dem Benutzungsmuster angepasste Implementierung auch hier für eine Verbesserung der Laufzeit sorgen kann. Hierbei sind Reduktionen der Laufzeit von bis zu 43 % pro Systemaufruf möglich. Dies resultiert aus der Vermeidung von im Spezialfall nicht benötigter Synchronisation, die aber für den allgemeinen Fall notwendig ist. Von dieser Spezialisierung profitiert die Antwortzeit einer jeden Anwendung, die nicht die vollständige Vielfalt der Zugriffsmuster verwendet, sondern nur einem wohl definierten Muster folgt.

Abschließend habe ich betrachtet, welche Hindernisse einer statischen Spezialisierung grundsätzlich entgegenstehen. Ergebnis dieser Analyse ist, dass es wenige grundsätzliche Hindernisse gibt. Diese beschränken sich auf Entscheidungen, die echt dynamisch auftretendes Anwendungswissen benötigen. Dies sind jedoch diejenigen Entscheidungen, die einen echten Mehrwert der dynamisch konfigurierten gegenüber den statisch konfigurierten Systemen bieten, da Letztere diese nicht unter-

stützen. Alle anderen Hindernisse sind auf zu frühen Informationsverlust im Implementierungs- und Übersetzungsprozess vom mentalen Modell hin in ausführbaren Maschinencode zurückzuführen. Dementsprechend wäre es technisch möglich, diesem zu frühen Informationsverlust zu begegnen, indem die Information bis zur Spezialisierung nicht verworfen wird. Dies bedarf jedoch zumeist besserer Werkzeuge und insbesondere auch Programmiersprachen, die genügend Ausdruckskraft bieten, sodass das mentale Modell vollständig abgebildet werden kann, ohne schon in diesem frühen Schritt die entscheidenden Informationen zu verlieren.

Insgesamt zeigt sich jedoch, dass jedes der gefundenen Hindernisse nur zu einer partiellen Degradierung des Spezialisierungserfolgs führt. Dies steht im Kontrast zu den bestehenden Arbeiten zur statischen Spezialisierung auf Interaktionsebene [Die19; Hof16].

7.2 Forschungsfragen

Abschließend möchte ich die eingangs aufgestellten Forschungsfragen zusammenfassend beantworten. Nachdem ich die einzelnen Teilaspekte in den vorangegangenen Kapiteln beleuchtet habe, bin ich nun in der Lage, eben diese Fragestellungen hier mit meinen Erkenntnissen zu beantworten.

FF1 *Lässt sich die Flexibilität eines dynamischen Systemmodells mit den Laufzeitvorteilen eines statischen Systemmodells kombinieren, sodass die Flexibilität falls notwendig erhalten bleibt, ansonsten aber die Laufzeitvorteile des statischen Systemmodells ausgenutzt werden können?*

Grundsätzlich kann ich diese Frage positiv beantworten. Genau diese Ausnutzung eines statischen Systemmodells für diejenigen Systemaufrufe, die spezialisierbar sind, ist möglich, während alle anderen unverändert alle Eigenschaften wie bisher zeigen. Konkret konnte ich dazu in Kapitel 4 zeigen, dass sich die Systemobjekte, deren Eigenschaften aus der statischen Systemanalyse eindeutig hervorgehen, bereits statisch instanziieren lassen. Gleichzeitig sind alle Systemobjekte, über die nicht genug Information mittels statischer Analyse ermittelt werden kann, unverändert geblieben. Das gleiche Bild zeichnet sich auch in Kapitel 5 in Bezug auf die Interaktionen zwischen den Systemobjekten. Ist genügend Information bezüglich deren Verwendung vorhanden, wird eine entsprechende Spezialisierung durchgeführt. Fehlen Informationen oder ist die Spezialisierung im Kontext der konkreten Anwendung nicht lohnenswert, so wird auf die generische Implementierung zurückgegriffen. Dies ermöglicht einen fließenden Übergang zwischen keiner Spezialisierung und einer vollständigen Spezialisierung auf Grundlage der ermittelbaren Informationen und einer daraus abgeleiteten Abschätzung der möglichen Laufzeitvorteile. Ein solches hybrides System, bestehend aus spezialisierten und nicht spezialisierten Bereichen ist weiterhin uneingeschränkt arbeitsfähig und erfüllt aus Sicht der auszuführenden Anwendung uneingeschränkt die funktionalen Spezifikationen, die von dieser erwartet werden.

Dies ist ein grundsätzlicher Unterschied zu den bestehenden Arbeiten statischer anwendungsspezifischer Betriebssystemspezialisierung auf Instanz- und Interaktionsebene [Die19; Hof16; Hof14; Sch11]. Diese setzen bisher Wissen über das Gesamtsystem und alle darin vorhandenen Instanzen und Interaktionen voraus. Ohne diese vollständige Gesamtsicht sind sie nicht anwendbar. Dies gilt auch für die Aktualisierbarkeit der Anwendung, da auch hier mit meiner Arbeit eine Änderung der Teilinformationen nicht zu einem zwangsläufigen Austausch der gesamten Implementierung von Anwendung und Betriebssystem führt, sondern nur die betroffenen Teile geändert werden müssen.

Mit den hier von mir vorgestellten statischen Spezialisierungen bin ich in der Lage, das vorhandene Wissen weitestmöglich auszunutzen und fehlende Information zu tolerieren. Mit dieser Arbeit bin ich in der Lage, nach Bedarf einen fließenden Übergang zwischen den beiden Konfigurationsparadigmen zu ermöglichen. Es sind dadurch im resultierenden System die Vorteile aus

beiden Paradigmen verfügbar. Die Anteile, die statisch spezialisierbar sind, profitieren von den Vorteilen eines statisch konfigurierten Systems wie Laufzeit [Hof14], Fehlertoleranz [Hof+15] und Speicherbedarf [DL18]. Diejenigen Bestandteile, die die dynamische Konfigurierbarkeit ausnutzen, haben diese jedoch weiterhin uneingeschränkt zur Verfügung.

FF2 *Wie ändern sich der Laufzeitaufwand und der Speicherbedarf durch die Verringerung des Verwaltungsaufwands zur Laufzeit durch die statische Vorausberechnung von Systemaufrufen in dynamischen Betriebssystemmodellen?*

Betrachten wir zunächst den Laufzeitaufwand. Die Spezialisierungen können deutliche Reduktionen des Laufzeitaufwandes während des Systemstarts (bis zu 67 % in Mikrobenchmarks) und während der Arbeitsphase (pro Systemaufruf bis zu 43 %) erzielt werden. Es gibt zwar auch Spezialisierungen, die nicht zu einer Reduktion der Laufzeit führen, dieser Umstand kann aber anhand der hier gewonnenen Erkenntnisse für zukünftige Spezialisierungen als Metrik verwendet werden, sodass die automatisierte Spezialisierung nur lohnenswerte Spezialisierungen ausführt. Dadurch ist bei geeigneter Wahl der Spezialisierung stets eine Reduktion der Laufzeit erzielbar.

In Bezug auf den Speicherbedarf der Spezialisierung ist das Ergebnis, dass dieser teilweise einer Abwägung zwischen Laufzeit- und Vorhersagbarkeit auf der einen Seite und Speicherbedarf auf der anderen Seite unterliegt. Während für die statische Initialisierung in Kapitel 4 im Regelfall die dynamischen Initialisierungsfunktionen entfallen und so in der Bilanz eine Reduktion des Speicherbedarfs zu erwarten ist, verhält sich dies bei den Interaktionen in Kapitel 5 anders. Für die verschiedenen Implementierungen derselben Interaktion wird der dazugehörige Programmcode, falls nicht alle Interaktionen dieselbe Spezialisierung verwenden, in jeder verwendeten Variante benötigt. Dies führt zu einer Vergrößerung des Programmabbildes. Hier bedarf es dementsprechend einer anwendungsspezifischen Metrik, die bewertet, ob eine jeweilige Spezialisierung jeweils lohnenswert ist.

