

Fakultät für Informatik
Institut für Verteilte Systeme
Lehrstuhl Embedded Smart Systems

Masterarbeit

Konzeption einer Prozesskette zur Analyse der Ausführungszeit von Simulink Modellen auf eingebetteten Systemen

Autor:

Markus Wolf

16. Mai 2017

Prüfer: Jun.-Prof. Dr. Sebastian Zug
Zweitprüfer: Dr. Lars Henning (IAV)

Wolf, Markus:

*Konzeption einer Prozesskette zur Analyse der Ausführungs dauer
von Simulink Modellen auf eingebetteten Systemen*

Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2017.

Inhaltsverzeichnis

Abbildungsverzeichnis	iv
Tabellenverzeichnis	v
Abkürzungsverzeichnis	vi
1 Einleitung	1
1.1 Hintergrund und Motivation	1
1.2 Aufgabenstellung	3
1.3 Struktur der Arbeit	3
2 Grundlagen und Stand der Technik	5
2.1 Grundlagen	5
2.1.1 MATLAB / Simulink	5
2.1.2 Entwicklung automotiver Steuergerätesoftware	6
2.1.3 Methodik der Befehlsabarbeitung in CPUs	8
2.1.4 Worst Case Execution Time Analysis	10
2.2 Stand der Technik	15
2.2.1 WCET - Tools	15
2.2.2 WCET-Analyse für Simulink Modelle mit wcetC	19
3 Anforderungsanalyse	21
3.1 Entwicklungstoolchain	21
3.2 Zielsetzung	23
3.3 Randbedingungen	24
3.4 Zusammenfassung	25
4 Konzept	27
4.1 Entwicklung der Prozesskette	27
4.2 Laufzeitbestimmung für Simulink Blöcke	31
4.3 Modellanalyse	37
4.3.1 Toolbox	37
4.3.2 Schleifen	40
4.3.3 Kontrollfluss	41
4.4 Cache- und Piping-Einflüsse	44
4.5 Zusammenfassung	45
5 Implementierung	47
5.1 Versuchsaufbau	47
5.2 Umsetzung	48

5.2.1	Laufzeitmessung der Funktionsblöcke	49
5.2.2	Implementierung der Toolbox	52
5.3	Zusammenfassung	54
6	Evaluierung	55
6.1	Testmethodik	55
6.2	Testdurchführung	56
6.3	Fazit	62
7	Zusammenfassung und Ausblick	65
7.1	Zusammenfassung	65
7.2	Ausblick	66
Literaturverzeichnis		67
A Anhang — Abbildungen		71

Abbildungsverzeichnis

1	Einfacher Standardregelkreis als Blockschaltbild in Simulink	5
2	Toolkette zur Erstellung von Steuergerätesoftware mit Simulink	6
3	V-Modell der Softwareentwicklung	7
4	Speicherhierarchie	9
5	Aufbau der superskalaren Pipeline des IBM PowerPC 750GX	10
6	Einfaches Quellcodebeispiel und dazugehöriger Kontrollflussgraph	12
7	Strukturelle Nebenbedingungen für den Graphen aus Abbildung 6	13
8	Laufzeitverteilungen	14
9	Kontrollflussgraph mit WCET Ergebnissen in aiT	16
10	Berechnung des WCET mit Chronos	18
11	Entwicklungsprozess für Simulink mit wcetC	19
12	Vereinfachtes V-Modell mit Laufzeitanalyse	21
13	Durschnittliche Blockhäufigkeiten pro Simulink Modell	22
14	Prozesskette zum Aufbau einer Lookup Table für Simulink-Funktionsblöcke	29
15	Exemplarische Prozesskette für den “Produkt-Block”	30
16	Die Modellanalyse verbindet Informationen aus Lookup Table und Modell	30
17	Ablauf der Laufzeitmessung für Simulink Blöcke	33
18	Zusammenspiel zwischen Modell, Toolbox und Lookup Table	36
19	Toolbox Aufbau und Ablauf	39
20	Einfaches Signalflussmodell mit Ausführungsreihenfolge	42
21	Beispiel für einen Switch Block	42
22	Aufbau einer If/else Verzweigung in Simulink	43
23	Versuchsaufbau zur Laufzeitmessung	47
24	Special Purpose Registers (SPR) des IBM PowerPC 750GL	49
25	Aufbau des Messmodells	50
26	Ablauf der Laufzeitanalyse (Funktionsaufruf-Schema)	53
27	Matlab-Konsolenausgabe nach der Laufzeitanalyse	54
28	Verteilung der Laufzeiten für das TMPMDL-Modell	56
29	Laufzeitwerte des TMPMDL-Modells	57
30	Verteilung der Laufzeiten für das DPFMON-Modell	58
31	Laufzeitwerte des DPFMON-Modells	59
32	Verteilung der Laufzeiten für das LAMBDA GAS-Modell	59
33	Laufzeitwerte des LAMBDA GAS-Modells	60
34	Durschnittliche Blocklaufzeiten pro Simulink Modell	61
35	System Outputs Block mit GUI in Simulink	71
36	Ausführungsduer der 1-D Lookup Table in Abhängigkeit der Stützstellen	71
37	Simulink Modell zur Vermessung mehrerer Funktionsblöcke	72
38	Aufbau von ControlDesk	73
39	Lookup Table Ausschnitt (Laufzeiten für <i>double</i>)	74

40	Lookup Table Ausschnitt (Laufzeiten für “Data Conversion”)	74
41	Lookup Table Ausschnitt mit eingebetteten Funktionen	75
42	Gesamtschema der Konzeptumsetzung mit Evaluierung	76
43	Laufzeit der Update-Funktion des TMPMDL-Modells	77
44	Laufzeit der Update-Funktion des DPFMON-Modells	77
45	Laufzeit der Update-Funktion des LAMBDA GAS-Modells	78
46	Anzahl der Cache-Misses des LAMBDA GAS-Modells	78
47	Laufzeit des TMPMDL Modells in Abhängigkeit der “Bricks”	79
48	Laufzeiten der Simulink Modelle mit Laufzeitschätzung	80
49	Laufzeitauswertung für LAMBDA GAS Modell ohne “Cache-Miss-Zeit” . .	81

Tabellenverzeichnis

1	Beispiele für Modellierungsrichtlinien in Matlab/Simulink	8
2	Prozessoren aus Steuergeräten für die Nutzung im Kraftfahrzeug	23
3	Anforderungskatalog	26
4	Gegenüberstellung zweier Analyseebenen	28
5	Wertbasierter Aufbau einer Lookup Table	34
6	Funktionsbasierter Aufbau einer Lookup Table	34
7	Skriptbasierter Aufbau einer Lookup Table	34
8	Versuchsaufbau	48
9	Versuchsmodelle	55

Abkürzungsverzeichnis

API	Application Programming Interface
ASCET	Advanced Simulation and Control Engineering Tool
AUTOSAR	Automotive Open System Architecture
BCET	Best Case Execution Time
BPU	Branch Processing Unit
CMMI	Capability Maturity Model Integration
CPU	Central Processing Unit
ECU	Electronic Control Unit
ELF	Executable and Linking Format
FPU	Floating Point Unit
HIL	Hardware in the Loop
HPC	Hardware Performance Counter
IAV	Ingenieurgesellschaft Auto und Verkehr
ILP	Integer Linear Programming
IU	Integer Unit
LSU	Load and Store Unit
MAAB	MathWorks Automotive Advisory Board
MATLAB	Matrix Laboratory
MMCR	Monitor Mode Control Register
PMC	Performance Monitor Counter
SCADE	Safety Critical Application Development Environment
SFP	Single Feasible Path
SPICE	Software Process Improvement and Capability Determination
SPR	Special Purpose Register
SRU	System Register Unit
SWEET	Swedish Execution Time Tool
WCC	WCET-Aware C Compiler
WCET	Worst Case Execution Time

1 Einleitung

1.1 Hintergrund und Motivation

In der Automobilindustrie ist der Einsatz von Elektronik und Mikroprozessoren nicht mehr wegzudenken. Versteckt übernehmen sie umfangreiche Aufgaben im Fahrzeug: Von der Kraftstoffeinspritzung über die Abgasnachbehandlung bis hin zum Auslösen der Airbags, sind sie heute in großer Zahl im Fahrzeug vorhanden. Aufgrund der zunehmenden Digitalisierung und der damit verbundenen Zunahme an Hard- und Softwarekomponenten im modernen Fahrzeugen werden auch die Aufgaben immer umfangreicher.

Angesichts der gestiegenen Komplexität bei der heutigen Funktions- und Softwareentwicklung hat sich das Arbeiten auf abstrakter Ebene mit Simulationswerkzeugen wie Matlab/Simulink etabliert und ist heute bereits Standard. Einig ist man sich darüber, dass nur durch diesen Einsatz rechnergestützter Entwurfsmethoden die Komplexität heutiger Systeme bewältigt werden kann [Tei97]. Ein Vorteil dieser modellgetriebenen Entwicklung ist die Möglichkeit automatisiert Software aus graphischen Modellen zu erzeugen. Über diese, auf jede Architektur zugeschnittene Codegenerierung und Kompilierung kann so effizienter Programmcode entwickelt werden, ohne, dass detaillierte Programmierkenntnisse notwendig sind. Modellbasierte Software kann aufgrund der graphischen, abstrakten Repräsentation einfacher gewartet und erweitert werden und ist für Erläuterungen von Softwarefunktionalitäten einfacher zu handhaben, als tatsächlicher Quellcode. Darüber hinaus lassen sich entwickelte Funktionalitäten bereits auf dem Computer simulieren und testen. Für den Entwickler besteht die Herausforderung bei zeitkritischen Anwendungen den zeitlichen Determinismus der Softwarekomponente nachzuweisen oder zumindest einzugrenzen. Natürlich kann eine Rechenaufwandsabschätzung auch für nicht zeitkritische Anwendungen sinnvoll sein, beispielsweise wenn sich die zu entwickelnde Komponente mit anderen Applikationen auf dem Steuergerät befindet.

Dem Funktionsentwickler fehlt jedoch auf abstrakter Ebene der Zugriff und damit das Wissen über benötigte Speicher- bzw. Rechenressourcen des zu entwickelnden Programms und seiner verschiedenen Module. Zudem ist das Untersuchen der Ausführungsduauer auf dem tatsächlichen Steuergerät während des Entwicklungsprozesses oft nicht möglich, da die vom Hersteller der Steuergeräte genutzte Toolkette nicht zur Verfügung steht und somit ein entsprechender Zugriff auf die Hardware fehlt. Nicht zuletzt stehen, besonders zu Beginn eines neuen Projekts, geeignete Steuergeräte mit passenden Softwareständen gar nicht, oder nicht in ausreichender Anzahl zur Verfügung [SKHX15]. Frühzeitiges Wissen über etwaige zeitliche Engpässe oder Optimierungspotentiale können sich jedoch positiv auf die Entwicklungszeit und damit auf die Kosten auswirken. Da Software heute bereits nahezu 40% der Gesamtfahrzeugkosten ausmacht, ist das Einsparpotential entsprechend groß [Men12].

Software wird im Automotiveumfeld heute in weiten Teilen immer noch nach dem V-Model entwickelt (siehe Kapitel 2.1.2) [SZ06, HSDZ⁺09]. Dieses Model geht implizit davon aus, dass Benutzeranforderungen zum Zeitpunkt der Anforderungsspezifikation (Abbildung 3) bereits vollständig bekannt und erfasst sind. Die Praxis zeigt jedoch, dass Kundenanforderungen häufig nicht vollständig bekannt sind und der Anforderungskatalog während der Entwicklung fortgeschrieben wird. Tatsächlich ist die Realität also eher durch eine inkrementelle und iterative Vorgehensweise gekennzeichnet [SZ06]. Dieser iterative Vorgang findet sich bei der Entwicklung von Funktionssoftware bei IAV wieder. Hier ist er dadurch bedingt, dass die tatsächliche Programmierung der Steuergeräte bzw. eingebetteten Systeme oft nicht von IAV durchgeführt wird, sondern der Steuergerätehersteller sich diesen Prozess vorbehält. So ist der Test der Funktionalität auf der Zielhardware, beispielsweise mit dem sogenannten Hardware-in-the-Loop¹ (HIL) Verfahren, erst in der Phase der Systemtests (Abbildung 3) möglich. Können bei diesen Tests z.B. Echtzeitanforderungen nicht eingehalten werden, ist die Spezifikation sowie die Programmierung anzupassen. Dieser Vorgang wird solange wiederholt bis die Software den Anforderungen genügt. Dieser iterative Prozess könnte durch das frühzeitige Wissen über die Ausführungsduer der Software verkürzt und damit Kosten eingespart werden.

Aktuell gibt es bereits Tools und Verfahren, um die Laufzeit, insbesondere von zeitkritischen Anwendungen, zu messen. Diese basieren zumeist auf der Analyse des Binär- oder Assemblercodes. Sie setzen also eine zuvor durchgeführte Kompilierung des Quellcodes voraus. Aufgrund dieser Hardwarenähe können mit diesen Verfahren, die zum Teil äußerst komplex sind, enge Obergrenzen für die Laufzeit eines Programms angegeben werden [Tan06]. Ohne detaillierte Programm(ier)kenntnisse sind solche Verfahren jedoch nicht anwendbar bzw. sorgen für einen deutlichen Mehraufwand bei der Funktionsentwicklung.

Neben dem praktischen Nutzen einer Laufzeitanalyse anhand eines Simulink Modells ist ferner die Frage nach der Qualität einer Aussage über hardwarenahe Vorgänge auf abstrakter Ebene interessant. Wie genau kann also eine Berechnung bzw. Abschätzung auf Modellebene für architekturspezifische Vorgänge sein? Welche Informationen werden benötigt und wie stark sind schlussendlich die Abweichung zu den tatsächlichen Messwerten?

¹Beim Hardware-in-the-Loop Test wird das Steuergerät (bzw. die Hardware) mit einem anderen Computersystem gekoppelt, wobei dieses eine reale Umgebung simuliert und entsprechende Werte an das Steuergerät sendet. So kann die Funktionalität des Geräts unter verschiedenen Bedingungen getestet werden.

1.2 Aufgabenstellung

Ziel dieser Masterarbeit ist die Spezifikation, Konzeption und prototypische Integration eines Werkzeugs in den automotiven Software-Entwicklungsprozess für eingebettete Steuergeräte bei IAV. Dieses Werkzeug soll Aussagen zum Zeitverhalten eines Simulink-Modells oder seiner Module erlauben. Mit dieser Erweiterung soll es dann möglich sein, die Auswirkung bestimmter Designentscheidungen auf die Ausführungsperformance sowohl frühzeitig abschätzen, als auch detailliert untersuchen zu können. Entsprechend gliedert sich die Aufgabe in folgende Teilaufgaben:

Herausarbeitung des Standes der Technik - Vor der Entwicklung eines eigenen Konzeptes ist ein intensives Studium der verfügbaren Methoden und Tools erforderlich, die für die Auswertung der Echtzeiteigenschaften von Simulinkmodellen bereitstehen. Zudem sind die entsprechenden Konzepte zu analysieren und zu vergleichen.

Entwicklung eines Anforderungskatalogs - Aufbauend auf einer Analyse der Entwicklungstoolchain unter Simulink und den Anforderungen bei IAV ist ein Katalog der notwendigen Elemente der intendierten Methoden zu entwickeln.

Konzeption einer Prozesskette - Auf der Basis der Analyse zum Stand der Technik und den Anforderungen ist eine Prozesskette zu modellieren, die die Analyse des Zeitverhaltens mit unterschiedlichen Genauigkeitsklassen erlaubt und es ermöglicht den Rechenaufwand auf Operationen und Module herunterzubrechen, um relevante Designentscheidungen zu identifizieren.

Implementierung - Die Umsetzung ist in die Mathworks Simulink Umgebung und den Generierungsprozess für den targetspezifischen Quellcode einzubetten.

Evaluation - Die Evaluation der Implementierung soll anhand verschiedener Ausprägungen mehrerer Beispielmodelle für bis zu zwei spezifische Targets erfolgen. Als Referenz für die Laufzeitaussagen anhand des Simulink Modells und des C-Codes dienen eigene Messungen auf der realen Hardware. Dabei ist die Fähigkeit des Ansatzes zu dokumentieren, dass eine optimierte Abstimmung zwischen Qualität des Ergebnisses und dem Zeitaufwand im Rahmen der Vorgaben erzielt werden kann.

Dokumentation der Ergebnisse - Die schriftliche Ausarbeitung besteht aus zwei Teilen, zum einen der Dokumentation des Quellcodes und zum anderen aus der eigentlichen Abschlussarbeit.

1.3 Struktur der Arbeit

In dieser Arbeit wird das Konzept einer Laufzeitanalyse auf Modellebene vorgestellt und umgesetzt. Dabei wird die Analyse nicht anhand von Quell- bzw. Binärkode durchgeführt, sondern mit Hilfe von Modell- sowie Laufzeitinformationen der einzelnen Funktionsmodule eines Simulink Modells.

Das Grundlagenkapitel (Kapitel 2) stellt den Stand der Technik dar und bietet die Wissensgrundlage für die folgenden Kapitel dieser Arbeit. Kapitel 3 definiert über die Analyse bestehender Entwicklungsprozesse und bereits erstellter Simulink Modelle einen Anforderungskatalog. In Kapitel 4 wird die Prozesskette für eine Laufzeitanalyse auf Modellebene entworfen. Es werden Ansätze zur Laufzeitmessung der Simulink Funktionsblöcke sowie der Aufbau einer in Matlab/Simulink integrierten Toolbox diskutiert. An das Konzeptkapitel schließt sich die Erläuterung der prototypischen Umsetzung der Konzeptvorgaben an. Implementiert wird eine Lookup Table mit entsprechenden Laufzeitinformationen der Simulink Funktionsblöcke, sowie eine Toolbox zur Umsetzung der Modell- und Laufzeitanalyse. Abschließend wird in Kapitel 6 das entwickelte Konzept anhand der Implementierung evaluiert. Durch die Untersuchung von drei Simulink Modellen hinsichtlich ihres Laufzeitverhaltens können die gestellten Anforderungen sowie die Konzeptvorgaben überprüft werden.

In Kapitel 7 wird die gesamte Arbeit abschließend zusammengefasst und ein Ausblick auf weiterführende Themen und mögliche Arbeiten gegeben.

2 Grundlagen und Stand der Technik

2.1 Grundlagen

In diesem Kapitel werden zunächst einige Grundlagen und grundlegende Begriffe sowie Herangehensweisen beschrieben und erläutert. Diese Erläuterungen sollen eine entsprechende Basis für das weitere Verständnis dieser Arbeit bieten. Dazu wird zunächst die Simulations- und Entwicklungsumgebung Matlab/Simulink vorgestellt und anschließend die Entwicklung von Steuergerätesoftware im Bereich Automotive dargestellt. Weiterhin werden wichtige Begrifflichkeiten wie *Caching* und *Pipelining* erläutert und der aktuelle Stand der Technik hinsichtlich bereits bestehender Lösungen zur Laufzeitanalyse untersucht.