Dies entspricht dem hybriden Modell mit den Varianten der dynamischen und statischen Welt. Gleichzeitig bedeutet dies aber auch, dass bei steigender Spezialisierbarkeit die Notwendigkeit für mehrere Varianten der Implementierungen sinkt. Im Fall einer gutmütigen Anwendung führt mein Ansatz somit auch zu einer Reduktion des Speicherbedarfs.

FF3 *Welche Hindernisse stehen der statischen Vorausberechnungen kategorisch entgegen und wie kann damit umgegangen werden?*

Mit den Erkenntnissen aus den Kapiteln 4 und 5 sowie der weiteren Auseinandersetzung mit dieser Fragestellung in Kapitel 6 kann ich nun auch diese Frage beantworten. Grundsätzlich teilen sich die Hindernisse in zwei Kategorien.

In die erste fallen diejenigen Entscheidungen, die eine echt dynamische Laufzeitentscheidung voraussetzen. Diese sind kategorische Ausschlusskriterien und stehen der statischen Spezialisierung grundsätzlich entgegen. Diese sind aber auch zeitgleich die charakterisierende Eigenschaft des dynamischen Systemmodells und damit entscheidender Vorteil gegenüber dem statischen Systemmodell.

In die zweite Kategorie fallen diejenigen Hindernisse, die aufgrund von zu frühem Informationsversverlust existieren. Hierbei handelt es sich sowohl um Ursachen aus der Verwendung der Betriebssystemschnittstelle als auch der Idiomatik, die aus Programmiersprache und Betriebssystemschnittstelle entsteht. In die zweite Kategorie fallen diejenigen Programmiermuster und Sprachkonstrukte, die für die eigentliche Implementierung unnötige Flexibilität in den Programmcode einführen. Dies sind beispielsweise das objektorientierte Programmierparadigma oder auch Ausnahmebehandlungen und der Einsatz von Entwurfsmustern jeweils ohne syntaktische und semantische Unterstützung der verwendeten Programmiersprache. Die für mehr als 70 % der Projekte verwendeten Programmiersprachen sind C und C++, diesen fehlen jedoch für die Analysierbarkeit wichtige

Auszeichnungsmöglichkeiten und Zusicherungen über Kontroll- und Datenflüsse. Die vollständige Lösung für diese Probleme wäre ein Wechsel hin zu einer Sprache, die die Verwendung der gewünschten Paradigmen mit syntaktischer und semantischer Unterstützung anbietet. Ohne diese führt deren Verwendung zu Unsicherheiten der statischen Analyseergebnisse, denen ich mit Heuristiken begegnet bin.

Dazu kommen Verluste aufgrund der eingesetzten Werkzeuge und der Wahl der Ebene der Programmiersprache, auf der die Werkzeuge arbeiten. Für diese Arbeit habe ich die LLVM-Zwischensprache als Arbeitsebene gewählt. Dies hat zur Folge, dass von der jeweiligen Programmiersprache der Anwendung in die Zwischensprache übersetzt werden muss. Während die verwendete Programmiersprache möglicherweise Abstraktionen und Sprachmittel bereitstellt, so werden diese jedoch zwecks Übersetzung in Maschinencode auf primitiven der nächst niedrigeren virtuellen LLVM-Maschine abgebildet. In diesem Schritt gehen für die Analyse wichtige Informationen verloren. Der dementsprechende Lösungsansatz ist die Wahl einer möglichst hohen Sprache für die Ebene, auf der die Spezialisierung durchgeführt wird. Dies steht jedoch in Konkurrenz zu dem Ziel, möglichst hochsprachenunabhängig die Spezialisierung für mehrere Hochsprachen auf Basis einer gemeinsam genutzten Zwischensprache anzubieten.

Ich habe mich für diese Arbeit dazu entschieden, die breite Anwendbarkeit meines Ansatzes stärker zu priorisieren und habe daher die Zwischensprache als Sprachebene gewählt. Damit ist zwar nicht unbedingt der theoretisch maximal mögliche Spezialisierungsgrad erreichbar, jedoch ist das Werkzeug ARA auf alle Anwendungen anwendbar, die in Sprachen implementiert sind, für die es einen Übersetzer zu LLVM-Zwischencode gibt. Um den so entstandenen Nachteil auszugleichen, wäre alternativ eine Erweiterung des Zwischencodes denkbar, sodass die Informationen aus der Hochsprache abgebildet werden können. Damit wäre sowohl die breite Anwendbarkeit als auch das Ausschöpfen des gesamten Wissens aus dem Programmcode möglich.

7.3 Ausblick und offene Fragestellungen

Bei der Betrachtung dieser Arbeit und den dazugehörigen analysierten Anwendungen fallen mir diverse Ansätze ein, die den Stand der Wissenschaft weiter voranbringen könnten. Von diesen möchte ich hier fünf vorstellen, die in direktem Bezug zu den von mir vorgestellten Spezialisierungen stehen.

Ausnutzung vollständiger Sprachunterstützung. In dieser Arbeit habe ich die grundsätzliche statische Spezialisierbarkeit dynamisch konfigurierter Systeme demonstriert, gleichzeitig aber auch die Grenzen der momentanen Implementierung. Ausgehend von den momentanen Hindernissen besteht die Frage, wie weit bestehende Anwendungen spezialisierbar wären, wären sie in einer Hochsprache wie Rust, C# oder Go geschrieben, die für alle verwendeten Idiome die entsprechende sprachliche Unterstützung anbieten. Wäre mit derartigen Implementierungen das theoretische Maximum erreichbar, das bei manueller Spezialisierung erreichbar ist oder besteht weiteres Anwendungswissen, welches so nicht im Programmcode ausgedrückt werden kann? Arbeiten in diese Richtung bringen ein besseres Verständnis für das maximal mögliche Maß an automatischer statischer Spezialisierung und haben das Potenzial richtungsweisend für zukünftige Implementierungen von Systemsoftware zu sein.

Erweiterung der Zwischensprache oder Analyse auf Quellcodelevel. Dem Informationsverlust im Übersetzungsprozess kann mit diesen beiden Mitteln entgegengewirkt werden. Die daraus entstehende Fragestellung für mich ist, ob eine Erweiterung der Zwischensprache so weit möglich ist, dass sie weiterhin für alle bisher unterstützten Hochsprachen als Basis des Übersetzungsvorganges verwendet werden kann oder ob sie dadurch so hochsprachenspezifisch wird, dass eine Analyse direkt auf Hochsprachenebene das lohnenswertere Ziel ist. Zu diesem Zweck müssten die unter-

schiedlichen Auszeichnungsmöglichkeiten und Garantien der Hochsprachen konsolidiert werden, um ein gemeinsames Modell zu formen, welches mächtigere Primitiven bietet als die bestehende Zwischensprache.

Erweiterung des Ansatzes auf Anwendungsobjekte Während ich im Rahmen dieser Arbeit nur die statische Instanziierung von Systemobjekten betrachtet habe, so denke ich, dass eine Erweiterung der Betrachtung auf Objekte der Anwendung lohnenswert sein könnte. Auch hier konnte ich in den Fallstudien beobachten, dass durchaus beträchtliche Teile des Initialisierungscodes damit verbracht werden, die immer gleichen Objekte systematisch zu erzeugen, um bei jedem Startvorgang zum gleichen Ausgangszustand zu gelangen. Eine systematische Ausweitung des bisherigen Ansatzes auf Anwendungsobjekte könnte hier gleichartige Erfolge erzielen. Auch ohne eine direkte Spezialisierung wäre zusätzlich auch eine Abschätzung des maximalen Speicherbedarfs möglich, sodass bereits zur Übersetzungszeit die Freispeicherverwaltung entsprechend dimensioniert oder eine Warnung bei zu kleiner Dimensionierung ausgegeben werden kann.

Anwendungsmodi und alternierende Existenz Im Rahmen dieser Arbeit habe ich nur solche Systemobjekte statisch instanziiert, die sicher bis zum Ende der Laufzeit existieren. Es gibt jedoch auch Anwendungen, die in verschiedenen Modi starten oder sich im Laufe ihrer Laufzeit rekonfigurieren. Dies bedeutet, dass während des Systemstarts selektiert wird, welche Konfiguration von Systemobjekten benötigt wird. Den momentanen Spezialisierungen erscheinen die Systemobjekte damit nur bedingt vorhanden, sodass sie nicht spezialisiert werden. Für eben diese Anwendungen erscheint es mir lohnenswert, die Analysen und Spezialisierungen dahingehend zu erweitern, dieses Verhalten zu erkennen und dementsprechend Objekte, die nicht zeitgleich existieren können, so im Speicher zu platzieren, dass diese diesen überlappend verwenden. Dieses Verhalten eröffnet weitere Möglichkeiten der Optimierung und kann zu einer weiteren Verringerung des Speicherbedarfs führen.

Anwendungen mit gemischtsprachlichem Quellcode In dieser Arbeit habe ich mich auf Anwendungen konzentriert, deren Quellcode in den Programmiersprachen C und C++ vorliegt. Für eine verbesserte Spezialisierbarkeit ist aber, wie in Abschnitt 6.7.1 dargelegt, die Verwendung von Sprachen mit höherer Auszeichnungskraft und der Möglichkeit, mehr Garantien geben zu können, notwendig. Ein vollständiger Sprachwechsel ist aufgrund der Menge des zu transformierenden Programmcodes eine große Belastung. Zudem werden Drittanbieterbibliotheken nicht automatisch in anderen Programmiersprachen verfügbar sein, sodass ohnehin kompatibilitätsschichten notwendig sein werden. Eine schrittweise Transition für Anwendungskomponenten, deren Neuentwicklung ohnehin notwendig ist, ist eine aufwandsoptimierte Lösung dieses Problems. Daraus erwächst die Frage, wie gut dies automatisch zu einer Verbesserung der Spezialisierbarkeit genau dieser Anwendungskomponenten führt. Eine solche schrittweise Transformation kann besonders von dem hybriden Ansatz dieser Arbeit profitieren, dass keine vollständige Information über alle Systemaufrufe vorliegen muss, jedoch die vorhandene weitestmöglich verwendet wird, um die entsprechenden Teile der Betriebssystemverwendung zu optimieren. Gleichzeitig könnten explizit Rückmeldungen gegeben werden, die beschreiben, welche Bereiche der Anwendung die größte Quelle für Unsicherheiten der Analyse sind. Für die ausstehende Sprachtransformation ist dies eine lohnenswerte Metrik mit dem Ziel, die Spezialisierung des Gesamtsystems zu steigern.

7.4 Fazit

Abschließend kann ich feststellen, dass ich in dieser Arbeit den Stand der Wissenschaft auf dem Gebiet der anwendungsgewahren statischen Spezialisierung in Bezug auf dynamische konfigurierte Systeme einen Schritt vorangebracht habe. Ich konnte zeigen, dass auch diese Systeme die Vorteile

statisch konfigurierter Systeme ausnutzen können, während die dynamischen Vorteile an den Stellen vorhanden bleiben, wo sie benötigt werden. Dies ebnet den Weg zu weiteren automatisierten statischen Spezialisierungen mit dem Ziel der Verbesserung von nichtfunktionalen Systemeigenschaften. Insgesamt zeigen die Ergebnisse, dass eine Betrachtung der tatsächlich von einer Anwendung verwendeten Komponenten und Interaktionsmuster dazu führt, dass deutlich effizientere und ressourcenschonende Systeme erzeugt werden können. Gleichzeitig zeigt die Vielfalt der zu betrachtenden Aspekte während des Spezialisierungsprozesses, dass diese Aufgabe nicht manuell ausgeführt werden sollte. Vielmehr bin ich der festen Überzeugung, dass durch eine automatische Ausführung und Integration in den Übersetzungsprozess der ideale und anzustrebende Weg ist, derartige Spezialisierungen anzuwenden. Nur so sind qualitativ hochwertige, ressourcenschonende und reproduzierbare Ergebnisse erzielbar.

Glossar

ABB Atomarer Basisblock **Siehe:** Abschnitt 3.2.1
Eine Menge von BBs, von denen nur einer eingehende Kanten und nur einer ausgehende Kanten hat, die nicht wieder auf Knoten der Menge zeigen. Hierbei kann Programmlogik, die systemaufruffrei ist, in einen ABB zusammengefasst werden, um die Komplexität des Kontrollflussgraphen zu reduzieren. Systemaufrufe oder Funktionsrufe, die indirekt einen Systemaufruf rufen, stehen jeweils alleine in ihrem eigenen ABB.

BB Basisblock **Siehe:** [ASU86]
Eine Sequenz von Instruktionen. Diese kann nur über die erste Instruktion betreten werden und über die letzte verlassen werden. Die dazwischenliegenden Instruktionen werden sequenziell abgearbeitet.

EZBS Echtzeitbetriebssystem **Siehe:** Abschnitt 2.1
Ein Betriebssystem, welches in der Lage ist, Ausführungszeiten zu garantieren.

ICFG interprozeduraler Kontrollflussgraph **Siehe:** Abschnitt 3.2.1
Kontrollflussgraph, der die in den LCFGs vorhandenen Aufrufstellen zu anderen Funktionen mit diesen verbindet. So entsteht ein Kontrollflussgraph, der die Aufrufbeziehungen der Funktionen beschreibt.

IIG Instanz- und Interaktionsgraph **Siehe:** Abschnitt 3.1, Abschnitt 3.1
Graphstruktur, die als Knoten die Systemobjektinstanzen einer gegebenen Anwendung enthält. Die Kanten repräsentieren die möglichen Interaktionen zwischen diesen.

INA statische Interaktionsanalyse **Siehe:** Abschnitt 3.3
Erweiterung der SIA um die Ermittlung der Interaktionen einer gegebenen Anwendung aus deren Quellcode.

LCFG lokaler Kontrollflussgraph **Siehe:** Abschnitt 3.2.1
Kontrollflussgraph, der den Ablauf innerhalb einer Funktion beschreibt.

OIL OSEK Implementation Language **Siehe:** Abschnitt 2.2.1.1
Konfigurationssprache der statisch konfigurierten OSEK-Systeme.

RAII Ressourcenallokation ist Initialisierung **Siehe:** [Str97] Kapitel 14.4
Ein Entwurfsmuster, das zur systematischen Allokation und Deallokation von Ressourcen verwendet wird. Dabei wird die Ressource durch Initialisieren einer lokalen Variable angefordert und automatisch wieder freigegeben, sobald die Gültigkeit der Variable endet.

RLE Lauflängencodierung (engl. run-length encoding, kurz RLE) **Siehe:** [Sal07]
Verlustloses Kompressionsverfahren, welches die Daten als Sequenz von Tupeln codiert. Jedes Tupel besteht jeweils aus einem Datum und einem Wiederholungszähler.