2.1.1 MATLAB / Simulink

MATLAB (*MATrix LABoratory*) ist eine Software des Unternehmens MathWorks zur Lösung mathematischer Probleme. Insbesondere numerische Berechnungen mithilfe von Matrizen können mit diesem Programm effizient erstellt werden. Matlab ist außerdem die Basis für das Programm Simulink, welches ebenfalls von MathWorks vertrieben wird. Simulink ermöglicht die Simulation von zeitgesteuerten dynamischen Systemen, wie beispielsweise Regelsystemen. Simulink bietet somit die Möglichkeit automotive Steuergerätesoftware zu spezifizieren, zu simulieren und zu testen. Dabei ist ein Modell als Signalflussplan zu verstehen, in dem Funktionsblöcke miteinander verknüpft werden. Abbildung 1 veranschaulicht an einem einfachen Standardregelkreis die grundsätzliche

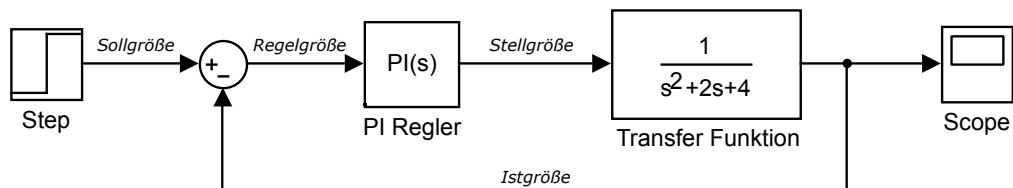


Abbildung 1: Einfacher Standardregelkreis als Blockschaltbild in Simulink

Struktur eines Simulink Modells. Deutlich komplexer, jedoch nach dem gleichen Prinzip, werden diese Regelkreise in der automotiven Funktionsentwicklung eingesetzt. Ein solcher Regelkreis besteht aus einer Sollgröße (*Step*), einem (*PI*)-*Regler* und einer Regelstrecke, welche in diesem Beispiel durch eine Übertragungsfunktion 2. Ordnung modelliert ist. Über *Scopes* oder *Displays* können Signalgrößen zur Simulationszeit eingesehen werden. Simulink bietet eine Vielzahl vorgefertigter Funktionsblöcke zur Berechnung von Differential- und Differenzgleichungen und bietet darüber hinaus die Möglichkeit über Subsysteme die häufig sehr komplexen Modelle zu strukturieren. Dabei wird ein Modell durch einen sogenannten *Solver* entweder in einer kontinuierlichen oder diskreten Simulation umgesetzt.

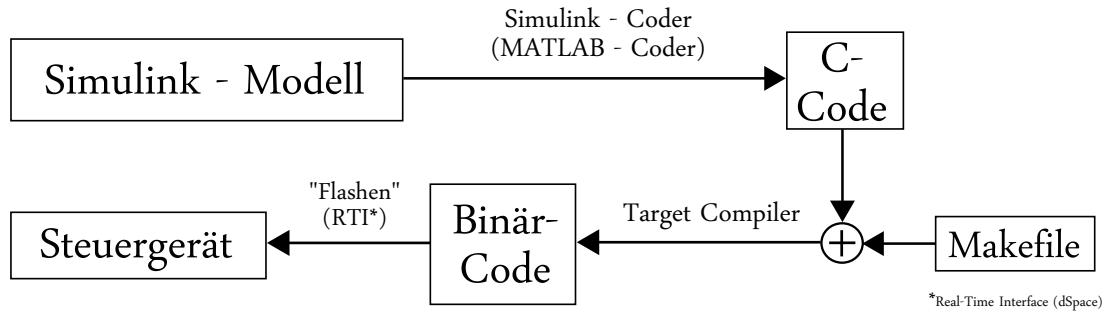


Abbildung 2: Toolkette zur Erstellung von Steuergerätesoftware mit Simulink

Über die „*Simulink Coder Toolbox*“ lässt sich aus den Modellen C-Code generieren. Dies ermöglicht dem Funktionsentwickler die modellierte Funktionalität, neben der Simulation, ebenfalls auf einem Steuergerät oder einer anderen Hardware zu testen. Oft kann bereits im Simulinkprogramm ein Buildprozess angestoßen werden, der anhand eines Makefiles mit dem entsprechenden Compiler das gewünschte Binärprogramm erzeugt. Diese Toolkette, wie sie auch in dieser Arbeit verwendet wird, ist in Abbildung 2 exemplarisch dargestellt.

2.1.2 Entwicklung automotiver Steuergerätesoftware

Zunächst soll als Grundlage erläutert werden, wie die Entwicklung von Steuergerätesoftware bzw. embedded Software in der Automobilindustrie gehandhabt wird, nach welcher Methode IAV vorgeht und wo in diesem Prozess die vorliegende Arbeit sich eingliedert.

Aufgrund der exponentiellen Zunahme an Software-Komponenten im Fahrzeug und der stetig wachsenden Anforderungen an die Zuverlässigkeit, Verfügbarkeit und Sicherheit nebst Varianten- und Skalierbarkeitsanforderungen an jene Software wurden verschiedene Vorgehensweisen bzw. Methodiken und Standards entwickelt, um der steigenden Komplexität jener digitalen Systeme Rechnung zu tragen [SZ06].

Einer dieser Ansätze zur Überwachung des Entwicklungsprozesses ist das Capability Maturity Model Integration (CMMI), welches auf die Idee des ISO 9001 zurück geht, in den USA entwickelt wurde und eine Sammlung von bewährten Prinzipien zur Produktentwicklung darstellt. In Europa wurde über das EU-Förderprojekt BOOTSTRAP der ISO/IEC 15504 Standard entwickelt. Dieser ist heute unter dem Kürzel SPICE (Software Process Improvement and Capability Determination) bekannt [HSDZ⁺09]. Aus diesem Standard sind verschiedene domänen spezifische Ausprägungen hervorgegangen, insbesondere das für die Automobilbranche zugeschnittene *Automotive Spice*. Hier liegt der Fokus auf der Embedded-Software-Entwicklung und ermöglicht mit verschiedenen vorgegebenen Prozessen und Levels eine Überprüfung der Sicherheit und Zuverlässigkeit der entwickelten Software. Teil dieser Standards sind Vorgehensmodelle, wie das V-Modell,

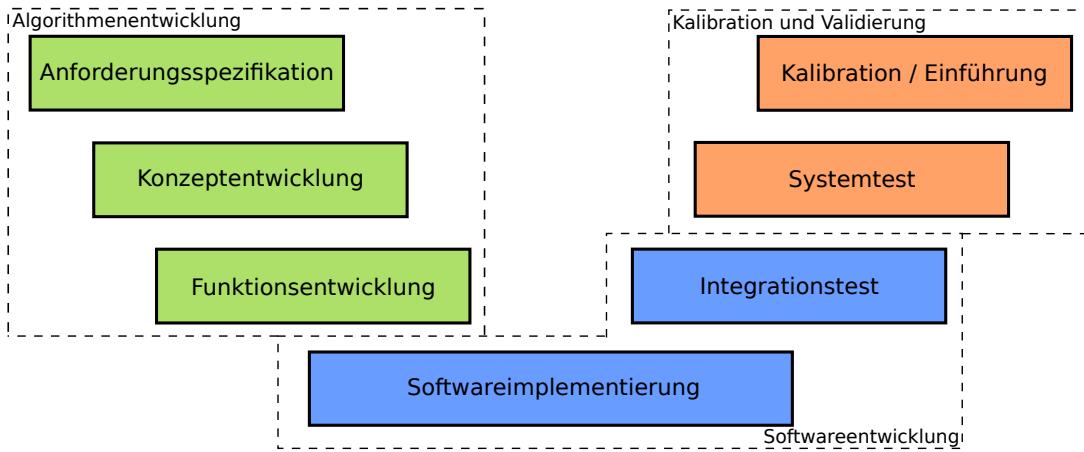


Abbildung 3: V-Modell der Softwareentwicklung

die Prozesse, wie die Softwareentwicklung in Phasen einteilen, zu denen auch entsprechende Testphasen zur Qualitätssicherung gehören. Abbildung 3 bildet das V-Modell ab, wie es auch bei IAV in der Funktionsentwicklung zum Einsatz kommt.

Neben den Prinzipien und Vorgehensweisen zur Überwachung und Strukturierung des Softwareentwicklungsprozesses existieren für die modellgetriebene Funktionsentwicklung weitere Richtlinien, die die Qualität, Codegenerierung und Wiederverwendbarkeit von Modellen sicherstellen sollen. Das MathWorks Automotive Advisory Board (MAAB) ist ein solches Dokument, welches Modellierungsrichtlinien zur Erstellung von Simulinkmodellen festlegt [maa12]. Es legt insbesondere Richtlinien für das Erscheinungsbild der Modelldiagramme sowie für die Verwendung der Funktionsblöcke fest. Tabelle 1 zeigt eine kleine Auswahl aus den MAAB-Richtlinien.

Durch immer kürzere Entwicklungszyklen und steigende Anforderungen hat sich die Entwicklung von Steuergeräte-Software auf Modellebene etabliert. Das Rapid-Prototyping ist aufgrund der schnellen und einfachen Erstellung von Signalfluss- bzw. Simulinkmodellen sowie der Codegenerierung einfach möglich. Darüber hinaus bietet ein solches Modell die Möglichkeit der Simulation der verschiedenen Funktionskomponenten. Zudem ist die Spezifikation auf abstrakter Ebene robust gegenüber Hardwareänderungen. Dies ist, aufgrund des langen Produktlebenszykluses eines Fahrzeugs von rund 25 Jahren verglichen mit dem deutlich kürzeren Änderungszyklus der Soft- und Hardware, ein entschiedener Vorteil [Men12, SZ06]. Nachteilig kann sich der Abstand zur realen Hardware jedoch aufgrund der aufwändigen Codegenerierungsprozesse auswirken.

Mit Hilfe dieser modellbasierten Entwicklung, den Richtlinien des MAAB und entlang des in Abbildung 3 dargestellten V-Modells wird bei IAV Steuergerätesoftware entwickelt. Wobei der Fokus der Entwicklung auf den Bereichen der Algorithmenentwicklung, Kalibration und Validierung liegt. Die Programmierung der Hardware anhand der in der Funktionsentwicklung erstellten Spezifikation findet im Bereich der Softwareentwicklung

Richtlinie	Beschreibung
na_0002	Blöcke, die numerische Operationen durchführen, dürfen nicht dazu genutzt werden, um logische Operationen auszuführen. Ergebnisse aus einer logischen Operationen sollen nicht direkt in einer numerischen Operation verarbeitet werden.
na_0027	Unternehmen sollen eine Untergruppe von Simulinkblöcken spezifizieren, die ausschließlich zur Modellierung genutzt wird. Davon unterschiedliche Blöcke sollen durch eine Farbe, Icon oder Anmerkung markiert werden. Diese sind vor der Codegenerierung zu entfernen.
db_0081	Ein System (Modell) darf keine nicht verbundenen Input- oder Output-Ports von Subsystemen oder Basisblöcken besitzen. Ein ansonsten nicht verbundener Port ist mit einem Ground- oder Terminatorblock zu verbinden.
jc_0201	Namen von Subsystemen dürfen nicht mit einer Ziffer beginnen und dürfen keine Leerzeichen und keine Zeilenumbrüche enthalten. Erlaubt sind nur Klein- und Großbuchstaben, Zahlen, sowie maximal ein Unterstrich, mit dem der Name nicht enden oder beginnen darf.

Tabelle 1: Beispiele für Modellierungsrichtlinien in Matlab/Simulink nach [maa12]

statt und wird häufig von den Herstellern der Steuergeräte ausgeführt. Ein Test der Funktionalitäten auf der Zielhardware ist also oft erst in der Phase der Systemtests möglich. Sollten in diesem Entwicklungsstadium Fehler entdeckt werden, ist ein Rücksprung in die Phase der Funktionsentwicklung notwendig. Dieser iterative Prozess wird fortgeführt, so lange die Systemtests nicht erfolgreich sind.

2.1.3 Methodik der Befehlsabarbeitung in CPUs

Bei der Analyse von Programmlaufzeiten spielt nicht nur die reine Berechnungszeit eine Rolle. Das Laden und Speichern von Daten sowie die parallele Abarbeitung (Superskalarität) von Operationen haben oftmals ebenfalls eine große Auswirkung auf die tatsächliche Ausführungszeit eines Programms.

In heutigen Prozessoren ist der Speicher, der Programme und Daten aufnimmt, in Hierarchien organisiert. Wobei diese Hierarchie so aufgebaut ist, dass sich sehr schnelle, aber kleine Speicherbausteine “nah” an der CPU (Central Processing Unit) befinden und langsamere aber deutlich größere Speicherkapazitäten im unteren Teil der Hierarchie zu finden sind. Dieser Aufbau stellt einen Tradeoff zwischen Zugriffsgeschwindigkeit und Kosten dar, denn je schneller die Zugriffsgeschwindigkeit auf ein Medium ist desto höher sind die Kosten für jenen Speicher. Abbildung 4 zeigt schematisch wie die einzelnen Speicher innerhalb eines Computers zusammenhängen.

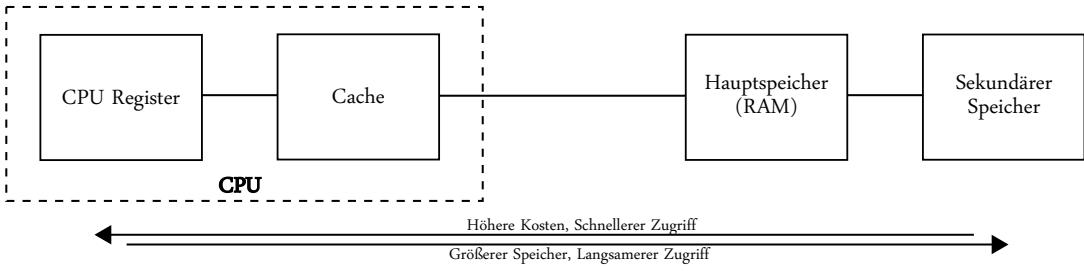


Abbildung 4: Speicherhierarchie nach [MF06]

Dieser Aufbau ermöglicht es der CPU durch den Cache auf Daten, die häufig genutzt werden, schnell zuzugreifen (zeitliche Lokalität). Außerdem nutzt ein Cache die räumliche Lokalität von Programmcode aus. Das bedeutet, dass nicht immer einzelne Bytes aus dem Hauptspeicher geladen werden, sondern größere Datenblöcke. Dies ist sinnvoll, da der Programmcode nicht verstreut im Speicher liegt und er überdies zumeist sequenziell abgearbeitet wird. So ist es sehr wahrscheinlich, dass die Daten, die sich in einem geladenen Datenblock befinden, in den nächsten Bearbeitungsschritten benötigt werden [MF06]. Caches sind häufig noch einmal in zwei Level unterteilt, wobei der Level 1 Cache mit dem Prozessortakt läuft und entsprechend schnell, dafür aber kleiner, ist. Sind Daten nicht im Level 1 Cache vorhanden, wird nach ihnen im Level 2 Cache gesucht. Dieser Vorgang nimmt im Vergleich zum Zugriff auf CPU Register deutlich mehr Prozessorzyklen und damit Zeit in Anspruch. Es zeigt sich also, dass die Ausführungszeit stark mit der Leistung des Caches bzw. einer entsprechenden Verdrängungsstrategie zusammenhängt [Rei08]. Hier zeigt sich außerdem bereits wie stark architekturbezogen *exakte* Laufzeitanalysen sein müssen. Überdies kommt hinzu, dass die Ausführung von Software mit verschiedenen Eingaben und initialem Hardwarezustand (Cachebelegung, Pipelinezustand) variiert. Eine weitere Hardwarekomponente, die die statische Laufzeitanalyse sowie die tatsächliche Laufzeit beeinflusst, ist das Pipelining.

Das Pipelining beschreibt eine Art Fließbandbearbeitung des Programmcodes. Hierbei wird die Abarbeitung von Maschinenbefehlen in einzelne Teilaufgaben unterteilt. Diese Teilaufgaben können wiederum gleichzeitig ausgeführt werden. Ein einzelner Befehl braucht nun zwar mehr Prozessorzyklen zur Abarbeitung, jedoch kann mit solch einer Pipeline der Durchsatz der Abarbeitung deutlich erhöht werden, da mehrere Befehle gleichzeitig bearbeitet werden. Typischerweise sind die Pipelinestufen in "Instruction Fetch", "Instruction Decoding", "Execution" und "Write Back" eingeteilt (vgl. Abbildung 5). Aufgrund des erhöhten Durchsatzes sinkt die durchschnittliche Programmlaufzeit und so zeigt sich, dass neben einem Cache auch eine Befehlsabarbeitung mit Pipeline Einfluss auf eine Laufzeitanalyse bzw. auf die tatsächliche Laufzeit nimmt [Lun02]. Eine Befehlspipeline wie mit den vier zuvor genannten Stationen, kann überdies in eine superskalare Struktur eingebettet werden. Superskalarität bezeichnet im Zusammenhang mit Prozessoren eine parallele Abarbeitung verschiedener Aufgaben. Wie so eine parallele Abarbeitung aussehen kann, ist in Abbildung 5 abgebildet. Hier können während

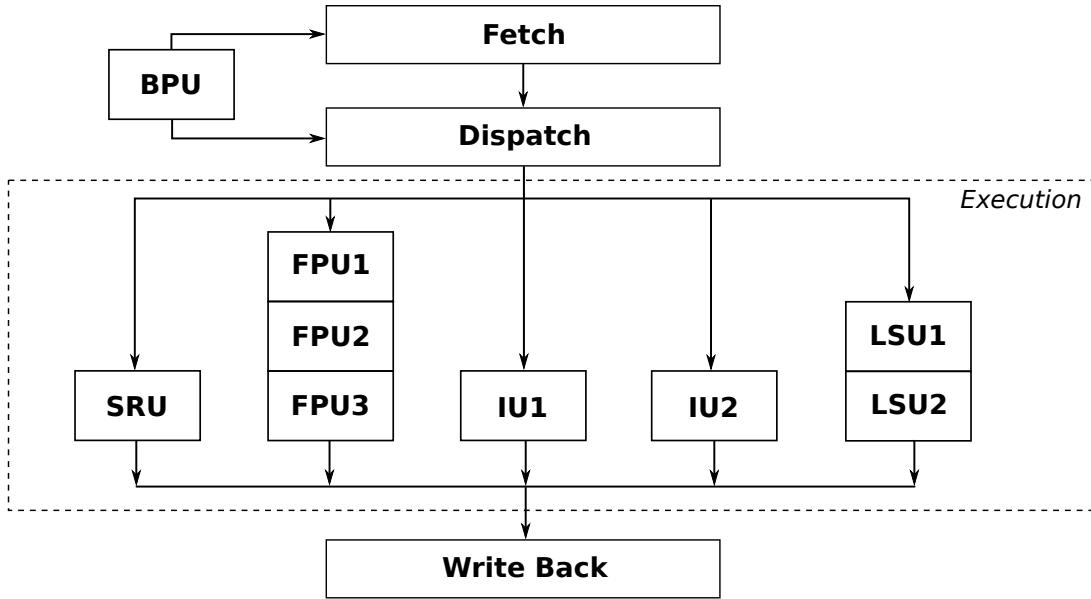


Abbildung 5: Aufbau der superskalaren Pipeline des IBM PowerPC 750GX

der Stufe der Ausführung (Execution) mehrere Befehle gleichzeitig bearbeitet werden. Dazu laufen die System Register Unit (SRU), Floating Point Unit (FPU), Integer Unit (IU) und Load/Store Unit parallel. Die FPU und LSU sind darüber hinaus wiederum als Pipelines organisiert, um einen höheren Durchsatz zu ermöglichen.

Eine weitere Komponente, die Einfluss auf die Laufzeitanalyse haben kann, ist die sogenannte Branch Processing Unit (BPU). Diese Einheit versucht mit entsprechenden Algorithmen Verzweigungen im Code vorherzusagen bzw. so aufzulösen, sodass nur die Befehle geladen und weitergegeben werden, die auch auf dem auszuführenden Pfad liegen. Der in Abbildung 5 dargestellte Ablauf verdeutlicht auf welche Weise sehr hardwarenah Programmcode umgesetzt wird und zeigt welche Maßnahmen ergriffen werden, um Leistungssteigerungen in der Programmausführung zu erzielen.