SIA statische Instanzanalyse **Siehe:** Abschnitt 3.2
Verfahren zur Ermittlung der Betriebssystemobjekte einer gegebenen Anwendung aus deren Quellcode.

SSP Systemstartpunkt **Siehe:** Abschnitt 4.1.1
Der Zeitpunkt im Ablauf des Systemstarts, ab dem das System als vollständig initialisiert und

arbeitsfähig gilt. Ab diesem Zeitpunkt befindet sich das System nicht mehr in der Initialisierungsphase, sondern betritt die reguläre Arbeitsphase.

Literatur

[ACT99] G. Antoniol, F. Calzolari und P. Tonella. „Impact of function pointers on the call graph“. In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*. 1999, S. 51–59. DOI: 10.1109/CSMR.1999.756682.

[AEE03] AEEC. *Avionics Application Software Standard Interface (ARINC Specification 653-1)*. ARINC Inc, 2003.

[AG21] Volkswagen AG. *Geschäftsbericht 2020*. 2021.

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[AUT13] AUTOSAR. *Specification of Operating System (Version 5.1.0)*. Techn. Ber. Automotive Open System Architecture GbR, Feb. 2013.

[Abd+16] F. Abdi, R. Mancuso, S. Bak, O. Dantsker und M. Caccamo. „Reset-based recovery for real-time cyber-physical systems with temporal safety constraints“. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, S. 1–8.

[Abd+17] F. Abdi, R. Mancuso, R. Tabish und M. Caccamo. „Restart-based fault-tolerance: System design and schedulability analysis“. In: *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2017, S. 1–10.

[Agr+06] Jason Agron, Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot und Jim Stevens. „Run-Time Services for Hybrid CPU/FPGA Systems on Chip“. In: *Proceedings of the 27th IEEE International Symposium on Real-Time Systems (RTSS '06)*. 2006, S. 3–12. DOI: 10.1109/RTSS.2006.45.

[Ake+20] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer und Robert I. Davis. „An Empirical Survey-based Study into Industry Practice in Real-time Systems“. In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, S. 3–11. DOI: 10.1109/RTSS49844.2020.00012.

[All70] Frances E. Allen. „Control Flow Analysis“. In: *ACM SIGPLAN Notices* 5.7 (Juli 1970), S. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[And94] Lars Ole Andersen. „Program Analysis and Specialization for the C Programming Language“. (DIKU report 94/19). Diss. DIKU, University of Copenhagen, Mai 1994.

[Ass10] TRON Association. *μItron 4.0 Specification, Ver. 4.03.00*. Hrsg. von Ken Sakamura. 2010.

[BDL22] Malte Bargholz, Christian Dietrich und Daniel Lohmann. „PSIC: Priority-Strict Multi-Core IRQ Processing“. In: *Proceedings of the 25th International Symposium on Real-Time Distributed Computing*. Västerås, Sweden: IEEE Computer Society, Mai 2022. ISBN: 978-1-6654-0627-7/22. DOI: 10.1109/ISORC52572.2022.9812796.

[BGS94] David F. Bacon, Susan L. Graham und Oliver J. Sharp. „Compiler transformations for high-performance computing“. en. In: *ACM Comput. Surv.* 26.4 (Dez. 1994), S. 345–420. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/197405.197406. URL: https://dl.acm.org/doi/10.1145/197405.197406 (besucht am 16. 06. 2021).

[BPS99] Sally C Brailsford, Chris N Potts und Barbara M Smith. „Constraint satisfaction problems: Algorithms and applications“. In: *European journal of operational research* 119.3 (1999), S. 557–581.

[BU02] Uwe Brinkschulte und Theo Ungerer. *Mikrocontroller und Mikroprozessoren*. Springer, 2002. ISBN: 3-540-43095-4.

[Bac+18] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt und Matthias Wählisch. „RIOT: An open source operating system for low-end embedded devices in the IoT“. In: *IEEE Internet of Things Journal* 5.6 (2018), S. 4428–4440.

[Bal+15] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli und Luca Benini. „Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems“. In: *IEEE Embedded Systems Letters* 7.1 (2015). Citation Key: balsamo:15:ELS, 15–18. ISSN: 1943-0671. DOI: 10.1109/LES.2014.2371494.

[Bar10] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd, 2010. ISBN: 978-1-4461-6914-8.

[Ber+06] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho und Nacho Navarro. „Building a Global System View for Optimization Purposes“. In: *Proceedings of the 2nd Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)* (Boston, USA). Washington, DC, USA: IEEE Computer Society Press, Juni 2006.

[CHT91] Keith D. Cooper, Mary W. Hall und Linda Torczon. „An experiment with inline substitution“. In: *Software: Practice and Experience* 21.6 (1991), S. 581–601. DOI: https://doi.org/10.1002/spe.4380210604. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380210604. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380210604.

[CHT92] Keith D. Cooper, Mary W. Hall und Linda Torczon. „Unexpected Side Effects of Inline Substitution: A Case Study“. In: *ACM Lett. Program. Lang. Syst.* 1.1 (März 1992), 22–32. ISSN: 1057-4514. DOI: 10.1145/130616.130619. URL: https://doi.org/10.1145/130616.130619.

[CRM91] Stuart K. Card, George G. Robertson und Jock D. Mackinlay. „The information visualizer, an information workspace“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '91. New York, NY, USA: Association for Computing Machinery, 1991, 181–186. ISBN: 978-0-89791-383-6. DOI: 10.1145/108844.108874. URL: https://doi.org/10.1145/108844.108874.

[Cha+92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen und Wen-Mei W. Hwu. „Profile-guided automatic inline expansion for C programs“. In: *Software: Practice and Experience* 22.5 (1992), S. 349–369. DOI: https://doi.org/10.1002/spe.4380220502. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380220502. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380220502.

[Com+11] C++ Standards Committee u. a. *IEC 14882: 2011 Information technology — Programming languages — C++*. 2011.

[DF99] Rina Dechter und Daniel Frost. *Backtracking algorithms for constraint satisfaction problems*. Techn. Ber. Technical Report, 1999.

[DH92] J.W. Davidson und A.M. Holler. „Subprogram inlining: a study of its effects on program execution time“. en. In: *IIEEE Trans. Software Eng.* 18.2 (Feb. 1992), S. 89–102. ISSN: 00985589. DOI: 10.1109/32.121752. URL: http://ieeexplore.ieee.org/document/121752/ (besucht am 10. 06. 2021).

[DHL15] Christian Dietrich, Martin Hoffmann und Daniel Lohmann. „Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems“. In: *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)* (Portland, Oregon, USA). New York, NY, USA: ACM Press, Juni 2015. ISBN: 978-1-4503-3257-6. DOI: 10.1145/2670529.2754963.

[DHL17] Christian Dietrich, Martin Hoffmann und Daniel Lohmann. „Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis“. In: *ACM Transactions on Embedded Computing Systems* 16.2 (2017), 35:1–35:25. DOI: 10.1145/2950053.

[DL17] Christian Dietrich und Daniel Lohmann. „OSEK-V: Application-Specific RTOS Instantiation in Hardware“. In: *Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '17)* (Barcelona, Spain). New York, NY, USA: ACM Press, Juni 2017. DOI: 10.1145/3078633.3078637.

[DL18] Christian Dietrich und Daniel Lohmann. „Semi-Extended Tasks: Efficient Stack Sharing Among Blocking Threads“. In: *Proceedings of the 39th IEEE Real-Time Systems Symposium 2018*. Hrsg. von Sebastian Altmeyer. Nashville, Tennessee, USA: IEEE Computer Society Press, 2018. DOI: 10.1109/RTSS.2018.00049.

[Dan+14] Daniel Danner, Rainer Müller, Wolfgang Schröder-Preikschat, Wanja Hofer und Daniel Lohmann. „Safer Sloth: Efficient, Hardware-Tailored Memory Protection“. In: *Proceedings of the 20th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '14)*. Washington, DC, USA: IEEE Computer Society Press, 2014, S. 37–47. DOI: 10.1109/RTAS.2014.6925989.

[Die+17] Christian Dietrich, Peter Wägemann, Peter Ulbrich und Daniel Lohmann. „SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems“. In: *Proceedings of the 23rd IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '17)*. Washington, DC, USA: IEEE Computer Society Press, 2017, S. 37–48. ISBN: 978-1-5090-5269-1. DOI: 10.1109/RTAS.2017.37.

[Die19] Christian Dietrich. „Interaction-Aware Analysis and Optimization of Real-Time Application and Operating System“. Diss. Leibniz Universität Hannover, 2019. DOI: 10.15488/7253. URL: https://www.repo.uni-hannover.de/handle/123456789/7306.

[Die+19] Christian Dietrich, Stefan Naumann, Robin Thrift und Daniel Lohmann. „RT.js: Practical Real-Time Scheduling for Web Applications“. In: *Proceedings of the 40th IEEE Real-Time Systems Symposium 2019*. Hong Kong, China: IEEE Computer Society Press, 2019. DOI: 10.1109/RTSS46320.2019.00017.

[Dij68] Edsger Wybe Dijkstra. „The Structure of the THE-Multiprogramming System“. In: *Communications of the ACM* 11.5 (Mai 1968), S. 341–346.

[Doh+04a] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit und Guy L. Steele. „DCAS is not a silver bullet for nonblocking algorithm design“. In: *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. SPAA '04. New York, NY, USA: Association for Computing Machinery, 2004, 216–224. ISBN: 978-1-58113-840-5. DOI: 10.1145/1007912.1007945. URL: https://doi.org/10.1145/1007912.1007945.

[Doh+04b] Simon Doherty, Maurice Herlihy, Victor Luchangco und Mark Moir. „Bringing practical lock-free synchronization to 64-bit applications“. en. In: *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing - PODC '04*. St. John's, Newfoundland, Canada: ACM Press, 2004, S. 31. ISBN: 978-1-58113-802-3. DOI: 10.1145/1011767.1011773. URL: http://portal.acm.org/citation.cfm?doid=1011767.1011773.

[Du+20] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu und Haibo Chen. „Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting“. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, 467–481. ISBN: 9781450371025. DOI: 10.1145/3373376.3378512.

[EET19] EETimes, Aspencore. *2019 Embedded Markets Study*. März 2019.

[EKO95] Dawson R. Engler, M. Frans Kaashoek und James O'Toole. „Exokernel: An Operating System Architecture for Application-Level Resource Management“. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain, CO, USA). New York, NY, USA: ACM Press, Dez. 1995, S. 251–266. ISBN: 0-89791-715-4. DOI: 10.1145/224057.224076.

[ENL22] Gerion Entrup, Jan Neugebauer und Daniel Lohmann. „RTOS-Independent Interaction Analysis in ARA“. In: *Proceedings of the 16th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '22)* (Modena, Italy). Juli 2022.

[ESD19] Gerion Entrup, Benedikt Steinmeier und Christian Dietrich. „ARA: Automatic Instance-Level Analysis in Real-Time Systems“. In: *Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '19)* (Stuttgart, Germany). Juli 2019.

[Ecm] *C# Language Specification*. Standard ECMA-334. 6. Ausgabe. Geneva, CH: European Computer Manufacturers Association (ECMA), Juni 2022.

[Eri] Erika. *ERIKA Enterprise*. URL: http://erika.tuxfamily.org (besucht am 29. 09. 2022).

[▷Fie+18] **Björn Fiedler**, Gerion Entrup, Christian Dietrich und Daniel Lohmann. „Levels of Specialization in Real-Time Operating Systems“. In: *Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '18)* (Barcelona, Spain). Juli 2018.

[▷Fie+21] **Björn Fiedler**, Gerion Entrup, Christian Dietrich und Daniel Lohmann. „ARA: Static Initialization of Dynamically-Created System Objects“. In: *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21)* (Virtual Event). Mai 2021, S. 400–412. DOI: 10.1109/RTAS52030.2021.00039.

[GH01] Elana Granston und Anne Holler. „Automatic Recommendation of Compiler Options“. In: *In Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO*. 2001.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN: 0-201-63361-2.

[Get16] Jens Getreu. „Embedded System Security with Rust“. In: *Tallinn University of Technology: Tallinn, Estonia* (2016).

[HDL13] Martin Hoffmann, Christian Dietrich und Daniel Lohmann. „dOSEK: A Dependable RTOS for Automotive Applications“. In: *Proceedings of the 19th International Symposium on Dependable Computing (PRDC '13)* (Vancouver, British Columbia, Canada). Fast abstract. Washington, DC, USA: IEEE Computer Society Press, Dez. 2013, S. 120–121. DOI: 10.1109/PRDC.2013.22.

[HFC76] Arie Nicolaas Habermann, Lawrence Flon und Lee W. Cooprider. „Modularization and Hierarchy in a Family of Operating Systems“. In: *Communications of the ACM* 19.5 (1976), S. 266–272.

[HLSP11] Wanja Hofer, Daniel Lohmann und Wolfgang Schröder-Preikschat. „Sleepy Sloth: Threads as Interrupts as Threads“. In: *Proceedings of the 32nd IEEE International Symposium on Real-Time Systems (RTSS '11)* (Vienna, Austria, 29. Nov.–2. Dez. 2011). IEEE Computer Society Press, Dez. 2011, S. 67–77. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.14.

[HPS02] Hai Huang, Padmanabhan Pillai und Kang G Shin. „Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems.“ In: *Proceedings of the USENIX Annual Technical Conference (ATC '02)*. 2002, S. 303–316.

[HW90] Maurice Herlihy und Jeannette M. Wing. „Linearizability: A Correctness Condition for Concurrent Objects“. In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), S. 463–492. DOI: 10.1145/78969.78972.

[Hei+19] Bernhard Heinloth, Marco Ammon, Dustin Nguyen, Timo Hönig, Volkmar Sieh und Wolfgang Schröder-Preikschat. „Cocoon: Custom-Fitted Kernel Compiled on Demand“. In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. Hrsg. von ACM. Huntsville, ON, Canada, 2019, S. 1–7. ISBN: 978-1-4503-7017-2. DOI: 10.1145/3365137.3365398. URL: https://www4.cs.fau.de/Publications/2019/heinloth_19_plos.pdf.

[Her+05] Maurice Herlihy, Victor Luchangco, Paul Martin und Mark Moir. „Nonblocking memory management support for dynamic-sized data structures“. In: *ACM Transactions on Computer Systems* 23.2 (2005), 146–196. ISSN: 0734-2071. DOI: 10.1145/1062247.1062249.

[Hin01] Michael Hind. „Pointer Analysis: Haven't We Solved This Problem Yet?“ In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, 54–61. ISBN: 1581134134. DOI: 10.1145/379605.379665. URL: https://doi.org/10.1145/379605.379665.

[Hof+09] Wanja Hofer, Daniel Lohmann, Fabian Scheler und Wolfgang Schröder-Preikschat. „Sloth: Threads as Interrupts“. In: *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)* (Washington, D.C., USA, 1.–4. Dez. 2009). IEEE Computer Society Press, Dez. 2009, S. 204–213. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.18.