All diese Faktoren müssen für eine genau Abschätzung der Programmlaufzeit analysiert und in die Berechnung mit einbezogen werden, um entsprechend enge Grenzen für die Worst Case Execution Times (WCETs) berechnen bzw. abschätzen zu können [Lun02]. Um jedoch verlässliche Aussagen hinsichtlich des Laufzeitverhaltens zu treffen, sind genaue Kenntnisse über die Programm- bzw. Codestruktur vonnöten. Die folgenden Kapitel sollen einen Einblick in aktuelle Vorgehensweisen der Laufzeitanalyse geben und entsprechende Programme vorstellen.

2.1.4 Worst Case Execution Time Analysis

Die statische Ermittlung von Laufzeiten bzw. Abschätzung von WCETs ist seit einigen Jahren bereits Bestandteil vieler Forschungen im Bereich der eingebetteten- und Echt-

zeitsysteme [KLFP02, LM10, Lis14, Kir00, SEG⁺06, Lun02, Rei08]. In Kontrast zu dieser Arbeit wird in diesen Forschungen versucht Laufzeitaussagen anhand von Quellcode und Hardwarespezifikationen² zu treffen. Zumeist werden hier Assembler- oder Maschinensprachen analysiert, um eine entsprechende Aussagegüte zu erzielen. Im Folgenden werden einige Techniken zur Analyse von Softwareprogrammen dargestellt und erläutert.

Zunächst ist zu bemerken, dass eine Betrachtung der Laufzeit eines Programms nur dann sinnvoll sein kann, wenn es terminiert. Somit ist eine solche Betrachtung mit der Nichtentscheidbarkeit des Halteproblems³ verbunden. Dies bedeutet, dass keine Laufzeitanalyse bzw. kein Algorithmus existiert, der für beliebige Programme eine Laufzeitschranke berechnen kann. Aus diesem Grund soll sich im Folgenden die Betrachtung auf Programme beschränken, zu denen ausreichende Informationen hinsichtlich der Termination vorliegen [Zöb08, LM10].

Grundsätzlich lässt sich die WCET-Analyse in zwei Bereiche aufteilen: Zum einen in die *statische* und zum anderen in die *dynamische* bzw. messbasierte Laufzeitanalyse. Die statische Analyse berechnet Laufzeitschranken anhand des Quellcodes, sowie hardwarespezifischer Faktoren wie Caching oder Pipelining. Dazu ist eine Ausführung des Programmes nicht notwendig. Die dynamische Analyse misst während der Ausführung des Programms auf der Hardware die Ausführungszeit. Diese Form der Laufzeitmessung wird im zweiten Teil dieses Kapitels ausführlicher erläutert, wobei folgend die statische Analyse Betrachtung findet.

Statische Laufzeitanalyse

Die Ausführungszeit von Programmen wird bestimmt durch den Programmfpad und die entsprechenden Operationen, die auf diesem Pfad auszuführen sind. Dieser Ablauf eines Programmes wird auf Ebene der Programmiersprache, bei eingebetteten Systemen handelt es sich hier überwiegend um Assemblersprachen, Maschinencode oder auch C-Code, in einzelne *Basisblöcke* unterteilt. Ein Basisblock besteht hierbei aus aufeinander folgenden Operationen, die nacheinander ohne Verzweigung abgearbeitet werden können. Abbildung 6 stellt ein einfaches Beispiel für ein Programm auf der linken und den entsprechenden Kontrollflussgraphen auf der rechten Seite dar. Die Konstruktion eines Graphen anhand des Quell- oder Binärcodes stellt häufig den ersten Schritt in der statischen Laufzeitanalyse dar. Mit Hilfe des Graphen und mittels ganzzahliger linearer Optimierung (integer linear programming; ILP) lässt sich der längste Pfad in diesem Kontrollflussgraphen berechnen. Der längste Pfad korrespondiert in diesem Fall mit dem Weg der längsten Ausführung. Um eine enge obere Laufzeitschranke berechnen zu können, ist es wichtig, neben den reinen Berechnungs- bzw. Zuweisungsoperationen mögliche Cache- oder Pipelinebelegungen zu betrachten. Dementsprechend ist die längste Ausführungszeit

²Bus- und Prozessortakte, (Cache-)Speichergröße, Länge der Pipeline, Befehlssatz etc.

³Alan Turing bewies in den 1930er Jahren, dass es keinen Algorithmus gibt, der für ein beliebiges gegebenes Programm entscheidet, ob es (für alle Eingaben) hält [Zöb08].

```

1 stop = k = a = 0
2 e = 0.000001
3 q = 0.5
4 while (stop != 1):
5     /*rte: loop
6     MAX 20; */
7     a = a + q**k
8     if (2 - a <= e):
9         /*rte: flow
10        == 1; */
11        stop = 1
12    else:
13        k = k + 1
14 s = k

```

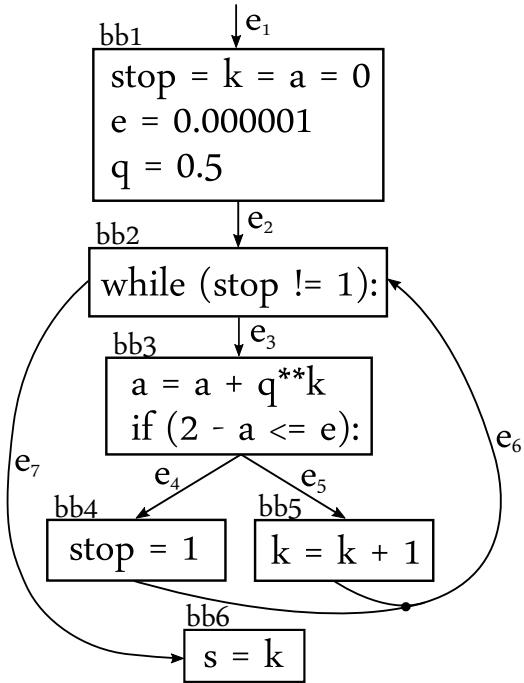


Abbildung 6: Einfaches Quellcodebeispiel und dazugehöriger Kontrollflussgraph

$a_i = WCET(bb_i, c_i)$ ebenfalls abhängig vom Hardwarekontext c_i in dem der Basisblock bb_i ausgeführt werden kann. Um diesen Kontext für jeden Knoten im Kontrollflussgraphen darzustellen ist eine (abstrakte) Modellierung des Cache- und Pipeliningverhaltens vonnöten⁴. Indem man beim Traversieren des Graphen den Cache sowie die Pipeline simuliert, kann jedem Programmfpunkt bzw. Basisblock eine Liste möglicher Cache-Inhalte und Pipelinebelegungen approximiert bzw. simuliert werden, dabei wird auch von “Abstract Interpretation” gesprochen [FW97, HF06, LM10, MPPU10].

Ausgehend von einem Programm mit N Basisblöcken, wobei jeder Basisblock bb_i eine Ausführungsduer von a_i hat und maximal x_i mal ausgeführt wird, ergibt sich die WCET wie folgt.

$$WCET = \sum_{i=1}^N a_i \cdot x_i = \sum_{i=1}^N WCET(bb_i, c_i) \cdot x_i \quad (1)$$

Oft ist die Anzahl an Ausführungen pro Basisblock nicht explizit bekannt, sondern steht in impliziter Beziehung mit den Ausführungen anderer Basisblöcke [Zöb08]. Demzufolge ergeben sich für das ILP-Problem, neben der Kontextabhängigkeit, weitere strukturelle Nebenbedingungen. Die Anzahl an Ausführungen mit entsprechenden Abhängigkeiten sind für das Beispielprogramm aus Abbildung 6 in den Gleichungen in (2) und (3) angegeben.

⁴Neben dem Cache und der Pipeline können weitere Hardwareeigenschaften untersucht werden, wie zum Beispiel “Branch-Prediction” [BR06] oder Mehrkernprozessoren [Kel15]

$$\begin{array}{lll}
 x_1 = e_2 = e_1 = 1 & & \\
 x_2 = e_2 + e_6 = e_3 + e_7 & & x_1 = 1 \\
 x_3 = e_4 + e_5 = e_3 & (2) & x_2 = 1 + e_6 = x_3 + x_6 \quad (3) \\
 x_4 = e_4 & & x_3 = x_4 + x_5 = e_6 \\
 x_5 = e_5 & & \\
 x_6 = e_7 & &
 \end{array}$$

Abbildung 7: Strukturelle Nebenbedingungen für den Graphen aus Abbildung 6

Bereits für einfache Programme zeigt sich ein entsprechender Umfang der statischen WCET-Analyse. Neben der Abhängigkeit vom Hardwarekontext, sowie gewissen strukturellen Bedingungen, können sich überdies weitere funktionale Beschränkungen aus der Programmstruktur ergeben. Zur Lösung des ganzzahligen Optimierungsproblems sind diese funktionalen Beschränkungen essentiell. Selbst bei der Annahme jedes a_i lässt sich exakt bestimmen, so kann nur über die strukturellen Bedingungen die Anzahl an Ausführungen x_i nicht bestimmt werden. Die Ursache hierfür ist, dass zum einen keine Informationen zur maximalen Anzahl an Durchläufen der While-Schleife vorhanden sind und zum anderen nicht bekannt ist, wie oft der Programmablauf verzweigt wird, also wie oft bb_4 und bb_5 ausgeführt werden. Da es häufig nicht realisierbar ist diese Informationen algorithmisch zu bestimmen, werden sie durch den Programmierer durch Anmerkungen (Annotations) im Programmcode definiert. Für das Beispiel aus Abbildung 6 könnten die funktionalen Beschränkungen “die While-Schleife wird maximal 20 mal durchlaufen” ($x_3 \leq 20$) und “Basisblock bb_4 wird nur einmal ausgeführt” ($x_4 = 1$) sein. Damit ergibt sich das ILP-Problem wie in Gleichung (4) dargestellt.

$$\begin{aligned}
 WCET = \max \Big\{ & \sum_{i=1}^N WECT(bb_i, c_i) \cdot x_i \Big| \sum_{j \in in(bb_i)} e_j = \sum_{k \in out(bb_i)} e_k = x_i, i = 1 \dots N \wedge \\
 & x_1 = 1 \wedge x_2 = 1 + e_6 = x_3 + x_6 \wedge x_3 = x_4 + x_5 = e_6 \wedge x_3 \leq 20 \wedge x_4 = 1 \Big\} \quad (4)
 \end{aligned}$$

Neben Annotations wie “Loop Bounds” oder “Flow Constraints” sollten außerdem Anmerkungen bei Rekursion zur Rekursionstiefe gegeben sein, oder sogenannter “toter Code” (z.B. bei Codegenerierung) aus der Berechnung ausgeschlossen werden. Wichtig zu bemerken ist hier jedoch, dass Anmerkungen, wie sie beispielhaft in Abbildung 6 als Kommentare im Quellcode angegeben sind, keine Verwendung finden, denn diese Informationen werden vom Compiler ignoriert und gehen somit verloren. Es sei denn, der Compiler ist entsprechend für eine solche Code-Erweiterung angepasst (vgl. Kapitel 2.2.2). Infolgedessen werden Code-Anmerkungen erst auf Binärebene angegeben, wo sie dann von einem Analysewerkzeug interpretiert werden können. Es zeigt sich folglich, dass in den meisten Fällen die statische Laufzeitanalyse nicht vollkommen automatisch und ohne explizite Angaben zum Programmkontext auskommt. Sind diese Angaben jedoch gewissenhaft angegeben, kann mit ausreichendem Wissen über die Zielplattform

(Hardwarespezifikation), eine entsprechend enge obere Laufzeitschranke berechnet werden. Darauf hinaus lassen sich die Nebenbedingungen leicht generieren bzw. angeben und ein solches Optimierungsproblem mit existierenden Tools⁵ lösen. Das Lösen eines ILP-Problems ist jedoch NP-Schwer und damit für größere Programme mit einer längeren Berechnungsdauer verbunden.

Messbasierte Laufzeitanalyse

Die messbasierte Laufzeitanalyse ist ein weiterer Ansatz zur Bestimmung der Laufzeit eines Programmes. Im Kontrast zur zuvor vorgestellten statischen Analyse wird für den messbasierten Ansatz neben einer Kompilierung des Quellcodes ebenfalls die Zielhardware benötigt. Dabei wird die Software auf der Hardware ausgeführt und dort die Laufzeit gemessen. Dazu stehen je nach Aufbau der Hardware verschiedene Möglichkeiten zur Auswahl. Beispielsweise kann über entsprechende Operationen in der Software ein digitaler Ausgang an der CPU bei Beginn und Ende der Ausführung „getoggelt“ werden. Mit Hilfe eines Oszilloskops ist dann die Ausführungszeit zu messen. Eine weitere Möglichkeit eine Laufzeit festzuhalten, ist das Nutzen eines Timers in der CPU. Diese Timer können über entsprechende Assemblerbefehle gesteuert werden und bieten eine einfache Möglichkeit Prozessortakte während der Ausführung eines Programmes zu zählen. Jedoch findet sich eine solche Implementierung nicht in jeder CPU. Programme wie „RapiTime⁶“ bieten darüber hinaus Toolkits an, die automatisiert Performancemessungen ausführen, sowie die ermittelten statistischen Daten auswerten und visualisieren.

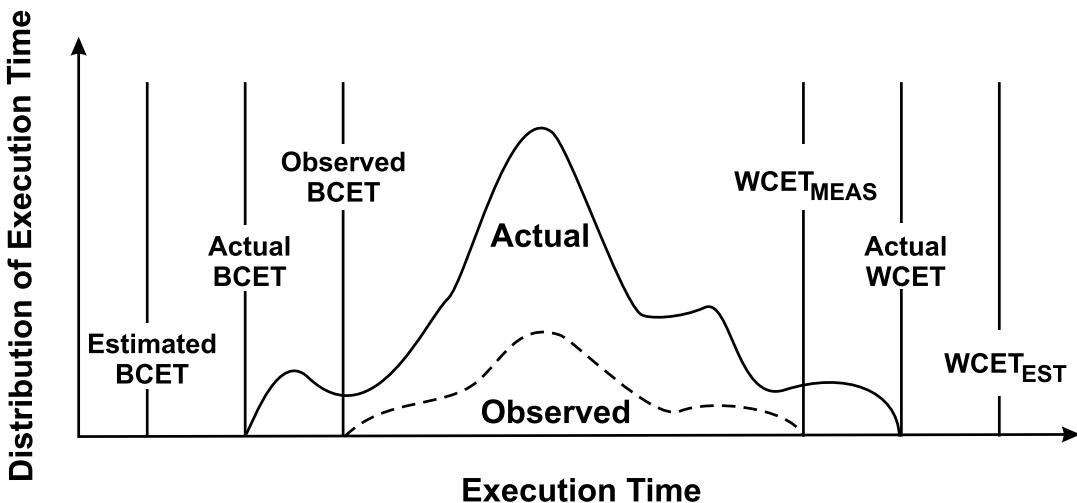


Abbildung 8: Laufzeitverteilungen aus [LM10]

Der Ansatz, durch das reine Messen der Ausführungszeit die tatsächliche WCET sicher zu ermitteln, ist jedoch für den allgemeinen Fall sowie für komplexere Software, wie sie im Automobilumfeld gefunden wird, äußerst aufwändig und impraktikabel. Da

⁵GNU Linear Programming Kit (GLPK), LP_Solve, Coin-OR Branch and Cut (CBC) etc.

⁶www.rapitasyystems.com

der längste Ausführungspfad des Programmes nicht bekannt ist, ist durch verschiedene Eingabeparameter zu testen, inwieweit diese die Laufzeit beeinflussen. Eine erschöpfende Suche dieses Pfades ist jedoch aufgrund des zu umfangreichen Zustandsraums für Software auf modernen Prozessoren nicht realisierbar [LM10]. In Abbildung 8 ist eine mögliche Laufzeitverteilung aufgetragen. Ferner sind die einzelnen Schranken verschiedener Ausführungen dargestellt. Dabei korrespondiert $WCET_{EST}$ zur geschätzten bzw. berechneten oberen Schranke und $WCET_{MEAS}$ zur längsten gemessenen Laufzeit. Für nicht erschöpfendes Suchen kann sich eine ähnliche Laufzeitverteilung entstehen, wobei sich häufig ergibt, dass $WCET_{MEAS} < WCET_{actual}$ ist.

Folgend werden Programme zur Laufzeitanalyse vorgestellt, die die zuvor genannte Theorie der statischen Laufzeitanalyse einsetzen, um entsprechend enge obere Zeitgrenzen berechnen zu können.

2.2 Stand der Technik

In diesem Kapitel werden (kommerzielle) Programme vorgestellt, die eine Laufzeitanalyse für bestimmte Architekturen, Compiler und Programmiersprachen ausführen. Ferner werden zwei weitere Ansätze zur Untersuchung der Laufzeit dargestellt. Kapitel 2.2.2 stellt eine Möglichkeit der Laufzeitanalyse in Zusammenarbeit mit Simulink vor und erläutert das Konzept des sogenannten “WCET-Aware-Compilers”.

2.2.1 WCET - Tools

Neben kommerzieller Software zur Bestimmung von WCET's bzw. Abschätzung von Laufzeiten gibt es eine ganze Reihe weiterer prototypische Programme (Heptane⁷, Tu-Bound⁸, SymTA/P⁹ etc.), die in der Forschung eingesetzt werden. Folgend sollen exemplarisch ein kommerzielles sowie zwei Forschungsprogramme vorgestellt werden.

aiT

Das “aiT WCET Analyzers”-Programm ist ein kommerzielles Werkzeug, dass von der Firma “AbsInt Angewandte Informatik GmbH”¹⁰ entwickelt wird. Es ist Teil einer Toolkette, die neben der statischen Laufzeitanalyse aus einem Quellcode-Checker, einem formal verifizierten Compiler sowie einem Stack-Analysierer und einem Timing-Profiler besteht. Diese Toolkette soll Entwicklern von sicherheitskritischen Systemen während der gesamten Entwicklungszeit die Möglichkeit bieten, ihre Funktionalitäten zu verifizieren und zu testen.

⁷Institut national de recherche en informatique et en automatique (INRIA)

⁸Technische Universität Wien

⁹Technische Universität Braunschweig

¹⁰www.absint.com

Um ein Programm mit aiT analysieren zu können, muss dieses im sogenannten Executable and Linking Format (ELF) vorliegen. Die Berechnung einer oberen Zeitschranke für das gegebene Programm wird dann von aiT nach den in Kapitel 2.1.4 beschriebenen Vorgehensweisen berechnet. Zunächst wird aus dem ELF-Programm ein Kontrollflussgraph konstruiert und darauf aufbauend eine Analyse der Speicherzugriffe, Cache-Analyse und Pipeline-Analyse erstellt. Mit den in diesem Prozess gewonnenen Informationen sowie weiteren Angaben des Entwicklers zum Programmkontext wird das Optimierungsproblem (ILP) erstellt und mit einem entsprechenden “LP Solver” gelöst [HF06].