[Hof+12] Wanja Hofer, Daniel Danner, Rainer Müller, Fabian Scheler, Wolfgang Schröder-Preikschat und Daniel Lohmann. „Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS“. In: *Proceedings of the 33rd IEEE International Symposium on Real-Time Systems (RTSS '12)* (San Juan, Puerto Rico, 4.–7. Dez. 2012). IEEE Computer Society Press, Dez. 2012, S. 237–247. ISBN: 978-0-7695-4869-2. DOI: 10.1109/RTSS.2012.75.

[Hof14] Wanja Hofer. „Sloth: The Virtue and Vice of Latency Hiding in Hardware-Centric Operating Systems“. Diss. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2014. URL: http://opus4.kobv.de/opus4-fau/files/4875/WanjaHoferDissertation.pdf.

[Hof+14] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk und Daniel Lohmann. „Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs“. In: *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)* (Reno, Nevada, USA). IEEE Computer Society Press, 2014, S. 230–237. DOI: 10.1109/ISORC.2014.26.

[Hof+15] Martin Hoffmann, Florian Lukas, Christian Dietrich und Daniel Lohmann. „dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel“. In: *Proceedings of the 21st IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '15)*. Washington, DC, USA: IEEE Computer Society Press, 2015, S. 259 –270. DOI: 10.1109/RTAS.2015.7108449.

[Hof16] Martin Hoffmann. „A Dependability-Oriented Static Embedded Kernel“. Diss. Erlangen: Friedrich-Alexander University Erlangen-Nuremberg, 2016.

[ISO11] ISO 26262-4. *ISO 26262-4:2011: Road vehicles – Functional safety – Part 4: Product development at the system level*. Geneva, Switzerland: International Organization for Standardization, 2011.

[JRR14] Hrishikesh Jayakumar, Arnab Raha und Vijay Raghunathan. „QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers“. In: *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. Citation Key: jayakumar:14:VLSID. 2014, 330–335. DOI: 10.1109/VLSID.2014.63.

[Jay+15] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee und Vijay Raghunathan. „QuickRecall: A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers“. In: *J. Emerg. Technol. Comput. Syst.* 12.1 (2015). ISSN: 1550-4832. DOI: 10.1145/2700249. URL: https://doi.org/10.1145/2700249.

[Jo+09] Heeseung Jo, Hwanju Kim, Hyun-Gul Roh und Joonwon Lee. „Improving the startup time of digital TV“. In: *IEEE Transactions on Consumer Electronics* 55.2 (2009), 721–727. ISSN: 1558-4127. DOI: 10.1109/TCE.2009.5174445.

[KN18] Steve Klabnik und Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.

[KSS20] Archanaa S. Krishnan, Charles Suslowicz und Patrick Schaumont. „Secure and Stateful Power Transitions in Embedded Systems“. en. In: *Journal of Hardware and Systems Security* 4.4 (2020), 263–276. ISSN: 2509-3436. DOI: 10.1007/s41635-020-00099-6.

[Kla+14] Tobias Klaus, Florian Franzmann, Tobias Engelhard, Fabian Scheler und Wolfgang Schröder-Preikschat. „Usable RTOS-APIs?“ In: *Proceedings of the 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '14)*. Hrsg. von Björn B. Brandenburg und Shinpei Kato. Madrid, 2014, S. 61–66. URL: https://www4.cs.fau.de/Publications/2014/klaus_14_ospert.pdf.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. First Edition. Kluwer Academic Publishers, 1997. ISBN: 0-7923-9894-7.

[Kri21] Archanaa Santhana Krishnan. *Top-down Approach To Securing Intermittent Embedded Systems*. Blacksburg, Virginia, 2021. URL: http://hdl.handle.net/10919/105128.

[Kum92] Vipin Kumar. „Algorithms for constraint-satisfaction problems: A survey“. In: *AI magazine* 13.1 (1992), S. 32–32.

[LA04] Chris Lattner und Vikram Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation“. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). Washington, DC, USA: IEEE Computer Society Press, März 2004.

[LP07] Enno Lübbers und Marco Platzner. „ReconOS: An RTOS Supporting Hard- and Software Threads“. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*. 2007, S. 441–446. DOI: 10.1109/FPL.2007.4380686.

[LP09] Enno Lübbers und Marco Platzner. „ReconOS: Multithreaded Programming for Reconfigurable Computers“. In: *ACM Transactions on Embedded Computing Systems* 9.1 (Okt. 2009), 8:1–8:33. ISSN: 1539-9087. DOI: 10.1145/1596532.1596540.

[Lam77a] L. Lamport. „Proving the Correctness of Multiprocess Programs“. In: *IEEE Trans. Softw. Eng.* 3.2 (1977), 125–143. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904. URL: https://doi.org/10.1109/TSE.1977.229904.

[Lam77b] Leslie Lamport. „Concurrent reading and writing“. en. In: *Communications of the ACM* 20.11 (1977), 806–811. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359863.359878.

[Lam83] Butler W. Lampson. „Hints for Computer System Design“. In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP '83)* (Bretton Woods, New Hampshire, USA). New York, NY, USA: ACM Press, 1983, S. 33–48. ISBN: 0-89791-115-6. DOI: 10.1145/800217.806614.

[Liu00] Jane W. S. Liu. *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0-13-099651-3.

[Liu21] Cong Liu. „A General Framework to Detect Design Patterns by Combining Static and Dynamic Analysis Techniques“. In: *International Journal of Software Engineering and Knowledge Engineering* 31.01 (2021), 21–54. ISSN: 0218-1940. DOI: 10.1142/S0218194021400027.

[Llv] *LLVM Opaque Pointers*. https://llvm.org/docs/OpaquePointers.html, visited 2022-11-22. URL: https://llvm.org/docs/OpaquePointers.html.

[Loh14] Daniel Lohmann. „Tailorable System Software“. Habilitationsschrift. Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät, 2014. URL: http://nbn-resolving.de/urn:nbn:de:bvb:29-opus4-62464.

[MRR04] Ana Milanova, Atanas Rountev und Barbara G. Ryder. „Precise Call Graphs for C Programs with Function Pointers“. In: *Automated Software Engineering* 11.1 (Jan. 2004), S. 7–26. ISSN: 1573-7535. DOI: 10.1023/B:AUSE.0000008666.56394.a1. URL: https://doi.org/10.1023/B:AUSE.0000008666.56394.a.

[MS96] Maged M. Michael und Michael L. Scott. „Simple, fast, and practical non-blocking and blocking concurrent queue algorithms“. In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. PODC '96. New York, NY, USA: Association for Computing Machinery, 1996, 267–275. ISBN: 978-0-89791-800-8. DOI: 10.1145/248052.248106. URL: https://doi.org/10.1145/248052.248106.

[MSS99] Christian Müller-Schloer und Schallenberger, Hrsg. *Vom Arbeitsplatzrechner zum ubiquitären Computer*. VDE-Verlag GmbH, 1999. ISBN: 3-8007-2364-6.

[Mar06] Peter Marwedel. *Embedded System Design*. Heidelberg, Germany: Springer-Verlag, 2006.

[Mas02] Anthony J. Massa. *Embedded Software Development with eCos*. New Riders, 2002. ISBN: 978-0130354730.

[Mas03] Anthony J. Massa. *Embedded Software Development with eCos*. Prentice Hall, 2003. ISBN: 978-0130354732.

[Mil68] Robert B. Miller. „Response time in man-computer conversational transactions“. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. AFIPS '68 (Fall, part I). New York, NY, USA: Association for Computing Machinery, 1968, 267–277. ISBN: 978-1-4503-7899-4. DOI: 10.1145/1476589.1476628. URL: https://doi.org/10.1145/1476589.1476628.

[Mis] *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*. Okt. 2004. ISBN: 0-9524156-2-3.

[Mül+14] Rainer Müller, Daniel Danner, Wolfgang Schröder-Preikschat und Daniel Lohmann. „MultiSloth: An Efficient Multi-Core RTOS using Hardware-Based Scheduling“. In: *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS '14)* (Madrid, Spain). Washington, DC, USA: IEEE Computer Society Press, 2014, S. 289–198. ISBN: 978-1-4799-5798-9. DOI: 10.1109/ECRTS.2014.30.

[Mur+19] Girish Mururu, Chris Porter, Prithayan Barua und Santosh Pande. „Binary Debloating for Security via Demand Driven Loading“. In: *CoRR* abs/1902.06570 (2019). arXiv: 1902.06570. URL: http://arxiv.org/abs/1902.06570.

[Mur+98] Gail C Murphy, David Notkin, William G Griswold und Erica S Lan. „An empirical study of static call graph extractors“. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7.2 (1998), S. 158–191.

[NR69] *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Garmisch, Germany, 7-11 Oct. 1968. Brussels: Scientific Affairs Division, NATO, 1969, S. 231. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html.

[Nie93] Jakob Nielsen. *Usability Engineering*. Englisch. 1. Aufl. Boston: Morgan Kaufmann, 1993. ISBN: 978-0-12-518406-9.

[OSE04] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. Techn. Ber. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2014-09-29. OSEK/VDX Group, 2004.

[OSE05] OSEK/VDX Group. *Operating System Specification 2.2.3*. Techn. Ber. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.

[PMI88] Calton Pu, Henry Massalin und John Ioannidis. „The Synthesis Kernel“. In: *Computing Systems* 1.1 (1988), S. 11–32.

[Pet05] Niklas Pettersson. „Measuring precision for static and dynamic design pattern recognition as a function of coverage“. In: *Proceedings of the third international workshop on Dynamic analysis*. WODA '05. New York, NY, USA: Association for Computing Machinery, 2005, 1–7. ISBN: 978-1-59593-126-9. DOI: 10.1145/1083246.1083253. URL: https://doi.org/10.1145/1083246.1083253.

[Pos] *Portable Operating System Interfaces (POSIX®) – Part 1: System Application Program Interface (API) – Amendment 1: Realtime Extension*. 1998.

[RC67] A.H. Robinson und C. Cherry. „Results of a prototype television bandwidth compression scheme“. In: *Proceedings of the IEEE* 55.3 (1967), S. 356–364. DOI: 10.1109/PROC.1967.5493.

[RG89] Stephen Richardson und Mahadevan Ganapathi. „Interprocedural Analysis vs. Procedure Integration“. In: *Inf. Process. Lett.* 32.3 (Aug. 1989), 137–142. ISSN: 0020-0190. DOI: 10.1016/0020-0190(89)90014-8. URL: https://doi.org/10.1016/0020-0190(89)90014-8.

[RHL14] Andreas Ruprecht, Bernhard Heinloth und Daniel Lohmann. „Automatic Feature Selection in Large-Scale System-Software Product Lines“. In: *Proceedings of the 13th International Conference on Generative Programming and Component Engineering (GPCE '14)* (Västerås, Sweden). Hrsg. von Matthew Flatt. New York, NY, USA: ACM Press, Sep. 2014, S. 39–48. ISBN: 978-1-4503-3161-6. DOI: 10.1145/2658761.2658767.

[RT12] Thomas Bryan Rushworth und Angus Richard Telfer. „Multi-Reader, Multi-Writer Lock-Free Ring Buffer“. Pat. US Patent 8,095,727 B2. 2012.

[Rad+22] Alexandru Radovici, Ioana Culic, Alexandru Radovici und Ioana Culic. „Embedded Systems Software Development“. In: *Getting Started with Secure Embedded Systems: Developing IoT Systems for micro: bit and Raspberry Pi Pico Using Rust and Tock* (2022), S. 27–47.

[Ras00] Jef Raskin. *The Humane Interface – New Directions for Designing Interactive Systems*. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN: 978-0-201-37937-2.

[Rou81] Robert Routledge. *Discoveries and inventions of the nineteenth century*. fifth edition. https://archive.org/details/discoveriesinven00routrich/page/6. George Routledge und Sons, London, 1881.

[SFP20] Paulo Silva, Daniel Fireman und Thiago Emmanuel Pereira. „Prebaking Functions to Warm the Serverless Cold Start“. In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, 2020, 1–13. ISBN: 9781450381536. DOI: 10.1145/3423211.3425682.

[SGG12] Abraham Silberschatz, Greg Gagne und Peter Bear Galvin. *Operating System Concepts*. Ninth. John Wiley & Sons, Inc., 2012. ISBN: 978-1-118-06333-0.

[SH97] Marc Shapiro und Susan Horwitz. „Fast and accurate flow-insensitive points-to analysis“. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, S. 1–14.

[SSP10] Fabian Scheler und Wolfgang Schröder-Preikschat. „The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems“. In: *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '10)* (Carmona, Spain). Washington, DC, USA: IEEE Computer Society Press, Mai 2010, S. 34–41. ISBN: 978-0-7695-4037-5. DOI: 10.1109/ISORC.2010.11.

[SX16] Yulei Sui und Jingling Xue. „SVF: Interprocedural Static Value-Flow Analysis in LLVM“. In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, 2016, 265–266. ISBN: 9781450342414. DOI: 10.1145/2892208.2892235.

[Sal07] D. Salomon. *Data compression: the complete reference*. en. 4th ed. London: Springer, 2007. ISBN: 978-1-84628-602-5.

[Sch11] Fabian Scheler. „Atomic Basic Blocks: Eine Abstraktion für die gezielte Manipulation der Echtzeitsystemarchitektur“. Diss. Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät, 2011. URL: http://opus4.kobv.de/opus4-fau/files/1740/FabianSchelerDissertation.pdf.

[Sch12] Kurt Schneider. *Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. de. Google-Books-ID: GxZ4DwAAQBAJ. dpunkt.verlag, 2012. ISBN: 978-3-86491-109-5.

[Ste96] Bjarne Steensgaard. „Points-to Analysis in Almost Linear Time“. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, S. 32–41. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237727.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[TNHTSA18] U.S. Department of Transportation National Highway Traffic Safety Administration. *Laboratory Test Procedure for FMVSS111 – Rear Visibility (other than schoolbusses)*. 2018.

[Tri+03] S. Triantafyllis, M. Vachharajani, N. Vachharajani und D.I. August. „Compiler optimization-space exploration“. In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* März 2003, S. 204–215. DOI: 10.1109/CGO.2003.1191546.

[Tsu+84] Tokuhiro Tsukiyama, Yoshie Kondo, Katsuharu Kakuse, Shinpe Saba, Syoji Ozaki und Kunihiro Itoh. „Method and system for data compression and restoration“. In: *US Patent No. 4589027* (Aug. 1984).

[UM21] Sumanth Umesh und Sparsh Mittal. „A survey of techniques for intermittent computing“. en. In: *Journal of Systems Architecture* 112 (2021), S. 101859. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2020.101859.

[Uc17] Tunç Uzlu und Ediz Şaykol. „On utilizing rust programming language for Internet of Things“. In: *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*. 2017, S. 93–96. DOI: 10.1109/CICN.2017.8319363.