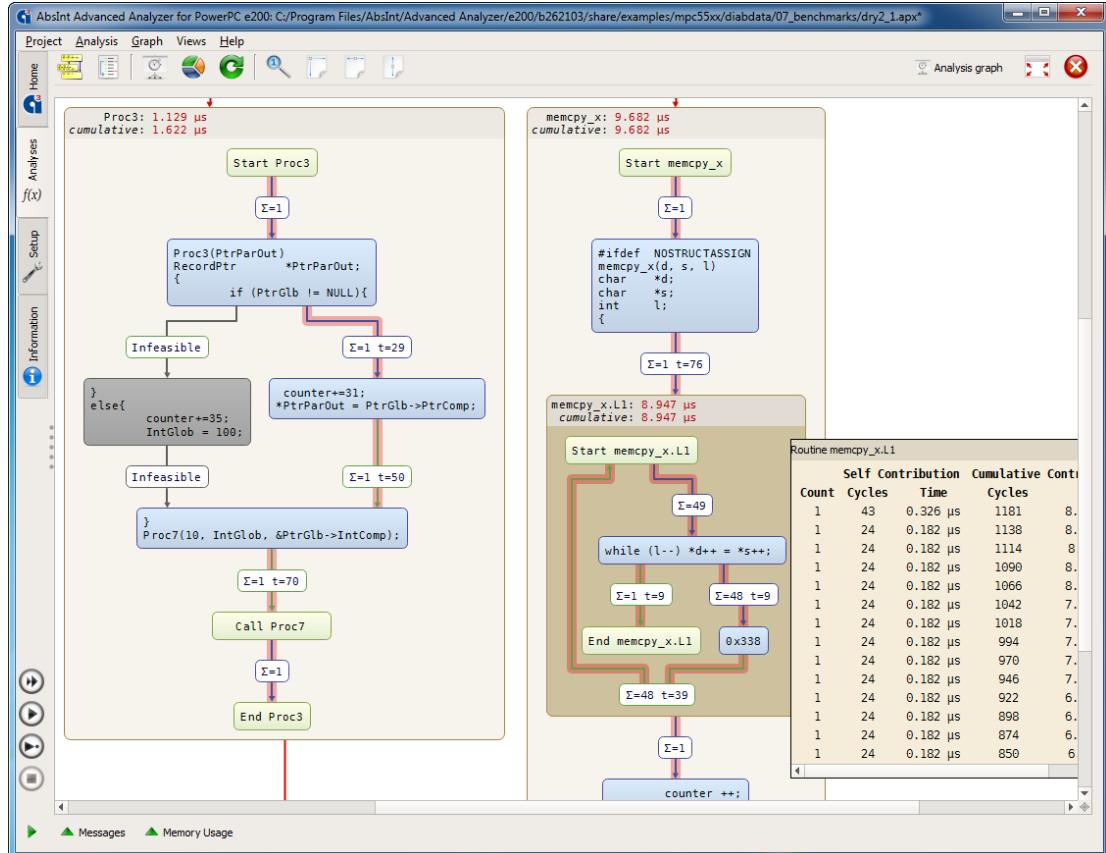


Abbildung 9: Kontrollflussgraph mit WCET Ergebnissen in aiT

Neben einer engen oberen Schranke für die Laufzeit des Programmes ermöglicht aiT außerdem die Visualisierung der Analyseergebnisse. Abbildung 9 zeigt einen beispielhaften Kontrollflussgraphen, sowie die entsprechenden WCET Ergebnisse zu den einzelnen Prozeduren. Detaillierte Informationen zu zeitlichen Schlüsselaspekten, wie beispielsweise der “Worst-Case”-Pfad durch das Programm oder der Hardwarezustand zu jedem gegebenen Programmpunkt, können so eingesehen und nachvollzogen werden. Des Weiteren kann aiT nahtlos in verschiedene weitere Prozessketten zur Entwicklung von Software für sicherheitskritische und eingebettete Systeme integriert werden. Hier sind insbesondere die SCADE (Safety Critical Application Development Environment) sowie ASCET (Advanced Simulation and Control Engineering Tool) zu nennen. Die Kombination von aiT und SCADE bzw. ASCET ermöglicht WCET-Analysen bereits auf Modellebene, indem

aus dem graphischen Modell Quellcode generiert wird, der für eine entsprechende Architektur kompiliert und anschließend von aiT analysiert wird [FHW⁺06]. Die Analyseergebnisse werden dann von aiT zurück an die SCADE Suite gegeben und dort entsprechend angezeigt¹¹. Dieser Prozess ist völlig automatisiert und geschieht im Hintergrund.

Diese Software ermöglicht in Zusammenarbeit mit graphischen Entwicklungsumgebungen bereits auf Modellebene sehr gute WCET-Berechnungen. Die Auswertung und Berechnung findet jedoch weiterhin auf Binärebene statt und bedient sich nicht alleinig den aus dem Modell gewonnenen Informationen. Sie benötigt weiterhin eine Codegenerierung und Kompilierung und ist dementsprechend aufwändig. Die in dieser Arbeit entwickelte Toolkette macht eine Codegenerierung und Kompilierung bei jeder Analyse überflüssig. Durch die Verwendung einer zuvor aufgebauten “Laufzeittabelle” sowie Informationen aus dem Modell lässt sich mit geringem Aufwand und früh in der Entwicklung eine erste Laufzeitanalyse durchführen.

SWEET

Das Swedish Execution Time Tool (SWEET) ist ein Forschungsprogramm des Mälardalen Real-Time Research Centre (Schweden)¹². Es berechnet WCETs bzw. BCETs (Best Case Execution Time) für verschiedene Architekturen. Die Hauptfunktionalität besteht in der Ermittlung von sogenannten “Flow Facts”. Diese geben beispielsweise Informationen über maximale Schleifendurchläufe oder mögliche bzw. unmögliche Pfade im Programmfluss an. Das Ziel des Programms ist solche “Flow Facts”, insbesondere “Loop Bounds”, automatisch zu berechnen. Wie bereits in den vorangegangenen Kapitel erläutert, sind diese Informationen äußerst wichtig, um enge Zeitschränken berechnen zu können. SWEET arbeitet jedoch nicht direkt auf dem Quellcode oder Binärcode, sondern auf dem sogenannten ALF Format (ARTIST2 Language for WCET Flow Analysis). Der Quellcode wird vor der Analyse von entsprechenden Konvertern in jenes Format, das speziell für die Laufzeitanalyse entwickelt wurde, übersetzt [Lis14].

Die WCET-Analyse muss sehr genau auf viele verschiedene Plattformen und Programmiersprachen jeweils zugeschnitten werden. Die Motivation hinter der Verwendung des ALF-Formats ist, dieses als Zwischensprache zu verwenden, um so nur noch eine einzige Analyse auf Basis des ALF-Formats implementieren zu müssen. SWEET versucht nun anhand dieses Formats möglichst viele “Flow Facts” zu ermitteln, mit denen dann über die genannten Algorithmen möglichst enge Zeitschränken berechnet werden können. Darüber hinaus ist es außerdem möglich, die ermittelten Informationen als Eingabe für aiT zu benutzen, um dort mit ihnen entsprechende WCETs/BCETs zu berechnen [Gus16].

¹¹www.absint.com/ait/scade.htm

¹²www.mrtc.mdh.se/projects/wcet/sweet

Chronos

Chronos ist ein WCET-Werkzeug der National University of Singapore (NUS). Es nutzt ebenfalls den Ansatz der statischen Laufzeitanalyse zur Berechnung enger Laufzeitschranken. Chronos zeichnet sich dadurch aus, dass es als reine Forschungssoftware konzipiert wurde und aus diesem Grund keine kommerziellen Prozessorarchitekturen adressiert. Die gesamte Analyse basiert auf der frei verfügbaren Software *SimpleScalar*. Diese Software ist ein sogenannter “Cycle-Accurate Simulator” und bietet die Möglichkeit Hardwarchitekturen und Prozesse in Software zu modellieren. SimpleScalar erlaubt es verschiedene Cache-Strategien, speculative execution und branch prediction sowie superskalare Prozessoren zu emulieren. Damit ein C-Programm analysiert werden kann, ist es zunächst für die SimpleScalar Architektur zu kompilieren. SimpleScalar simuliert dann das Binärprogramm anhand eines definierten Prozessormodells, wobei verschiedene Instruction Sets, wie PISA, ARM oder x86 emuliert werden können. Damit bietet Chronos neben der statischen Analyse, die Möglichkeit ein Programm zu simulieren und aus dieser Simulation entsprechende Laufzeitinformationen zu gewinnen [LLMR07].

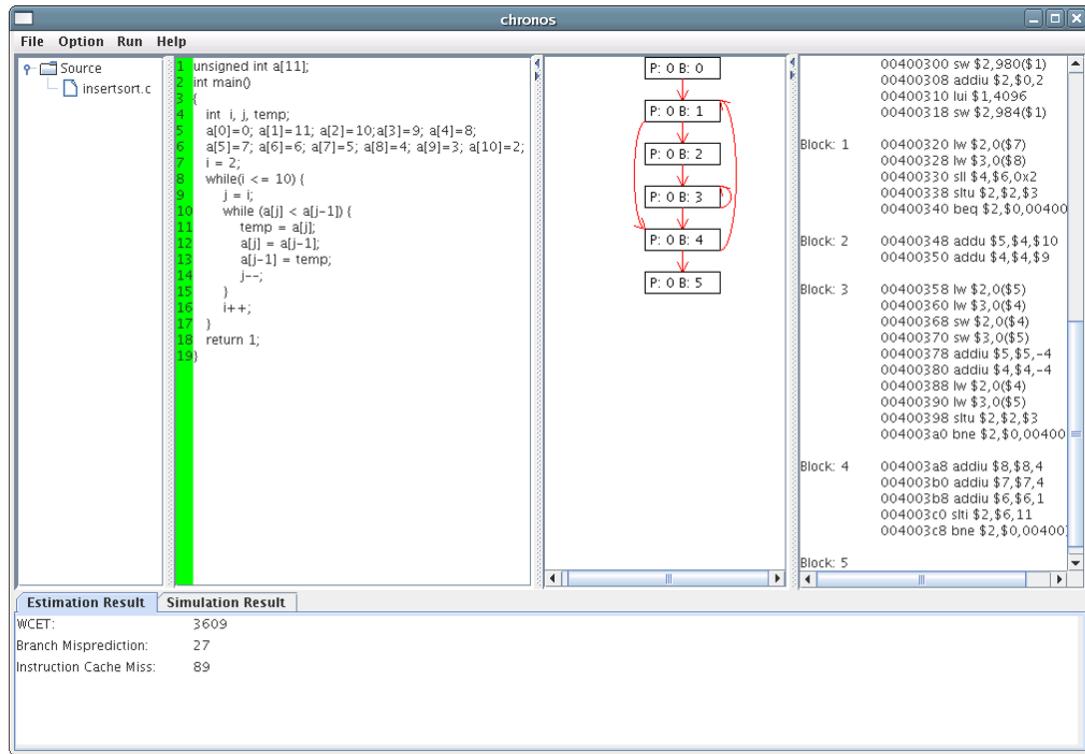


Abbildung 10: Berechnung des WCET mit Chronos [LLMR07]

Die statische Laufzeitanalyse ist der Analyse, wie sie auch bei aiT umgesetzt wird, sehr ähnlich. Nach der Kompilierung des C-Programmes (GCC für SimpleScalar) wird durch eine Pfad-Analyse der Kontrollflussgraph aufgebaut, sowie strukturelle Nebenbedingungen abgeleitet. Diese Bedingungen können durch den Nutzer ebenfalls erweitert

werden. Dariüber hinaus werden von Chronos Basisblöcke identifiziert, deren Ausführung unabhängig von Programmeingaben ist. Die Ausführung dieser Blöcke wird anhand eines vom Nutzer konfigurierbaren Prozessormodells von SimpleScalar simuliert und die Ausführungszeit in die Berechnung mit einbezogen. Anhand der von SimpleScalar genutzten Prozessor-Konfiguration werden ferner durch “microarchitecture modeling”, welches Branch Prediction-, Instruction Cache- und Pipeline-Analyse umfasst, weitere Nebenbedingungen abgeleitet. All diese Information werden abschließend zu einem ILP-Problem zusammengefasst und mit “LP_Solve” gelöst [LLMR07].

In Abbildung 10 ist die Nutzung der Software anhand eines Programmes (Insertion Sort) exemplarisch dargestellt. Neben dem C-Code wird ebenfalls der Kontrollflussgraph sowie der Assembler-Code dargestellt.

2.2.2 WCET-Analyse für Simulink Modelle mit wcetC

Aus den vorangegangen Kapiteln wird ersichtlich, dass Compiler eine essentielle Rolle in der Laufzeituntersuchung spielen. Infolge des hohen Optimierungspotentials durch Compiler, sind diese als effektives Werkzeug für eine immer präzisere Laufzeitanalyse vielversprechend. Folgend soll die Idee eines “WCET-Aware C Compilers” (WCC) sowie einer angepassten Programmiersprache “wcetC” vorgestellt werden. Darüber hinaus findet eine Methode der statischen Laufzeitanalyse in Verbindung mit der Codegenerierung in Simulink Betrachtung (siehe Abbildung 11).

Wie bereits in Kapitel 2.1.4 angemerkt, sind bei der statischen Laufzeitanalyse häufig Laufzeitinformationen (Annotations) durch den Nutzer bzw. Programmierer im Programmcode zu hinterlegen, um das entsprechende Optimierungsproblem vollständig zu definieren. Zudem ist es wichtig die Analyse auf einer hardwarenahen Programmerebene durchzuführen, um möglichst präzise Aussagen bezüglich der Laufzeit treffen zu können. Für die statische Laufzeitanalyse wird daher das Binärprogramm zur Untersuchung herangezogen. Erst auf dieser Ebene sind Code-Annotations möglich, falls der Compiler nicht entsprechend angepasst ist und durch eine Erweiterung die bereits auf Quellcode-Ebene angegebenen Laufzeitinformationen auf die Binärebene überträgt. Andernfalls kann es für den Programmierer recht aufwändig sein, Quellcode-Strukturen im Binärprogramm zu identifizieren. Ferner kann eine Optimierung des Compilers hinsichtlich des WCETs sinn-

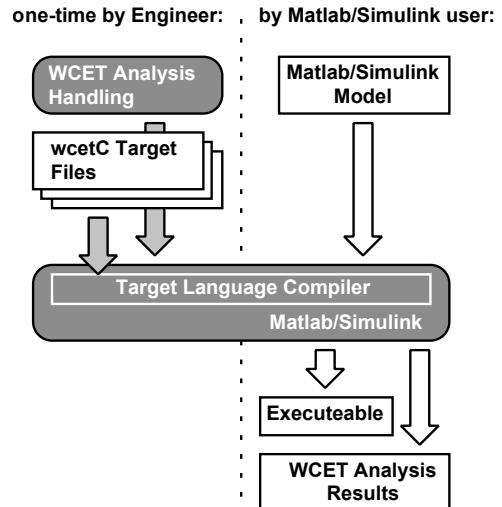


Abbildung 11: Entwicklungsprozess für Simulink mit wcetC aus [KLFP02]

voll sein. Gegenwärtig wird ausschließlich eine Optimierung des ACETs (Average Case Execution Time) erreicht, da dem Compiler keine Laufzeitinformationen der Zielarchitektur zur Verfügung stehen. Eine Umsetzung eines WCC wird in [LM10] sowie [FL10] vorgestellt. Dabei wird eine Erweiterung der ANSI C-Programmiersprache (wcetC) genutzt, um Laufzeitinformation bereits auf Ebene einer Hochsprache einzubetten [Kir02].

Die Überlegung die Kompilierung mit entsprechenden Laufzeitinformationen anzupassen ist in [KLFP02] mit der Codegenerierung von Simulink Modellen verbunden worden (vgl. Abbildung 11). Dabei wurde der Target Language Compiler (TLC), welcher für die Codegenerierung zuständig ist, so angepasst, dass Informationen aus dem Simulink Modell als Laufzeitinformationen im generierten Code eingebettet werden und somit wcetC-Code erzeugt wird. Durch diese einmalige Anpassung der Codegenerierung, ist der Funktionsentwickler davon befreit selber auf Binärebene eigene Anmerkungen im Code einzupflegen, da diese aus dem Simulink Modell abgeleitet werden können. Darüber hinaus ist es dem Entwickler trotzdem möglich eigene Annotationen auf Modellebene für einen Funktionsblock anzugeben. Der generierte wcetC Code kann anschließend mit einem WCC kompiliert und optimiert werden.

Anhand des kompilierten (und annotierten) Binärprogramms kann die statische Laufzeitanalyse durchgeführt und die Laufzeitschranken bestimmt werden. Als weiteren Ansatz wird in [KLFP02] die sogenannte “Back-Annotation” vorgestellt. Dabei ist während der Codegenerierung für jeden Funktionsblock im C-Code ein Start- bzw. End-Block einzufügen, um so die einzelnen Funktionalitäten pro Block von einander abzugrenzen. Diese Informationen werden über die Kompilierung hinweg erhalten und später in der Laufzeitanalyse entsprechend ausgewertet. So ist es möglich jedem Simulink Block eine eigene Laufzeit zuzuweisen. Um diese Ergebnisse zu visualisieren, werden die Daten mit Hilfe einer eingebetteten Funktion in Simulink (S-Function¹³) ausgelesen und für jeden Funktionsblock entsprechend angezeigt.

¹³Eine S-Function ist ein Simulink Funktionsblock für den individuell Quellcode hinterlegt werden kann

3 Anforderungsanalyse

Dieses Kapitel erläutert die Anforderungen an die zu entwickelnde Toolkette zur Laufzeitanalyse. Es greift die in den Grundlagen genannten Aspekte auf und erläutert den aus den Randbedingungen und der Zielsetzung entwickelten Anforderungskatalog. Dieser Katalog entsteht im Folgenden aus der Analyse der Entwicklungstoolchain unter Berücksichtigung der Zielsetzung, welche sich aus der Aufgabenstellung ergibt, sowie entsprechenden Randbedingungen.

3.1 Entwicklungstoolchain

Kapitel 2.1.2 hat bereits einen Einblick in die Softwareentwicklung im Automotiveumfeld gegeben und dabei vorgestellt, inwieweit eine Trennung zwischen der Algorithmenentwicklung, Softwareimplementierung und Kalibration bzw. Validierung besteht. Hinsichtlich dieser Trennung (vgl. Abbildung 3) kann eine erste Laufzeitabschätzung bereits frühzeitig im Entwicklungsprozess, also zum Zeitpunkt der Modellentwicklung, vorteilhaft sein. Einen weiteren Mehrwert bietet die Toolkette bei Neuentwicklungen, bei denen noch keine Erfahrungswerte hinsichtlich der Hardware vorhanden sind und ebenso bei Modularerweiterungen, um abschätzen zu können in welchem Umfang Erweiterungen möglich sind. Da in einigen Fällen eine entsprechende Toolkette zur Implementierung der Software auf dem jeweiligen Steuergerät nicht zur Verfügung steht, ist eine Laufzeitanalyse, die ohne Kompilierung des generierten Codes und ohne Hardware auskommt, gewünscht. Die Anforderung besteht somit darin vor der Phase der Softwareimplementierung, auf Modellebene, eine erste Schätzung der Laufzeit der gegebenen Funktionalität vorzunehmen. Abbildung 12 veranschaulicht wo sich die Laufzeitanalyse im Ablauf eines vom

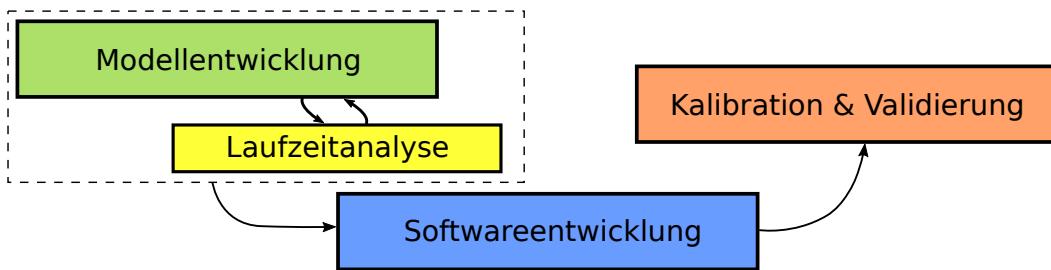


Abbildung 12: Vereinfachtes V-Modell mit Laufzeitanalyse (vgl. Abbildung 3)

V-Modell definierten Entwicklungsprozesses eingliedert. Wie im V-Modell vorgesehen, ist auch zwischen Modellentwicklung und Laufzeitanalyse eine Verifikation vorgesehen: Sind entsprechende Designentscheidungen hinsichtlich der Laufzeit als sehr Zeitaufwändig identifiziert, so ist gegebenenfalls das Modell anzupassen.

Da die Laufzeitanalyse auf Modellebene stattfinden soll, ist eine Betrachtung der bereits bei der IAV entwickelten Simulink-Modelle, deren Umfang sowie eine Betrachtung

der adressierten Architekturen sinnvoll. Ein Aspekt ist bei der Betrachtung der Modelle besonders auffällig: Bei allen¹⁴ untersuchten Simulink Modellen beschränkt sich der Aufbau des Modells auf 31 Basis-Simulinkblöcke¹⁵. Selbst sehr komplexe Modelle von mehr als 1200 Blöcken weisen die gleiche Verteilung an Funktionsblöcken auf. Dies resultiert aus der Einhaltung der MAAB-Vorschriften, die die Verwendung der Simulink Standardblöcke vorschreiben. Somit kann die Entwicklung eines entsprechenden Werkzeugs fokussiert auf die Verwendung dieser Simulink-Bibliothek ausgerichtet werden. Es ist jedoch zu beachten, dass obwohl einige Funktionsblöcke nur sehr selten Verwendung finden, sie jedoch aufgrund einer möglichen komplexen Funktionalität dennoch bedeutend zur gesamten Laufzeit eines Programmes beitragen können. Abbildung 13 stellt die durchschnittliche Verteilung der untersuchten Funktionsblöcke pro Modell dar. Funktionsblöcke, die weder zum Signalfluss noch zur Berechnung oder Codegenerierung benötigt werden, sind aus dieser Betrachtung bereits ausgeschlossen¹⁶. Auffällig ist bei der Vertei-

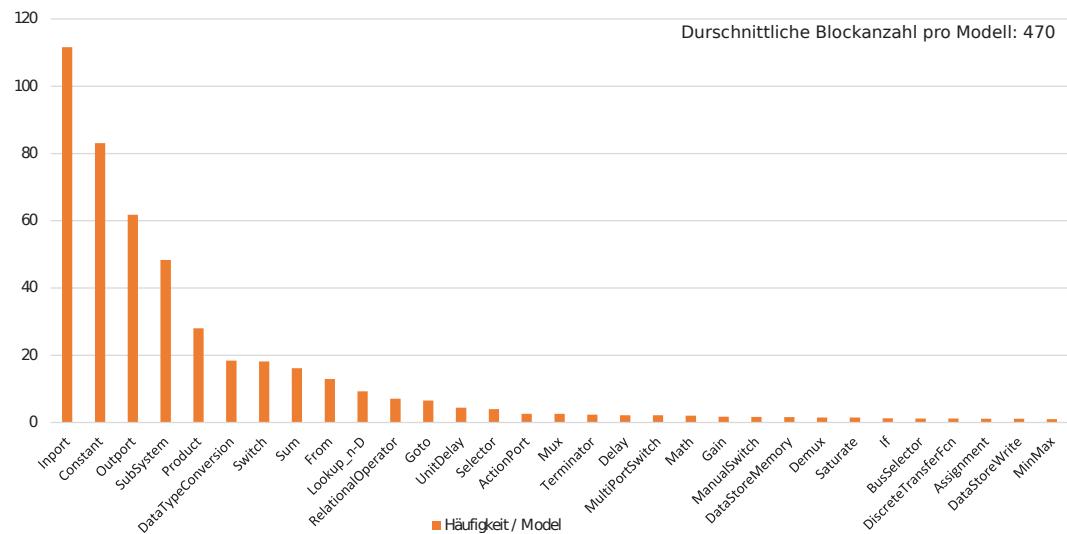


Abbildung 13: Durchschnittliche Blockhäufigkeiten pro Simulink Modell

lung, dass es nur wenige Blöcke gibt, die sehr frequentiert genutzt werden. Der Großteil findet nicht häufiger als zehn mal pro Modell Verwendung. Ebenfalls auffällig ist, dass “In- und Outport”, “SubSystem” sowie “From” und “Goto” Funktionsblöcke zusammen im Durchschnitt 51,3% aller verwendeten Blöcke ausmachen, jedoch nur zur Strukturierung des Modells beitragen und damit keinen Einfluss auf die Funktionalität und den generierten Code haben. Ferner ist zu untersuchen, welche Zielarchitekturen bzw. welche Steuergeräte adressiert werden. Neben einer Auswahl an Steuergeräten, die häufig im Bereich der Antriebsstrangsteuerung zum Ansatz kommen, ist in Tabelle 2 ebenfalls die *MicroAutoBox* von dSpace aufgeführt. Hierbei handelt es sich um ein Prototypensteuergerät, das bei IAV für erste Funktionstests eingesetzt wird.

¹⁴ Betrachtet wurden zwölf verschiedene Simulink Modelle

¹⁵ Gezählt wurden nur Blöcke, die im Durchschnitt wenigstens einmal verwendet wurden

¹⁶ “Display”, “Scope”, “To File”, “To Workspace”, “Reference” etc.

Tabelle 2: Prozessoren aus Steuergeräten für die Nutzung im Kraftfahrzeug

Steuergerät	Ausstattung
PKW-Motorsteuergerät	Prozessor: TC1793 (TriCore), 270MHz, 32-Bit Architektur, Multiple on-chip Speicher: 16-KB Instruction und 16-KB Data Cache, 128-KB Data Scratch-Pad RAM u.a., Superskalar mit 6-Stufiger Pipeline und vollständiger FPU Pipeline, Schnittstellen: Multi-CAN Modul, FlexRay, S/ASC ¹⁷ , MLI ¹⁸ + Performance Counter Registers.
Nutzfahrzeug-Motorsteuergerät	Prozessor: MPC5566 (e200z6), 132MHz, 32-Bit Architektur, 32-KB L1 Cache, 128-KB (on-chip) SRAM, Skalare Pipeline mit (LSU, BPU, SPE ¹⁹ , IU und zwei FPUs), Schnittstellen: CAN, RS232, Serial Peripheral Interface.
MicroAutoBox (dSpace)	Prototypensteuergerät bestehend aus Prozessorboard und I/O Board. Prozessorboard: IBM PowerPC 750GX, 800MHz, 100MHz Bustakt, 32-Bit Architektur, 32-KB L1 Cache, 1-MB L2 (on-chip) Cache, Superskalar mit FPU, BPU, SRU, LSU und zwei UIs (Abbildung 5) I/O Board: 156-Pin Anschluss mit Schnittstellen für CAN, ECU, RS232, LIN, FlexRay + Performance Monitor.

Da die Funktionsentwicklung bei IAV vorrangig in Matlab/Simulink gestaltet wird, ist die Einbindung einer Laufzeitanalyse, als Script, Toolbox, oder Funktionsblock in jene Entwicklungsumgebung anzustreben. Um überdies die Entwicklung des Modells nicht zu lange zu unterbrechen, ist eine möglichst benutzerfreundliche Abschätzung umzusetzen. Dies gewährleistet weiterhin einen entsprechenden “Workflow” bei der Funktionsentwicklung. Die Entwicklungstoolchain ist mit der Untersuchung des Vorgehensmodells, der Matlab-Entwicklungsumgebung, verschiedener Simulink Modelle sowie möglicher Zielarchitekturen hinreichend analysiert, um im Folgenden anhand der Aufgabenstellung eine konkrete Zielsetzung zu definieren und unter entsprechenden Randbedingungen einen Anforderungskatalog zu erstellen.

3.2 Zielsetzung

Aus der Aufgabenstellung und der etablierten Entwicklungstoolchain lassen sich im Folgenden konkrete Ziele ableiten. Mit diesen Zielvorgaben und entsprechenden Randbedingungen soll eine abschließende Evaluation der zu entwickelnden Software ermöglicht werden.

Das Verfahren ist in den automotiven Software Entwicklungsprozess von IAV zu integrieren. Dies bedeutet, dass die Laufzeitanalyse während der Funktionsentwicklung möglich sein soll. Damit relevante Designentscheidungen identifiziert werden können, ist

¹⁷Synchronous/Asynchronous Serial Channel

¹⁸Micro Link Interface

¹⁹Signal Processing Extension

die Laufzeitanalyse auf einzelne Module, sprich Simulink Funktionsblöcke, herunterzubrechen. Aufgrund von verschiedenen Architekturen, soll die zu entwickelnde Software darüber hinaus so zu konfigurieren sein, dass eine Laufzeitanalyse für verschiedener Steuergeräte bzw. Architekturen möglich ist. Der Mehrwert dieser Software soll darin liegen, ohne stete Codegenerierung sowie Kompilierung und früh während der Modellentwicklung verschiedene Subsysteme eines Simulinkmodells auf Modellebene hinsichtlich ihrer geschätzten Laufzeit untersuchen zu können. Zu entwickeln ist ein Konzept und eine entsprechende Software, die, eingebettet in die Matlab/Simulink Umgebung, dem Funktionsentwickler ermöglicht bereits in der sehr frühen Phase der Entwicklung und mit möglichst wenig Aufwand (hinsichtlich WCET-Analyse, Programmierkenntnisse oder Wissen über die Zielarchitektur) die Laufzeit der Funktionssoftware abzuschätzen, um so entsprechende Designentscheidungen nachzuvollziehen und verbessern zu können. Zu beachten ist bei der Konzeptionierung, dass ein “Trade-Off” zwischen Handhabung, Umsetzung sowie Genauigkeit der Abschätzung gefunden wird.

Das zu entwickelnde Konzept soll neben den bereits erwähnten Anforderungen ebenfalls die Machbarkeit für Laufzeitanalysen auf abstrakter Ebene untersuchen und über eine entsprechende Implementierung sowie Evaluierung die Aussagegüte einer Laufzeituntersuchung dokumentieren. Diese Zielsetzung soll im Folgenden durch Randbedingungen eingegrenzt und weiter definiert werden.

3.3 Randbedingungen

Wie bereits im Kapitel 3.1 erläutert, kann eine Laufzeitanalyse aufgrund der beschränkten Verwendung verschiedener Funktionsblöcke auf eine Teilmenge der Simulink Funktionsbibliothek eingeschränkt werden. Eine Analyse soll zudem auf MAAB-konforme, simulationsfähige Modelle zugeschnitten sein. Simulationsfähig bedeutet in diesem Kontext, dass alle Daten und Einstellungen so zur Verfügung zu stellen sind, dass eine fehlerfreie Simulation des Modells in Simulink möglich ist. Die zu konzeptionierende Prozesskette soll eine erste Laufzeitabschätzung auf abstrakter (Modell-)Ebene ermöglichen, weniger eine aufwändige exakte Berechnung einer WCET. Dabei können jene Prozesse erweiternd zu den bereits etablierten Werkzeugen, wie aiT, eingesetzt werden, wobei während der Entwicklung eines Modells stets Designentscheidungen mit Laufzeitinformationen ergänzt und nach ausgereifter Entwicklung die Software im Detail mit der statischen Laufzeitanalyse untersucht werden kann. Ferner soll ausschließlich eine Laufzeitschätzung ohne Unterbrechung (Interrupts) oder Scheduling-Einflüsse seitens eines Taskmanagers oder eines Betriebssystems Betrachtung finden.

Werden unbekannte Funktionsblöcke im Modell verwendet, so sind diese zu ignorieren (mit entsprechender Warnung für den Nutzer) und es ist für den Rest des Modells eine entsprechende Abschätzung auszugeben. Benutzerdefinierte Funktionsblöcke, wie sogenannte “S-Functions” und “Matlab-Fuctions”, in denen vom Benutzer geschrie-

bener Quellcode ausgeführt wird, sollen ebenfalls keine Betrachtung finden, da sie einer aufwändigen Quellcodeanalyse bedürfen und somit der Idee einer kosteneffizienten und schnellen Laufzeitabschätzung widersprechen. Darüber hinaus kann die Funktionalität des Quellcodes äquivalent in Funktionsblöcken abgebildet werden. Die Laufzeitanalysen, wie sie bisher in dieser Arbeit untersucht werden, setzen stets das Vorhandensein nur eines Prozessorkerns voraus. Für das im folgenden Kapitel entwickelte Konzept soll ebenfalls ausschließlich die Befehlsabarbeitung mit einem Prozessorkern betrachtet werden.

3.4 Zusammenfassung

Über eine Analyse der bestehenden Entwicklung bei IAV, bestehende Simulink-Modelle sowie potentielle Steuergerätehardware und anhand der Aufgabenstellung konnte eine konkrete Zielsetzung sowie ebenfalls einige Randbedingungen definiert werden.

Aufgrund des bei IAV etablierten Entwicklungsablaufes (vgl. Abbildung 3 und 12) ist eine Laufzeitanalyse zu entwickeln, die bereits während der Modellentwicklung einsetzbar ist. Dies bedeutet, die Analyse sollte ohne die ständige Codegenerierung sowie -komplilierung möglich sein. Damit die zu entwickelnde Software einen Mehrwert für den Funktionsentwickler darstellt, sollte sie in die Modellentwicklungsumgebung eingebettet werden, eine schnelle Laufzeitabschätzung ermöglichen sowie ohne Kenntnisse über eine Laufzeitanalyse einsetzbar sein. Es soll ferner möglich sein Laufzeiten je Funktionsblock aufzuschlüsseln, um zeitaufwändige Designentscheidungen identifizieren zu können. Dabei konzentriert sich die Auswertung der Laufzeit auf Einkernprozessoren und die tatsächliche Laufzeit des Programms ohne Scheduling-Einflüsse. Einen besonderen Mehrwert stellt die zu konzeptionierende Prozesskette dar, wenn sie als schneller, auf Modellebene arbeitender Prozess eine Ergänzung zur aufwendigen statischen Laufzeitanalyse darstellt. Dabei kann der iterative Prozess, wie er in Abbildung 12 zwischen Modellentwicklung und Laufzeitanalyse abgebildet ist, effektiv neben der Modellentwicklung eingesetzt werden. Die statische Laufzeitanalyse kann dann, nach längeren Entwicklungszeiten, als absichernder Prozess eine detaillierte Untersuchung des Binärprogrammes vornehmen.

Der Anforderungskatalog soll folgend sowohl die Anforderungen an die zu entwickelnde Prozesskette sowie die Randbedingungen zusammenfassen und tabellarisch festhalten.

Tabelle 3: Anforderungskatalog

Anforderung	Beschreibung
Prozesskette	Die zu entwickelnde Prozesskette soll es ermöglichen anhand eines Simulink Modells eine Laufzeitabschätzung für eine entsprechende Architektur zu berechnen, wobei nicht stets eine Codegenerierung oder Kompilierung des Quellcodes vonnöten sein soll.
Einbettung in Matlab	Die zu entwickelnde Software ist in die Matlab-Simulink-Entwicklungsumgebung einzubetten. Dies bedeutet, dass die Laufzeitanalyse aus Matlab bzw. Simulink heraus aufrufbar sein soll.
Abschätzung auf Modellebene	Da die Laufzeitanalyse sehr früh in der Modellentwicklung Anwendung finden soll, ist die Abschätzung auf Modell-Ebene durchzuführen.
Ergänzung zur statischen Analyse	Aufgrund unterschiedlicher Zielsetzungen und Ausführungsebenen kann die zu entwickelnde Prozesskette auf Modellebene eine Ergänzung zur aufwendigen statischen Analyse auf Binärebene bieten.
Effiziente Umsetzung	Damit eine Analyse mit entsprechenden Prozessen zum frühen Zeitpunkt der Entwicklung zu einem Mehrwert beitragen kann, ist eine kosten- bzw. zeit-effiziente Umsetzung anzustreben
Nach MAAB Standard	Modelle die hinsichtlich ihrer Ausführungsduer untersucht werden, sollen dem MAAB-Standard genügen sowie alle Daten und Einstellungen bereitstellen, um eine Simulation in Simulink zu ermöglichen
Konfigurierbar	Da die Ausführungsduer eines Programmes eng mit der ausführenden Hardware verknüpft ist, soll die in Matlab-Simulink eingebettete Software hinsichtlich verschiedener Hardwarearchitekturen konfigurierbar sein.
Einfacher Zugang	Die Laufzeitanalyse soll ohne Kenntnisse über die Laufzeitanalyse oder die adressierte Architektur auskommen und so jedem Funktionsentwickler auch ohne Programmierkenntnisse den einfachen Einsatz der Laufzeitanalyse ermöglichen.
Einkernprozessoren	Die Laufzeituntersuchungen sowie das im folgenden Kapitel entwickelte Konzept soll sich auf Architekturen mit Einkernprozessoren beschränken.
Tatsächliche Programmlaufzeit	Die Betrachtung wird begrenzt auf die tatsächliche Ausführungszeit des Programmes. Ausgenommen sind somit mögliche Beeinflussungen durch Scheduling- oder Verdrängungsstrategien eines Betriebssystems oder Taskmanagers
Laufzeit auf Module herunterbrechen	Damit Designentscheidungen im Detail hinsichtlich ihrer Laufzeit untersucht werden können, sind Informationen zur Ausführungszeit für einzelne Module bzw. Funktionsblöcke anzugeben.

4 Konzept

In diesem Kapitel soll zunächst auf Grundlage der Anforderungsanalyse eine Diskussion verschiedener Techniken zur Laufzeituntersuchung erfolgen. An diese Diskussion schließt sich die Entwicklung eines, zu den etablierten Vorgehensweisen, alternativen Konzepts zur Laufzeitbestimmung für Matlab/Simulink Modelle an. Das Konzept umfasst den Aufbau einer Tabelle (im Folgenden “Lookup Table” genannt), in der Laufzeitinformationen für Funktionsblöcke hinterlegt sind, sowie eine Modellanalyse von Simulink Modellen.

4.1 Entwicklung der Prozesskette

Es ist im Folgenden eine Prozesskette zu entwerfen, die, eingebettet in die Matlab/Simulink Entwicklungsumgebung, eine Laufzeitanalyse eines Simulink-Modells ermöglicht. Eine Analyse soll dabei den Entwicklungsprozess einer Funktionalität nicht zu lange unterbrechen und ohne durchgehendes Vorhandensein eines Steuergerätes ausführbar sein. Zunächst ist zu diskutieren, inwieweit bereits bestehende Entwicklungen den Anforderungen aus Kapitel 3 gerecht werden.

Wie im Stand der Technik vorgestellt, besteht die Möglichkeit einer engen Verknüpfung zwischen einem Programm zur modellgetriebenen Softwareentwicklung (SCADE, ASCET) und einem Programm zur Laufzeitanalyse (aiT). Hier ist es möglich nach einer Codegenerierung und anschließender Kompilierung des Codes eine sehr gute Laufzeitabschätzung vorzunehmen. Der besondere Vorteil besteht dabei darin, dass der Funktionsentwickler selber keine Angabe mehr zum Programmcode machen muss und die gesamte Analyse inklusive Codegenerierung und Kompilierung im Hintergrund ausgeführt wird. Die Analyse basiert bei diesem Verfahren jedoch weiterhin auf mathematischen Theorien wie dem ILP, die enge Laufzeitschranken liefern, jedoch für größere Programme deutliche längere Berechnungszeiten bedürfen. Darüber hinaus ist für jeden Analysevorgang ein Compiler sowie eine Codegenerierung notwendig. Aufgrund der Analyse auf Binärebene ist eine derartige Untersuchung außerdem stets sehr exakt auf verschiedene Architekturen anzupassen. Ebenso ist eine Anpassung an verschiedene Compiler vonnöten.