[VA20] Mónica M Villegas und Hernán Astudillo. „OTA updates mechanisms: a taxonomy and techniques catalog“. In: *XXI Simposio Argentino de Ingeniería de Software (ASSE 2020)-JAIIO 49 (Modalidad virtual)*. 2020.

[Wäg+18] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich und Wolfgang Schröder-Preikschat. „Whole-System WCEC Analysis for Energy-Constrained Real-Time Systems (Artifact)“. In: *Dagstuhl Artifacts Series* 4.2 (2018), 7:1–7:4. ISSN: 2509-8195. DOI: 10.4230/DARTS.4.2.7. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8975.

[Wit00] Michael P. Witzak. *Echtzeit-Betriebssysteme*. Franzis Verlag, 2000. ISBN: 3-7723-4293-0.

[Wul+73] William Allan Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs und Charles M. Geschke. *The design of an optimizing compiler*. en. Elsevier Science Inc., 1973, S. 107.

[Wä+18] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich und Wolfgang Schröder-Preikschat. „Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems“. In: *Proceedings of the 30th Euromicro Conference on Real-Time Systems 2018* (Barcelona, Spain). Hrsg. von Sebastian Altmeyer. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. DOI: 10.4230/LIPIcs.ECRTS.2018.24.

[Zep] Zephyr. *Zephyr Project homepage*. https://www.zephyrproject.org/.

[Zie+19] Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig und Daniel Lohmann. „Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries“. In: *ACM Transactions on Embedded Computing Systems* 18.5s (Okt. 2019), 102:1–102:23. ISSN: 1539-9087. DOI: 10.1145/3358222.

[Zim+14] Michael Zimmer, David Broman, Chris Shaver und Edward A. Lee. „FlexPRET: A processor platform for mixed-criticality systems“. In: *Proceedings of the 20th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '14)*. Washington, DC, USA: IEEE Computer Society Press, 2014, S. 101–110. DOI: 10.1109/RTAS.2014.6925994.

Abbildungsverzeichnis

1.1 Der Instanzgraph der Beispielanwendung GPSLogger 3

2.1 Beispielanwendung eines statisch konfigurierten Systems. 17
2.2 Beispielanwendung eines dynamisch konfigurierten Systems. 18
2.3 Beispielanwendung eines hybrid konfigurierten Systems. 19
2.4 Ebenen der Konfigurierbarkeit und Spezialisierung von Betriebssystemen 21

3.1 Überblick über den Aufbau von ARA 33
3.2 Kontrollflussgraph der Beispielanwendung 35
3.3 FreeRTOS Betriebssystemmodell 36
3.4 Werteflussgraph 39
3.5 Instanz- und Interaktionsgraph der Beispielanwendung aus Quellcode 3.1 40

4.1 Integration der Bestandteile von ARA in den LLVM-Übersetzungsprozess 52
4.2 Ergebnisse der spezialisierten Instanziierung von Warteschlangen 60
4.3 Ergebnisse der spezialisierten Fadenerzeugung *vor* dem Start des Planers 61
4.4 Ergebnisse der spezialisierten Fadenerzeugung *nach* dem Start des Planers 62
4.5 LibrePilot CopterControl: Ergebnisse der Systemstart-Spezialisierung 64
4.6 Instanzliste GPSLogger 66
4.7 GPSLogger: Ergebnisse der Systemstart-Spezialisierung 67

5.1 Laufzeiten der verschiedenen Puffer-Implementierungen im nicht blockierenden Fall 75
5.2 Laufzeiten der verschiedenen Puffer-Implementierungen im blockierenden Fall 76
5.3 Speicherbedarf für die unterschiedlichen Puffer-Implementierungen 79
5.4 Instanz- und Interaktionsgraph der Beispielanwendung aus Quellcode 3.1 79
5.5 Bedingungsgraph der Beispielanwendung 80
5.6 Interaktionsgraph des GPSLoggers 81

Tabellenverzeichnis

3.1 Instanzliste der Beispielanwendung aus Quellcode 3.1 38

4.1 Statisch spezialisierbare Phasen und maximale Spezialisierungstiefe 49

6.1 Hindernisse der statischen Spezialisierung und deren Ursachen. 101

Quellcodeverzeichnis

3.1 Beispielanwendung 30
3.2 Nicht eindeutig bestimmbarer Systemaufrufparameter und Kontext. 31
3.3 Die SIA-Analyse 37
4.1 Codeänderungen einer vollständigen Spezialisierung 50
4.2 Codeänderungen einer partiellen Spezialisierung 50
4.3 Nicht spezialisierbarer Anwendungscode 51
4.4 Statische Initialisierung eines Fadens 54
4.5 Statische Registrierung im Betriebssystem 55
4.6 Dynamische Registrierung eines partiell spezialisierten Systemobjekts 55
4.7 Dynamische Umsetzung der Seiteneffekte eines instanziierenden Systemaufrufs . . . 55
4.8 Benchmark zur Instanziierung von Warteschlangen 61
5.1 FreeRTOS Queue Implementierung mit globaler Unterbrechungssperre 74
5.2 Widersprüchliche Verwendung von Systemaufrufen mit Wartezeit 82
6.1 Fallbeispiel aus dem LibrePilot 88
6.2 Fallbeispiel aus dem GPSLogger 89

Björn Fiedler

Persönliche Daten

Name	Björn Fiedler
E-Mail	fiedler@sra.uni-hannover.de
Geburtsdaten	13. 05. 1992, Hannover

Wissenschaftlicher Werdegang

2002 – 2011	**Abitur**, *IGS Langenhagen*
2011 – 2012	**Juniorstudium**, *Gottfried Wilhelm Leibniz Universität Hannover* Informatik
2012 – 2015	**Bachelor of Science**, *Gottfried Wilhelm Leibniz Universität Hannover* Informatik
2015 – 2017	**Master of Science**, *Gottfried Wilhelm Leibniz Universität Hannover* Informatik
2020 – 2022	**Master of Science**, *Gottfried Wilhelm Leibniz Universität Hannover* Technische Informatik, mit Auszeichnung
2018 – 2023	**Promotionsstudium**, *Gottfried Wilhelm Leibniz Universität Hannover*, Fachgebiet für System- und Rechnerarchitektur Informatik
2018 –	**Wissenschaftlicher Mitarbeiter**, *Gottfried Wilhelm Leibniz Universität Hannover*, Fachgebiet für System- und Rechnerarchitektur

Preise und Auszeichnungen

2016	**Hack@Home Hannover**, 1. Platz
2016	**Hack@Home Deutschland**, Unter den 3 Besten
2016	**2nd BMVI Data-Run - Echtzeitdaten im Verkehr**, 1. Platz in der Kategorie Wirtschaftlichkeitspotenzial (Hauptpreis), Bundesministerium für Verkehr und digitale Infrastruktur
2018	**Best Paper Award**, *OSPERT'18*, Barcelona, Levels of Specialization in Real-Time Operating Systems
2022	**Dean's List**, *Fakultät für Elektrotechnik und Informatik*, Master Technische Informatik
	Drittbeste Masterprüfung im Studienjahr 2021/2022, *Fakultät für Elektrotechnik und Informatik*, Master Technische Informatik

Publikationen

Gerion Entrup, Björn Fiedler, and Daniel Lohmann. MultiSSE: Static syscall elision and specialization for event-triggered multi-core RTOS. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'23)*, May 2023.

Björn Fiedler, Gerion Entrup, Christian Dietrich, and Daniel Lohmann. Levels of specialization in real-time operating systems. In *Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '18)*, July 2018.

Björn Fiedler, Gerion Entrup, Christian Dietrich, and Daniel Lohmann. ARA: Static initialization of dynamically-created system objects. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21)*, pages 400–412, May 2021.