Die messbasierte Laufzeitmessung ist, bis auf einen geeigneten Compiler, nicht weiter für verschiedene Architekturen anzupassen. Nach den in Kapitel 2.1.4 beschriebenen Vorgehensweisen zur messbasierten Laufzeitanalyse lässt sich jedoch nicht für beliebige Architekturen eine Laufzeit in einer Genauigkeit bestimmen, die für die in dieser Arbeit entwickelten Prozesse nötig ist. Das Schalten digitaler Ausgänge ist im Wesentlichen vom Prozessor-, oder Bustakt abhängig. Diese weisen zumeist nicht die benötigte Taktrate auf, um auch sehr kurze (einzelne Assembleroperationen) zeitlich abzubilden. Ferner ist für eine solche Methode, neben einer entsprechenden Toolkette, ebenso das Steuergerät bzw. die Hardware selbst erforderlich. Hinzu kommt eine aufwändige Handhabung, da für jeden

zu untersuchenden Entwicklungsschritt eine Messung auf der Hardware erforderlich ist. Folglich ist dieser Ansatz unter den gegebenen Anforderungen zur Laufzeitbestimmung ungeeignet.

Eine Möglichkeit der Laufzeitanalyse, die ohne die Zielhardware auskommt, ist die Prozessoremulation²⁰. Dabei wird durch Software die gewünschte Architektur emuliert, welche dann die verschiedenen Programme ausführt. Eine entsprechende Software, wie beispielsweise *SimpleScalar* oder der “*ARMulator*”²¹, kann über eine solche Emulation ebenfalls Prozessortakte zählen. In der Arbeit von *Sandell* [San04] konnte gezeigt werden, dass der “*ARMulator*” im Vergleich mit aiT enge Laufzeitschranken angeben kann und dementsprechend für eine Laufzeitabschätzung einsetzbar ist (vgl. Kapitel 2.2.1 (Chronos + SimpleScalar)). Als erheblicher Nachteil einer möglichst exakten Nachbildung von hardwarenahen Prozessen in Software ist jedoch die Komplexität einer solchen Umsetzung. Zudem ist für jede Analyse eine Codegenerierung und Kompilierung des Simulink Modells notwendig.

Die bereits diskutierten Ansätze zur Laufzeituntersuchung werden in die statische bzw. die messbasierte (dynamische) Laufzeitanalyse eingeteilt. Es zeigt sich, dass eine möglichst exakte Bestimmung von Laufzeitschranken stets auch eine Umsetzung des Modells in Quell- bzw. Binärkode bedingt. Dies gilt sowohl für den statischen wie auch für den messbasierten Ansatz. Damit sehr hardwarenahe Vorgänge bei der Ausführung der Software nachvollzogen werden können, ist es vorteilhaft wenig abstrakte Operationen zu analysieren. Dazu ist das Binärformat, in dem einzelne Maschinenbefehle angegeben sind, besonders gut geeignet. Auf der abstrakten Modellebene können jedoch die tatsächlich auf der Hardware ausgeführten Bearbeitungsschritte kaum nachvollzogen werden.

Tabelle 4: Gegenüberstellung zweier Analyseebenen

Ebene	Darstellung	Laufzeitanalyse
Modellebene	Nachvollziehbare, graphische Repräsentation	Keine konkreten Aussagen möglich
Binärebene	Nicht nachvollziehbare Bitfolgen	Hohes Analysepotential

Wie in [LM10] vorgestellt wird, ist ebenfalls ein hybrider Ansatz möglich. Dabei werden die Konzepte der messbasierten und statischen Analyse kombiniert. Der hybride Ansatz identifiziert, beispielsweise mit Hilfe von abstrakten Syntaxbäumen, sogenannte *single feasible paths* (SFP), deren Ausführung von den Eingabedaten unabhängig ist. Diese SFPs werden im nächsten Schritt auf der Zielhardware oder mit entsprechenden Emulatoren vermessen. Von Eingabedaten abhängige Pfade, sind weiterhin mit allen möglichen Eingabewerten zu vermessen. Abschließend sind jene gewonnenen Zeiten mit den Methoden der statischen Laufzeitanalyse zu verbinden, um für das gesamte Programm

²⁰Auch: “Computer Architecture Simulator” oder “Cycle-Accurate Simulator”

²¹www.arm.com

den längsten Ausführungspfad und damit eine obere Laufzeitschranke zu bestimmen. Der Vorteil dieser Methode ist, dass eine Analyse nicht mehr abhängig von einem komplexen Hardwaremodell ist. Die Unsicherheit, mit den Messungen tatsächlich die WCET bestimmt zu haben, bleibt bestehen, ist jedoch minimiert [LM10].

Die Idee eines hybriden Ansatzes soll im Folgenden von der Codeebene, also dem Identifizieren von SFPs, so wie es in [LM10] beschrieben wird, auf die Modellebene übertragen werden. Aufgrund der Forderung, eine Laufzeitanalyse ist ohne ständiges Vorhandensein der Toolkette bzw. der Hardware auszuführen und aufgrund der Modularität eines Simulink-Modells, kann ein Ansatz das Speichern von Laufzeitinformationen zu jedem Funktionsblock in einer Art Lookup Table sein. Auf diese Weise gewonnene Information können nun in jeder Analyse eingesetzt werden, ohne, dass stets eine Codegenerierung und Kompilierung vonnöten ist. Die Identifikation von SFPs wird damit durch eine Analyse der einzelnen Funktionsblöcke ersetzt. Dazu sind die einzelnen Blöcke zunächst so umzusetzen, dass eine Laufzeituntersuchung möglich ist (vgl. Tabelle 4). Diese Untersuchung ist auszuwerten und die Ergebnisse entsprechend abzuspeichern (Lookup Table). Da die Zeit für eine Ausführung einer Funktionalität eines Funktionsblocks direkt von dessen konkreter programmiertechnischer Umsetzung abhängig ist, ist eine Laufzeitanalyse auf Modellebene ohne jegliches Wissen über eine Umsetzung und Übersetzung in einen (möglichst hardwarenahen) Code nicht zielführend. Da Simulink-Modelle aus einer

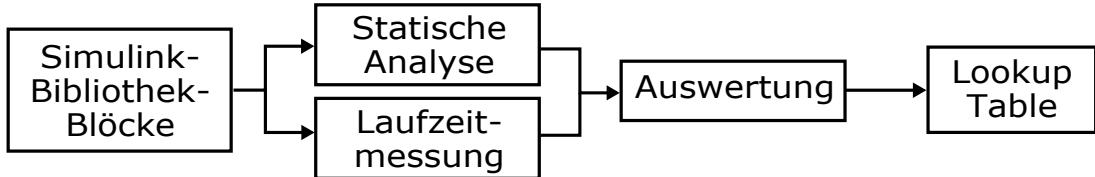


Abbildung 14: Prozesskette zum Aufbau einer Lookup Table für Simulink-Funktionsblöcke

endlichen Menge an (Grund-)Funktionsblöcken bestehen und sich ihre Funktion, im Gegensatz zu SFPs in verschiedenen Programmen nicht ändert, kann auf die Informationen einer einmalig durchgeführten Analyse bei jeder Laufzeituntersuchung zurückgegriffen werden. In Abbildung 14 ist der schematische Ablauf einer solchen Prozesskette dargestellt. Dabei können zwei unterschiedliche Herangehensweisen unterschieden werden.

Eine Möglichkeit zum Aufbau einer Lookup Table ist weniger ein hybrider Ansatz, als die Anwendung des statischen Analyseverfahrens auf die modulare Modellebene. Dabei ist jeder Funktionsblock in Quellcode, vorzugsweise Assembler- oder Binärcode, umzusetzen und einzeln zu analysieren. Mit den Methoden der statischen Analyse kann nun die Laufzeit der Funktionalität auf Codeebene möglichst exakt bestimmt werden. Eine solche Analyse ist darüber hinaus deutlich weniger umfangreich als die Analyse der gesamten Software. Es ist jedoch zu beachten, dass die Laufzeit ganz elementar von den Eingabewerten abhängt. Diese Abhängigkeit ist bei der anschließenden Auswertung der

Analyse zu beachten und mit in die Lookup Table aufzunehmen. Die Prozesskette aus Abbildung 14 ist beispielhaft anhand des “Produkt-Blocks” aus der Simulink Bibliothek in Abbildung 15 dargestellt. Durch diese Prozesskette kann für alle benötigten Funktionsblöcke eine umfangreiche Infomationstabelle aufgebaut werden. Diese Tabelle weist so

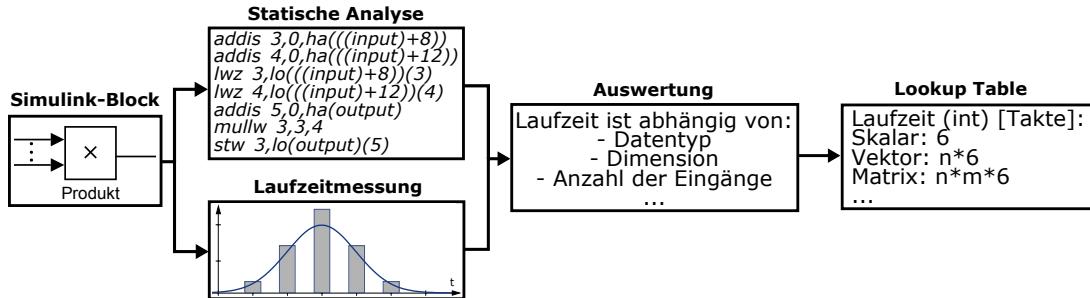


Abbildung 15: Exemplarische Prozesskette für den “Produkt-Block”

verschiedenen Konfigurationen eines Blocks (Anzahl, Dimension und Datentyp der Eingangssignale etc.) eine Laufzeitabschätzung zu. Der Aufbau einer solchen Datenbank ist aufgrund der Methodik der statischen Analyse sowie den verschiedenen möglichen Konfigurationen zunächst relativ umfangreich. Entgegen einem Ansatz ohne Lookup Table ist dieser Aufwand jedoch lediglich einmalig zu betreiben.

Die weniger aufwändige Möglichkeit den Aufbau einer Lookup Table zu gestalten, ist ein hybrider Ansatz, der ebenfalls die Modularität von Simulink Modellen ausnutzt. Dabei wird die statische Analyse durch eine Laufzeitmessung auf dem Steuergerät ersetzt. Im Kontrast zum zuvor erläuterten Ansatz wird nun der Code jedes Blocks direkt auf der Zielarchitektur ausgeführt. Bei dieser Ausführung ist die Laufzeit des generierten Codes zu messen. Wie eine solche Messung vorgenommen werden kann, wird im folgenden Kapitel 4.2 diskutiert. Der Prozess der Auswertung und der Aufbau der Lookup Table sind analog zum Ansatz der statischen Analyse. Zu beachten ist, dass der Ansatz der statischen Analyse keine tatsächliche Hardware des Steuergeräts benötigt, jedoch ist die Analyse sehr exakt und dementsprechend aufwändig. Im Gegensatz dazu ist die messbasierte Analyse weniger komplex, benötigt jedoch zum Aufbau der Lookup Table die Zielhardware.

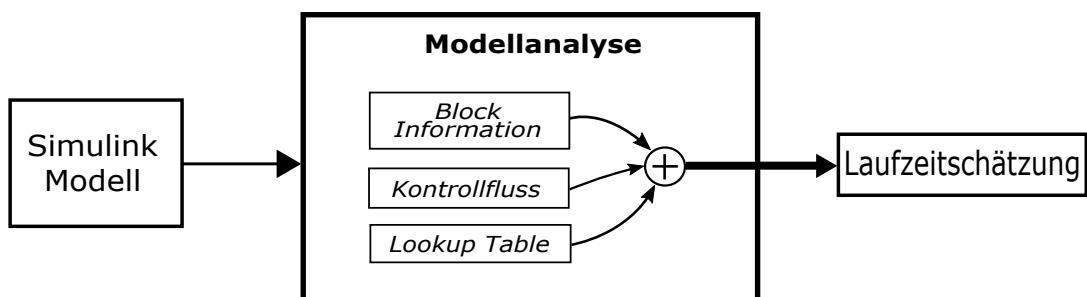


Abbildung 16: Die Modellanalyse verbindet Informationen aus Lookup Table und Modell

Die vorliegende Arbeit soll folgend den Ansatz der hybriden Laufzeitmessung auf Modellebene genauer untersuchen und herausarbeiten wie fundiert eine Laufzeitschätzung auf Modellebene und auf Basis einer messbasierten Lookup Table sein kann. Darüber hinaus ist die bereits vorgestellte Prozesskette um Prozesse zu erweitern, so dass der gesamte Ablauf einer Laufzeitanalyse auf Modellebene abgebildet werden kann.

Nachdem eine Lookup Table durch die zuvor entwickelten Prozesse aufgebaut ist, sind die jeweils zu analysierenden Simulink Modelle zu betrachten. Dabei sind insbesondere Informationen zu den verwendeten Blöcken, den verschiedenen Datentypen, zu Schleifen-durchläufen und zum Kontrollfluss auszuwerten. Bevor eine Laufzeit abgeschätzt werden kann, ist also eine Modellanalyse vorzunehmen (siehe Abbildung 16). Wie eine Modellanalyse eingebettet in die Matlab/Simulink Umgebung zu konzeptionieren ist, wird in Kapitel 4.3 diskutiert.

Das Konzept, bestehend aus der Kombination von Lookup Table und Modellanalyse, soll in den folgenden Kapiteln als Grundlage zur weiteren Diskussion verschiedener Ansätze zur Vermessung von Funktionsblöcken oder zur Untersuchung von Simulink Modellen dienen.

4.2 Laufzeitbestimmung für Simulink Blöcke

Als erster Prozessschritt ist in Abbildung 14 die Codegenerierung implizit vorausgesetzt, um eine Laufzeitmessung bzw. statische Analyse auszuführen. Insbesondere bei der Umsetzung von einer abstrakten Darstellung in eine vom Computer ausführbare Form wird ein Großteil der Aspekte der späteren Ausführungsduer festgelegt. Folgend soll neben einer Betrachtung der Codegenerierung unter Matlab-Simulink ebenfalls mögliche Methoden zur Laufzeitmessung und Auswertung sowie Möglichkeiten zum Aufbau einer Lookup Table diskutiert und somit eine konkrete Umsetzung der in Abbildung 14 und 15 dargestellten theoretischen Prozesse erarbeitet werden.

Die Codeerzeugung unter Simulink geschieht mittels eines sogenannten “Simulink Coders”, dieser übersetzt das Blockdiagramm in C-Code. Dieser C-Code kann anschließend, je nach Architektur, mit einem Compiler zu Binärkode kompiliert werden. Es wird also deutlich, dass eine Laufzeitanalyse auf Modellebene stets eine starke Abhängigkeit zur Codegenerierung aufweist. Eine weitere Abhängigkeit der Laufzeit ist durch die Kompliierung gegeben, da auch hier Optimierungen und Umstrukturierungen einen Einfluss auf die Programmlaufzeit haben. Dabei kann erneut auf die Problematik aus Tabelle 4 verwiesen werden: Erst auf Binärebene ist eine verlässliche Aussage über die Ausführungsduer möglich. Damit der Aufbau einer Lookup Table mit entsprechenden Laufzeitinformationen möglich ist, sind also die Funktionsblöcke jeweils einzeln zu vermessen. Für einen hybriden (messbasierten) Ansatz bedeutet dies, dass die in Abbildung 13 identifizierten Funktionsblöcke, jeweils in C- bzw. Binärkode umzusetzen sind. Die Ausführungsduer

dieser Blöcke ist anschließend auf der Zielhardware zu messen. Es ist bei diesem Vorgehen darauf zu achten, dass die Ausführungsduer mitunter von den Eingabeparametern abhängt. Sowohl für die Auswertung der ermittelten Laufzeitdaten als auch für die Messung der Laufzeit lassen sich verschiedene Methoden unterscheiden.

Eine Möglichkeit der Laufzeitmessung stellt das tatsächliche Messen von logischen Pegeln an Pins der ausführenden CPU dar. Dabei ist das zu vermessene Programm, wie folgend in Pseudocode dargestellt, anzupassen. Bei Beginn der zu messenden Ausführung wird ein beliebig gewählter digitaler Ausgang (Pin) auf logisch eins bzw. "high" gesetzt. Danach wird das Programm entsprechend ausgeführt. Nach Beendigung der Ausführung ist der selbe Ausgang wieder auf logisch null bzw. "low" zu setzen.

Quellcode 1: Pseudocode für die Laufzeitmessung mit CPU-Pins

```
set_pin_high();
/* simulink block functionality */
set_pin_low();
```

Der Signalverlauf eines solchen digitalen Ausgangs kann nun mit Hilfe eines Oszilloskops angezeigt und die Laufzeit durch die Differenz von steigender und fallender Flanke berechnet werden. Dabei ist es von der Auflösung des Oszilloskops abhängig welche Laufzeiten tatsächlich gemessen werden können. Da viele Simulink Funktionsblöcke sehr elementare Rechenoperationen (Addieren, Multiplizieren etc.) ausführen, ist eine entsprechend hohe Auflösung vonnöten, um die Laufzeiten exakt abzubilden. Da die Ausführungsduer der Funktionsblöcke aufgrund von verschiedenen Eingabeparametern schwankt, ist die Messung mittels eines Oszilloskops aufwändig. Überdies ist die Messung sowie die Auswertung der gewonnenen Daten bei dieser Vorgehensweise schlecht zu automatisieren.

Eine weitere Strategie zur Messung von Laufzeiten, bei der eine höhere Automatisierung möglich ist, ist die Verwendung von "Timern", die die CPU-Zeit messen. Dabei ist die CPU-Zeit, die Dauer, die ein Prozessor benötigt, um ein Programm abzuarbeiten. Laufen mehrere Programme zur selben Zeit auf einem Prozessor, so gilt in den meisten Fällen: *Programmlaufzeit > CPUZeit*, aufgrund von Scheduling- bzw. Verdrängungsstrategien, oder dem Warten auf Nutzereingaben. Da als Randbedingung die Laufzeitanalyse ohne Betriebssystem- oder Taskmanager-Einflüsse definiert ist, ist ebenfalls die Messung unter dieser Bedingung durchzuführen. Das Programm läuft somit ohne Unterbrechung. Auch bei der Verwendung von "Timern" ist auf eine möglichst hohe Auflösung zu achten, um eine verwertbare Lookup Table aufbauen zu können. Software zum Messen von relativen Zeiten basiert auf diesen sogenannten "Timern"; allgemein werden sie als "Hardware Performance Counters" (HPC) bezeichnet. Dies sind Zähler, die verschiedene Ereignisse auf der CPU zählen. Ereignisse können neben Prozessortakten ebenfalls Cache Misses und Hits oder durchgeführte Instruktionen der UI oder FPU

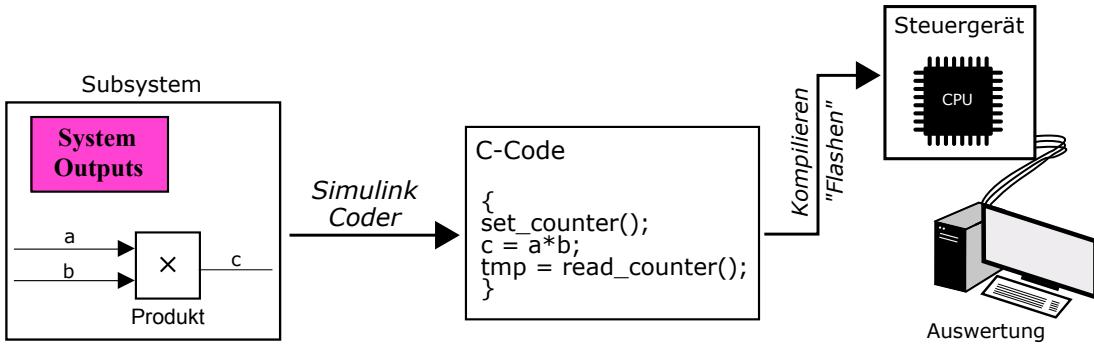


Abbildung 17: Ablauf der Laufzeitmessung für Simulink Blöcke

sein. Diese HPCs sind in fast allen gängigen Prozessorarchitekturen verbaut [MZK11]. Damit bieten sie aufgrund der Möglichkeit Prozessortakte zählen zu können, eine perfekte Grundlage, um Laufzeiten zu analysieren. Die im KFZ-Bereich sehr weit verbreiteten *TriCore* Prozessoren der Firma Infineon sowie das Prototypensteuergerät *MicroAutoBox* von dSpace (siehe Tabelle 2) bieten ebenfalls entsprechende HPCs an.

Da HPCs weit verbreitet sind und sie verschiedenste Prozessorgänge exakt messen können sowie leicht über wenige Assemblerbefehle steuerbar sind, bieten sie eine optimale Möglichkeit Programmlaufzeiten zu analysieren. Im Folgenden wird erläutert, wie diese Messmethodik mit der Codegenerierung in Simulink verbunden werden kann, um einzelne Funktionsblöcke hinsichtlich ihres Laufzeitverhaltens zu untersuchen und anhand gewonnener Informationen eine Lookup Table aufzubauen. Damit entsprechende Laufzeitinformationen für einen Teil eines Programmes ermittelt werden können, ist dieser Abschnitt, ähnlich dem Quellcode Beispiel 1, zu kapseln: Zu Beginn der Ausführung ist der entsprechende Zähler zurückzusetzen bzw. zu starten und im Anschluss auszulesen. Damit diese Methodik in Simulink umgesetzt werden kann, sind entsprechende Quellcodezeilen in den generierten C-Code einzufügen. Mit dem *System Output* Block bietet Simulink die Möglichkeit vor und nach der Ausführung von Subsystemen eigenen Quellcode einzufügen (siehe Anhang A Abb. 35). Nach jeder Ausführung sind die entsprechenden Zählregister auszulesen und zu speichern. Um möglichst repräsentative Werte zu erhalten, sind mehrmalige Ausführungen mit unterschiedlichen Eingabewerten durchzuführen. Abbildung 17 stellt den Vorgang der Laufzeitmessung exemplarisch anhand des Produktionsblocks dar. Die Zeilen `set_counter();` und `tmp = read_counter();` werden durch den *System Outputs* Block während der Codegenerierung eingefügt. Während das Programm auf dem Steuergerät ausgeführt wird, können die gemessenen Werte beispielsweise mit einer Kalibrations- und Applikationssoftware auf einem verbundenen Computer gespeichert und anschließend ausgewertet werden. Mit Hilfe mehrerer Subsysteme und Kontrollflussblöcke (Switch, If/Else...) im Simulink Modell ist es möglich, mit nur einer Codegenerierung mehrere Funktionsblöcke auf einmal vermessen zu können (siehe Anhang Abbildung 37).

Tabelle 5: Wertbasierter Aufbau einer Lookup Table

Blockname	Prozessortakte	Standardabweichung	WCET
Product	5,00	1,00	12,00
Sum	4,00	1,00	10,00

Tabelle 6: Funktionsbasierter Aufbau einer Lookup Table

Blockname	Prozessortakte	Standardabweichung
1-D Lookup Table	$\log_e(\#breakpoints) \cdot 18 + 64$	10,6
Assignment	$34 + 2 \cdot \#max_loops$	5,5

Tabelle 7: Skriptbasierter Aufbau einer Lookup Table

Blockname	Prozessortakte	Standardabweichung
1-D Lookup Table	rte.lookupTable(handle)	10,6
Assignment	rte.assignment(handle)	5,5

Abhängig davon, welche Aussage anhand einer späteren Laufzeitabschätzung getroffen werden soll, ist die Messung entsprechend anzupassen. Ist beispielsweise eine pessimistische Abschätzung gewünscht, so kann jeweils die gemessene WCET für jeden Funktionsblock herangezogen werden. Damit ist es äußerst wahrscheinlich, dass es zu einer deutlichen Überabschätzung der tatsächlichen Laufzeit kommt, jedoch ist so die Wahrscheinlichkeit einer Unterschätzung der Laufzeit minimiert. Eine Abschätzung der durchschnittlichen Laufzeit kann überdies ebenfalls sinnvoll sein. Dabei sind für die verschiedenen Blöcke Messreihen mit unterschiedlichen Eingabeparametern aufzunehmen. Der berechnete Mittelwert dieser Messungen sowie die entsprechende Standardabweichung sind anschließend in die Lookup Table mit aufzunehmen (siehe Tabelle 5). Eine spätere Laufzeitschätzung auf Modellebene besteht dann neben der geschätzten Laufzeit ebenfalls aus einem akkumulierten Fehler bzw. einer Standardabweichung, die sich aus der Summe der Abweichungen der verschiedenen Funktionsblöcke ergibt.

Aufgrund mannigfaltiger Einstellungen, die für viele Simulink Blöcke angegeben und verändert werden können, ist eine Lookup Table, wie sie in Tabelle 5 aufgeführt ist, nicht umfassend genug, da die Laufzeiten mitunter eine hohe Abhängigkeit zu Eingabegrößen und Einstellungen aufweisen. Als Beispiel für eine mehrdimensionale Abhängigkeit dient der sogenannte *Lookup Table* Funktionsblock aus der Standard Simulink Blockbibliothek. Der “Lookup Table”-Block führt anhand von N Stützpunkten (Lookup Table) für einen gegebenen Eingabewert x eine Interpolation $f(x)$ zwischen den zwei zum Eingabewert nächsten Werten (aus der LookupTable) durch. Dabei ist diese Berechnung von der Dimension der Lookup Table, der Dimension von x , den verwendeten Algorithmen sowie insbesondere der Anzahl an Stützstellen abhängig. Eine einfache Wertangabe, wie in Tabelle 5, ist dementsprechend nicht möglich. Für einfache Funktionalitäten ohne vielfältige Einstellungsmöglichkeiten sind Zahlenwerte jedoch weiterhin hinreichend. Abbildung 36

(Anhang A) stellt die Laufzeit des *1-D Lookup Tables* in Abhängigkeit der Stützpunkte (Breakpoints) dar. Es zeigt sich, dass die Laufzeit logarithmisch von der Anzahl der genutzten Stützpunkten abhängt. Hinge die Berechnungsduer nur von der Anzahl der Stützpunkte ab, so ergäbe eine Regression anhand der Messpunkte für diesen Funktionsblock mit $t_{est} = \log_e(\#breakpoints) \cdot 18 + 64$ einen sehr guten Laufzeitschätzer und eine Lookup Table könnte, wie in Tabelle 6, die Funktionen zur Laufzeitschätzung aufnehmen. Da sich jedoch die einzelnen Einstellungen gegenseitig beeinflussen, kann die Entwicklung eines entsprechenden Laufzeitschätzers äußerst aufwändig werden. Um einer effizienten Umsetzung zu entsprechen, können approximierende Annahmen getroffen, oder gewisse Einstellungen in der Laufzeitanalyse ausgelassen werden. Dies ist insbesondere dann möglich, wenn Einstellungen nur geringen Einfluss auf die Ausführungsduer haben.

Infolge unterschiedlicher Einstellungen der verschiedenen Funktionsblöcke unter Simulink sind die Größen von denen die Laufzeit eines Blocks abhängen kann ebenfalls unterschiedlich. Die Berechnungsvorschrift für den Laufzeitschätzer ändert sich dementsprechend (siehe Tabelle 6). So wie sich die Ausführungsduer eines *Lookup Table* Blocks mit unterschiedlichen Eingabewerten und Stützstellen ändert, so ändert sich beispielsweise die Ausführungsduer eines *Assignment*²² Blocks abhängig davon, ob er sich in einer Schleife (For- oder While-Subsystem) befindet, oder nicht. Es zeigt sich somit, dass die Berechnung eines Laufzeitschätzers sehr individuell für jeden Block angepasst werden muss. Diese Anpassung hat entsprechende Auswirkungen auf den Aufbau einer Lookup Table. Ferner ist auch eine Angabe für die WCET nicht mehr möglich, da es keine festen Laufzeiten mehr für die einzelnen Blöcke gibt und so eine obere Schranke nicht angegeben werden kann. Um jedoch trotzdem eine gewisse Absicherung zu erreichen ist die Verwendung einer sogenannten „*safety margin*“, wie sie in [LM10] vorgestellt wird, denkbar. Dabei wird zu jedem berechneten Wert ein konstanter „Sicherheitswert“ addiert.

Ein Aufbau, wie er in Tabelle 5 dargestellt ist, ist für die meisten Funktionsblöcke unzureichend und eine Lookup Table mit Funktionen als Laufzeitschätzer (Tabelle 6) ist aufgrund verschiedener Abhängigkeiten für eine spätere Interpretation durch ein Computerprogramm aufwändig. In Tabelle 7 ist ein weiterer Ansatz für eine Lookup Table aufgeführt, der einen Kompromiss zwischen den zwei zuvor genannten Ansätzen bietet. Beim skriptbasierten Ansatz wird die Forderung nach einer Integration in die Matlab/Simulink Umgebung ausgenutzt. Dabei weißt eine Lookup Table jedem Block eine Matlab-Funktion zu²³. Diese Funktion kann nun individuell die Schätzung für die Prozessortakte berechnen.

²²Weist einem Vektor einen gegebenen Wert an einer gegebenen Stelle x zu

²³In gleicher Weise kann auch für die Standardabweichung eine Funktion definiert werden

Mit Hilfe der in Matlab eingebetteten Funktionalität “`eval()`” kann ein solcher Ausdruck in Matlab evaluiert bzw. interpretiert und damit ausgeführt werden²⁴. Um eine Eindeutige Schnittstelle zu schaffen, sind die Block-Funktionen wie folgt zu definieren.

```
function [ cpu_cycles ] = function_name( block_handle )
```

Wobei `function_name` durch den entsprechenden Funktionsnamen zu ersetzen ist. Simulink bietet die Möglichkeit über sogenannte *Handles* auf die einzelnen Funktionsblöcke eines Modells zuzugreifen, um so beliebige Informationen des Blocks abzufragen (zu nutzen sind hier die `get()` bzw. `get_param()` Funktionen). Eine Funktion muss somit nur dieses Handle übergeben bekommen und kann so individuell Abhängigkeiten für die Laufzeitschätzung auflösen. Darüber hinaus muss die Funktion einen numerischen Rückgabewert besitzen. Dieser Rückgabewert enthält die berechnete Schätzung für die Laufzeit (Prozessortakte) bzw. die Standardabweichung. Damit eine derart aufgebaute Lookup Table in eine Laufzeitschätzung integriert werden kann, sind zum einen die Laufzeitfunktionen Matlab/Simulink zur Verfügung zu stellen, sowie eine Toolbox zu entwickeln, die die Modellanalyse mit der Ausführung der Laufzeitfunktionen kombiniert. Um die Implementierung eigener Laufzeitfunktionen zu erleichtern, kann ebenfalls eine API (Application Programming Interface) angeboten werden, mit deren Hilfe Modellinformationen abgefragt werden können. Wie eine in Matlab/Simulink eingebettete Toolbox zur Laufzeitschätzung mit Lookup Table und Modell in Beziehung steht bzw. welche Informationen verarbeitet werden, ist in Abbildung 18 dargestellt.

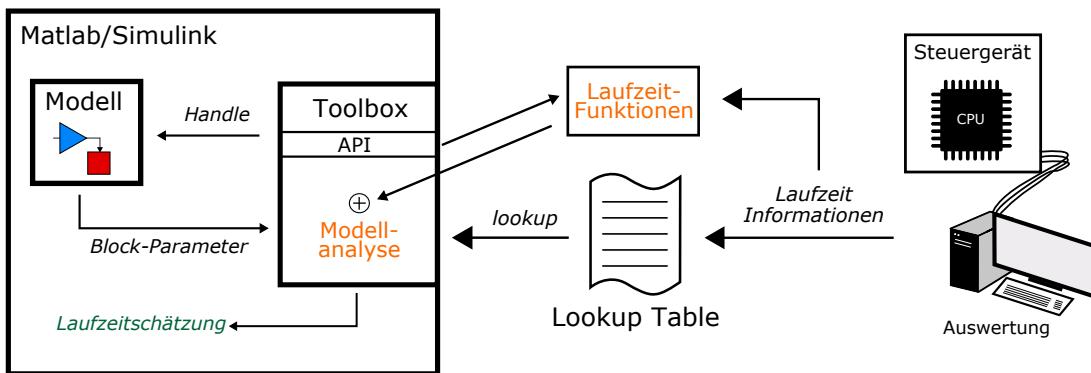


Abbildung 18: Zusammenspiel zwischen Modell, Toolbox und Lookup Table

Für den ersten Teil der Prozesskette, wie er in Abbildung 15 dargestellt ist, lässt sich für Simulink eine konkrete Ausprägung wie folgt zusammenfassen: Damit möglichst automatisiert und mit hoher Genauigkeit einzelne Simulink Funktionsblöcke hinsichtlich ihrer Laufzeit vermessen werden können, ist der *System Outputs* Funktionsblock mit entsprechenden Befehlen zur Nutzung von HPCs einzusetzen. Je nach Funktionalität und Einstellungsmöglichkeiten sind die Messergebnisse entsprechend auszuwerten. Hängt die Laufzeit eines Funktionsblocks von vielen Eingabeparametern ab, kann die Erstellung

²⁴Wichtig ist, dass diese Funktionen Matlab bekannt sind: rte ist dabei ein Ordner, in dem sich die Funktionen befinden (siehe Tabelle 7)

einer Matlab-Funktion sinnvoll sein. Über eine Toolbox wird die Lookup Table ausgewertet und mit entsprechenden Modellinformation verknüpft, um eine Laufzeitschätzung für ein gegebenes Modell zu berechnen. Das folgende Kapitel entwickelt den zweiten Teil der Prozesskette, der die Modellanalyse und den Aufbau einer Toolbox mit Lookup-Table-Auswertung beinhaltet.

4.3 Modellanalyse

Als zweiter Teil des Konzepts werden im Folgenden die Prozesse der Modellanalyse erarbeitet. Die Modellanalyse kombiniert die Laufzeitinformationen, welche durch die zuvor definierten Prozesse gewonnen werden, mit den aktuellen Modellinformationen. Der schematische Aufbau, wie er in Abbildung 16 dargestellt ist, soll folgend umgesetzt werden. Dazu wird zunächst eine Konzept für eine Toolbox entwickelt, die es ermöglicht möglichst automatisiert für ein ausgewähltes Modell eine Laufzeitabschätzung zu erstellen. Diese Toolbox soll überdies eine Schnittstelle bieten, um Konfigurationen der Analyse zu ermöglichen. Als weiteren Teil der Modellanalyse werden die Ablaufplanung sowie Schleifen in einem Simulink Modell untersucht.

4.3.1 Toolbox

Um eine möglichst genaue Laufzeitschätzung durchzuführen, ist zum einen zu untersuchen, welche Informationen über ein Modell benötigt werden, wie sie abgefragt werden können und auf welche Weise sie zu verarbeiten sind. Diese Aufgaben soll die im folgenden konzipierte Toolbox übernehmen.

Wie bereits in Kapitel 4.2 erläutert, sind die Einstellungsmöglichkeiten der verschiedenen Funktionsblöcken sehr unterschiedlich und damit ist auch die Berechnung der Laufzeiten von vielen unterschiedlichen Parametern abhängig. Die Anforderungsanalyse zeigt außerdem, dass die Häufigkeit der Verwendung der Blöcke stark variiert. Darüber hinaus ist bereits aufgezeigt, dass die am häufigsten verwendeten Funktionsblöcke *Import*, *Outport* oder *Subsystem* keinen Einfluss²⁵ auf die Laufzeit eines Programmes haben. Wie Abbildung 13 zeigt, sind die nächst häufigen Funktionsblöcke die Grundrechenarten *Product* (mit Division) und *Sum* (mit Subtraktion), der *Constant Block* sowie das Konvertieren (im Folgenden als “casten” bezeichnet) von Datentypen. Der *Constant Block* nimmt dabei eine gewisse Sonderrolle im Bezug auf die Laufzeit ein, da er an sich keine auszuführende Funktionalität implementiert, der Signalwert jedoch geladen werden muss und damit Einfluss auf die Laufzeit eines Programmes hat.

Um zu verstehen, welche Modell- bzw. Blockeinstellungen Einfluss auf die Codegenerierung und Ausführungsdauer haben, ist es vorteilhaft den generierten C-Code zu

²⁵Der äußerst geringe Einfluss, den (atomare) Subsysteme haben können wird hier vernachlässigt

betrachten, um etwaige Muster in der Codegenerierung zu erkennen und somit nicht alleinig von der Laufzeitmessung auf der Hardware abhängig zu sein. Es zeigt sich, dass der erzeugte Code für die Grundrechenarten von der Anzahl der Eingänge, deren Dimension und den Datentypen abhängt. Da sie sonst keine weiteren Einstellungsmöglichkeiten²⁶ haben, lässt sich die Abschätzung der Laufzeit mit diesen Informationen bereits hinreichend beschreiben. Dabei sind für diese Schätzung bereits Annahmen über die Codegenerierung zu machen. Zum einen wird stets die Zeit berechnet, die benötigt wird, um jeden Eingangswert zum größten Eingangsdatentyp zu casten. Außerdem wird angenommen, dass stets ein Cast vom Ergebnis zum Ausgangswert vorgenommen ist. Diese Annahmen sind dadurch legitimiert, dass sie stets den schlimmsten Fall annehmen und somit die tatsächliche Laufzeit eher überschätzen als unterschätzen. Die Abschätzung der Laufzeit kann wie folgt angegeben werden.

$$t_{est} = (C_{res \rightarrow out} + C_{input} + P \cdot R) \cdot D \quad (5)$$

Wobei $C_{res \rightarrow out}$ die Zeit für das Casten des Ergebnisses zum Ausgangsdatentyp, C_{input} die Zeit für das Casten der Eingangssignale, P der ‘‘Portfaktor’’, also die Anzahl an durchzuführenden Rechnungen und R die zuvor durch die Laufzeitmessung ermittelte Zeit für eine Rechnung angibt. Darüber hinaus gibt D den Wert der größten Dimension der Eingangssignale an. Als Beispiel habe ein Produkt-Block drei Eingänge, zwei vom Typ *int32*, einen vom Typ *double* und einen Ausgang vom Typ *int16*, so sind für eine Berechnung die zwei *int32* Eingänge zunächst zum Typ *double* zu casten (C_{input}). Anschließend werden die drei Eingangswerte multipliziert ($P \cdot R$) und das Ergebnis zum Ausgangsdatentyp konvertiert ($C_{res \rightarrow out}$). Sind ein oder mehrere Eingänge keine Skalare, so werden diese Schritte entsprechend häufig wiederholt (D). Die Zeit, die für das Laden von Operanden (Constant Block) benötigt wird, ist durch die Messung auf der Hardware implizit in R enthalten (siehe Kapitel 4.4).

Quellcode 2: Algorithmus zur Laufzeitabschätzung in Simulink

```

1 B = get_all_blocks(system);
2 total_est = total_std = 0;
3 For each block b in B
4     e = get_lookup_table_entry(b);
5     if is_numeric(e)
6         [est, std] = calc_regular(b, e);
7     else
8         [est, std] = eval(b, e);
9     total_est = total_est + est*loops(b);

```

²⁶Betrachtet wird nur die elementweise Berechnung

Um die Laufzeitwerte für die einzelnen Casts zu ermitteln, sind diese ebenfalls auf der Hardware zu vermessen und entsprechend in die Lookup Table mit aufzunehmen. Die Berechnung, wie sie in (5) dargestellt ist, wird im Folgenden als “regulär” bezeichnet (siehe Quellcode 2: *calc_regular*)

Mit dem Laufzeitwissen über die “*DataTypeConversions*” (Casts) und die Grundrechenarten ist bereits ein relativ großer Teil eines durchschnittlichen Modells abgedeckt. Jedoch können die Laufzeiten vieler weiterer Funktionsblöcke mit der zuvor erläuterten Formel nicht berechnet werden. Die Toolbox muss deshalb zwischen den einzelnen Blöcken weiterhin deutlich unterscheiden und entsprechende Modell- bzw. Blockinformationen individuell abfragen und verarbeiten. Matlab/Simulink bietet die Möglichkeit alle Informationen über ein (geladenes) Modell programmatisch abzufragen. Dazu zählen insbesondere Informationen zu Eingangs- und Ausgangssignalen, sowie alle vom Nutzer veränderbaren Einstellungen sowohl für jeden Block, als auch für das Modell allgemein. Der folgende abstrakte Algorithmus gibt an, wie die Laufzeitschätzungen zu berechnen sind. Bewusst ist in diesem Algorithmus nur die Akkumulation der Laufzeitschätzungen (*est*) bzw. Standardabweichungen (*std*) dargestellt. In den folgenden Kapiteln wird dieser Algorithmus, um die Definition der *loops*-Funktion sowie durch Betrachtungen des Kontrollflusses (if/else) erweitert. Quellcode 2 zeigt, dass je nach Eintrag in der Lookup Table die Berechnung der Laufzeit unterschiedlich vorgenommen wird. Ist der Eintrag ein numerischer Wert, so entspricht dieser *R* und die Schätzung ist mit der Formel (5) zu berechnen. Andernfalls ist der Eintrag zu evaluieren (Laufzeitfunktion wird aufgerufen).

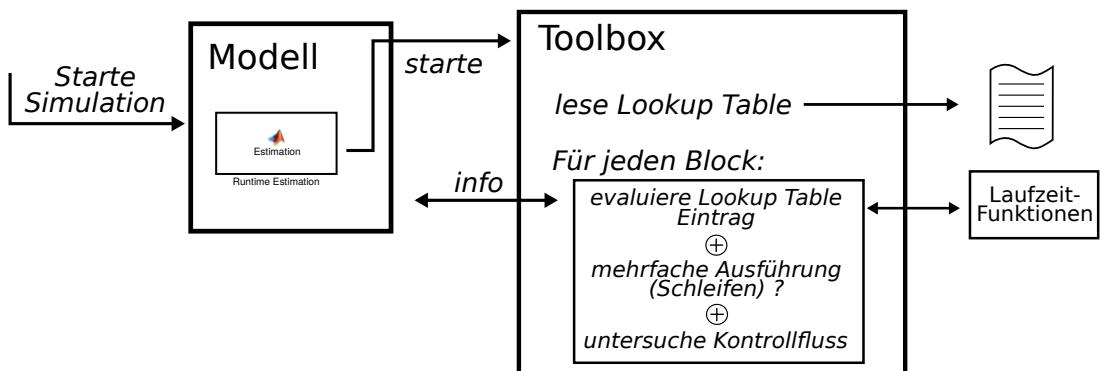


Abbildung 19: Toolbox Aufbau und Ablauf

Es ist zu beachten, dass ein Modell sich im Status der Simulation (run) befinden muss, damit Informationen über Signale ausgelesen werden können. Eine komplette Modellanalyse lässt sich folglich nur während einer laufenden Simulation erstellen. Damit stets sichergestellt ist, dass die Simulation läuft und das Modell geladen ist, ist der Einsatz eines Simulink Blocks (Matlab-Function Block) sinnvoll. Dieser soll die Laufzeitschätzung starten, wenn die Simulation startet. Eine solche Möglichkeit dient darüber hinaus der einfachen Bedienung, denn die Laufzeitschätzung muss nicht extra vom Funktionsentwickler angestoßen werden. Um der Anforderung nach einer Analyse verschiedener Designent-

scheidungen gerecht zu werden, kann die Laufzeitanalyse, abhängig davon in welchem Subsystem sich der Laufzeitblock befindet, ausschließlich auf dieses Subsystem angewendet werden. Als weiteren Bestandteil einer Toolbox sind Funktionen zur Auswertung der Lookup Table und zum Aufrufen und Ausführen der Laufzeitfunktionen zu nennen. Abbildung 19 stellt den schematischen Ablauf der Laufzeitanalyse innerhalb der Toolbox dar.

Die Toolbox verbindet alle zur Analyse benötigten Informationen und koordiniert ihren Ablauf. Sie stellt eine API zur Verfügung, um während der Simulation auf Modell- und Blockparameter zuzugreifen. Diese Schnittstelle kann von den Laufzeitfunktionen genutzt werden, um Abhängigkeiten aufzulösen und Laufzeitschätzer für Simulink Blöcke zu definieren. Sie bietet darüber hinaus die Möglichkeit mit Hilfe eines eigenen Funktionsblocks gezielt Subsysteme im Einzelnen zu untersuchen. Dabei wird durch eben diesen Block beim Start der Simulation die Laufzeitanalyse angestoßen. Berechnet wird die Laufzeit eines Modell mit der Auswertung der Lookup Table, sowie den entsprechenden Informationen aus dem untersuchten Modell.

In den folgenden zwei Kapiteln werden weitere Bestandteile der Laufzeit-Toolbox konzipiert und erläutert. Dabei werden der Kontrollfluss eines Simulink Modells sowie Mehrfachausführungen von Funktionsblöcken (Schleifen) hinsichtlich ihres Beitrags zur Laufzeit eines Programmes untersucht.

4.3.2 Schleifen

Simulink bietet mit *For Iterator Subsystems* und *While Iterator Subsystems* die Möglichkeit Funktionsblöcke wiederholt (je Simulationsschritt) auszuführen. Mehrfachausführungen (Schleifen) können einen bedeutenden Einfluss auf die Laufzeit eines Programmes haben. Der Algorithmus aus Quellcodebeispiel 2 zeigt bereits wie Schleifen in die Laufzeitschätzung mit einbezogen werden. Für jeden Funktionsblock, für den eine Laufzeit aus

Quellcode 3: Algorithmus zur Verzweigungsberechnung in Simulink

```

1 int loops(Blockhandle h) {
2     if not is_root(h) {
3         p = get_parent(h);
4         if is_iterator_subsystem(p)
5             return get_iter_limit(p)*loops(p);
6         return loops(p);
7     }
8     return 1;
9 }
```

den Informationen der Lookup Table berechnet wird, wird ebenfalls das Modell hinsichtlich der “Iterator Subsystems” untersucht. Für jeden dieser Subsysteme ist ein *Iterationslimit* festzulegen. Eine Endlosschleife ist nicht zulässig (vgl. Halteproblem, Kapitel 2.1.4). Durch ein Iterationslimit ist ein “loop bound” und somit auch eine “Worst-Case”-Ausführung gegeben. Die Laufzeitschätzung für eine einfache Ausführung eines Funktionsblocks (Lookup Table) ist mit dem Iterationslimit des “Iterator Subsystem” zu multiplizieren, in dem sich der Funktionsblock befindet (siehe Quellcode 2, Zeile 9). Es ist möglich Iterator-Subsysteme zu verschachteln. Dies ist bei der Berechnung der “loop bounds” zu beachten. In Quellcode 3 wird die bereits in Quellcode 2 angewendete Funktion “loops” definiert. Sie berechnet für einen gegebenen Funktionsblock wie häufig er ausgeführt wird. Da ein Simulink Modell als Hierarchie (baumartige Struktur), aufgebaut ist, kann ein “loop bound” durch den in Quellcode 3 definierten rekursiven Algorithmus berechnet werden. Gestartet wird auf der Ebene des zu untersuchenden Blocks. Es wird geprüft, ob das Subsystem in dem sich der Block befindet ein “Iterator Subsystem” ist. Ist dies der Fall, wird das Iterationlimit ausgelesen und die Funktion *loops* ruft sich selbst auf. Ist das Subsystem kein “Iterator Subsystem”, so wird nur eine Ebene Richtung Wurzel weiter gewandert. Dieser Vorgang wird so lange wiederholt, bis man an der Wurzel(oberste Ebene im Simulink Modell) angekommen ist.

4.3.3 Kontrollfluss

Wie bereits in den Grundlagen beschrieben, ist ein Simulink Modell ein sogenanntes Blockmodell, welches den Signalfluss abbildet. Es unterscheidet sich somit zu einem Kontrollflussgraphen (vgl. Abbildung 6), der den Ablauf eines Programmes darstellt. In einem Kontrollflussgraphen existiert stets ein fest definierter Startpunkt und es gibt keine nebenläufige Funktionsabarbeitung. Damit lässt sich ein Pfad vom Startknoten bis zu einem Endknoten festlegen und somit eine mögliche Abarbeitung des Programms definieren. Folglich ist es möglich einen “Worst-Case”-Pfad zu berechnen und aus diesem eine Laufzeit zu ermitteln. Dieses Verfahren ist für ein Simulink Modell nicht anwendbar, da sich für ein Signalflussmodell kein “Worst-Case”-Pfad ermitteln lässt bzw. ein solcher Pfad alle Knoten eines Modells beinhalten würde. Dies soll anhand der folgenden Abbildung 20 deutlich werden.

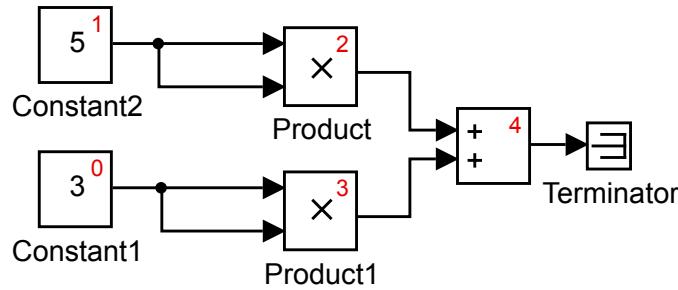


Abbildung 20: Einfaches Signalflussmodell mit Ausführungsreihenfolge

Abbildung 20 bildet die Berechnung $3^2 + 5^2$ ab. Deutlich wird, dass es keinen Pfad durch den Graphen gibt, der die Berechnung vollständig beschreibt. Für ein Ergebnis müssen stets alle Blöcke ausgeführt werden. Bei der Simulation liegt zu jedem Zeitschritt ein Wert an jedem Ausgang eines Blocks an. Im ersten Zeitschritt sind dies Initialwerte. Danach wird in jedem Zeitschritt, nach der in rot angegebenen Ausführungsreihenfolge, jede Blockfunktionalität ausgeführt. Dies überträgt sich ebenso auch auf den generierten C-Code und damit auf die Laufzeit des Programms.

Es gibt darüber hinaus Funktionsblöcke, die den Programmablauf des Codes beeinflussen können. Dazu zählen *Switches*²⁷, *Switch Case (mit Action Subsystems)*, *If (mit Action Subsystems)* und *Triggered Subsystems*. Dabei muss zwischen den Switches und den Subsystemen unterschieden werden. Wie in Abbildung 21 dargestellt ist, bieten Switches

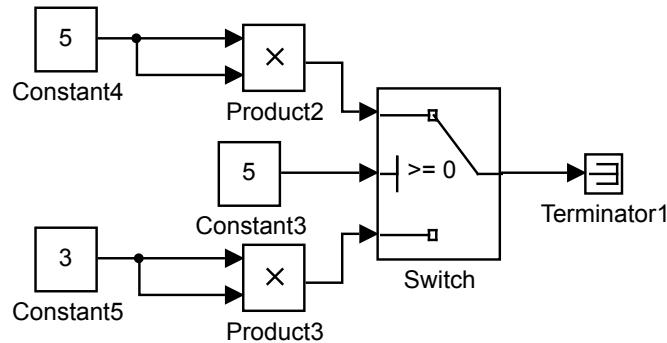


Abbildung 21: Beispiel für einen Switch Block

die Möglichkeit abhängig von einer Bedingung (mittlerer Eingang) entweder den oberen Eingangswert weiterzuleiten oder den unteren. In Programmcode abgebildet ergibt dies die folgende Struktur.

```
s=0; if (Constant3>=0) then s=Product2; else s=Product3;
```

Diese Implementierung führt entweder *Product2* oder *Product3* aus und hat damit Einfluss auf die Laufzeit des Programms. Es werden nun nicht mehr alle Funktionsblöcke berechnet, wie es in Abbildung 20 der Fall ist. Die Generierung eines solchen Codes

²⁷Switch, Multiport Switch, Manual Switch

lässt sich über die Einstellung “*Conditional Input Branch Execution*” erzeugen. Ist diese Einstellung nicht angewählt, so entspricht der erzeugte Quellcode dem Folgenden:

```
x1=Product2; x2=Product3; if (Constant3>=0) then s=x1; else s=x2;
```

Somit werden alle Blockfunktionalitäten ausgeführt. Die “*Conditional Input Branch Execution*” Einstellung ist eine Codeoptimierung und wird bei der Funktionsentwicklung meist ausgeschaltet, da ansonsten Signale, die im nicht ausgeführten Zweig liegen, statisch bleiben. Damit eine möglichst exakte Laufzeit geschätzt werden kann, muss die Modellanalyse jene Blöcke identifizieren, deren Funktionalität im generierten Code ausgeführt wird und welche nicht. Bei sich ändernden Eingangswerten für die Bedingung ist die Laufzeit des “Worst-Case”-Pfades in die weitere Berechnung zu übernehmen.

Die genannten Subsysteme erzeugen ebenfalls eine Verzweigung im generierten Code. Anders als beim Switch wird zuerst eine Bedingung auf ihren Wahrheitswert hin untersucht und anschließend das entsprechende Subsystem ausgeführt. Dadurch ist bereits durch die Struktur des Modells ersichtlich welche Form die Verzweigung im generierten Code hat. Abbildung 22 macht dies deutlich. Abhängig von der Eingangsgröße u_1 wird entweder das “If-Subsystem” oder das “Else-Subsystem” ausgeführt. Diese Subsysteme können wiederum Ein- bzw. Ausgänge besitzen (siehe Anhang Abbildung 37). Ein Switch Case Action Subsystem funktioniert auf die gleiche Weise wie das If Action Subsystem. Der Algorithmus aus Quellcodebeispiel 2 und 3 berechnet für alle Blöcke des gesamten

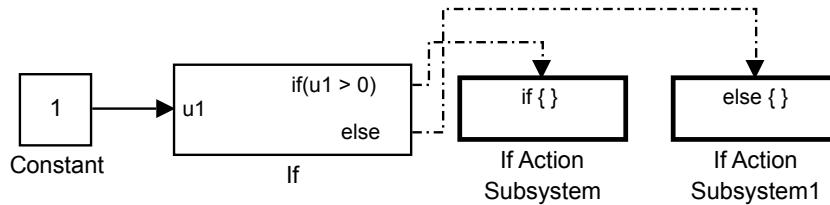


Abbildung 22: Aufbau einer If/else Verzweigung in Simulink

Systems eine akkumulierte Laufzeitabschätzung. Mit der Einführung von Verzweigungen gibt es Bereiche, die teilweise (je nach Programmablauf) nicht ausgeführt werden und somit auch nicht zur (“Worst-Case”)-Laufzeit beitragen. Für die Funktionsblöcke, die aus “Action Subsystems” aufgebaut werden bedeutet dies, dass alle Verzweigungen (Subsysteme) hinsichtlich ihrer Laufzeit zu untersuchen sind und jene Laufzeiten von der berechneten Gesamtlaufzeit $total_est$ zu subtrahieren sind, welche nicht den “Worst-Case”-Pfad abbilden. Für das Beispiel in Abbildung 22 bedeutet dies, dass die Laufzeit des “If Action Subsystems” mit dem “Else Action Subsystem” zu vergleichen ist und der kleinere Laufzeitwert vom Gesamtwert $total_est$ abzuziehen ist. Das Vorgehen für Verzweigungen ist in Quellcode 4 als Pseudocode abgebildet. Dabei ist $runtime_est(system)$, die Funktion, die die Laufzeitschätzung für alle Blöcke durchführt (Quellcode 2, 3). Die

Quellcode 4: Algorithmus zur Verzweigungsberechnung in Simulink

```

1 total_est = runtime_est(system);
2 B = find_all_branch_blocks(system);
3 For each block b in B
4   if is_switch(b)
5     ex = calc_excluded_blocks(b);
6   else
7     ex = sum(runtime_est(get_not_wc_blocks(b)));
8   total_est = total_est - ex;

```

Funktion *calc_excluded_blocks(b)* berechnet die Laufzeiten für die durch einen Switch nicht ausgeführten Funktionsblöcke. Ist der Verzweigungsblock *b* kein Switch so wird mit *get_not_wc_blocks(b)* alle Subsysteme identifiziert, die nicht die längste Ausführungszeit (not worst-case (not_wc)) haben. Die Summe der Laufzeiten dieser “Action Subsystems” wird abschließend von der Gesamtlaufzeit subtrahiert.

Die Toolbox ist mit der Beschreibung einer Lookup Table, deren Auswertung sowie einer Modellanalyse vollständig konzipiert. Eine Umsetzung dieser Toolbox soll es dem Funktionsentwickler ermöglichen, auf abstrakter Ebene und früh in der Modellentwicklung, eine erste Aussage zur Laufzeit des entwickelten Modells zu treffen. Abschließend werden im folgenden Kapitel die in den Grundlagen vorgestellten Hardwareprozesse hinsichtlich ihres Einflusses auf das in diesem Kapitel entwickelte Konzept untersucht.

4.4 Cache- und Pipeling-Einflüsse

Da die auf Modellebene entwickelten Funktionen nicht nur simuliert, sondern nach der Entwicklung direkt auf der Hardware ausgeführt werden, spielen Speicherverwaltung (Caching) und das Pipelining ebenfalls eine Rolle in der Laufzeitanalyse.

Die bereits in den Grundlagen erwähnte “Abstrakte Interpretation” von Quellcode wird in einigen Programmen genutzt, um Aussagen über die Speichernutzung und die Abarbeitungsreihenfolge zu treffen. Eine solche Interpretation setzt voraus, dass das zu untersuchende Programm in einer entsprechend hardwarenahen Codierung vorliegt. Andernfalls ist eine Zuordnung zwischen auszuführenden Operationen und der ausführenden Hardware kaum möglich. Darüber hinaus ist sowohl das Caching als auch das Pipelining von der Ausführungsreihenfolge der einzelnen Programmabschnitte abhängig. Kapitel 4.3.3 zeigt, dass durch ein Signalfluss-Modell keine exakte Ausführungsreihenfolge vorgegeben ist (vgl. Abbildung 20) und damit eine Vorhersage über die Speicherausnutzung und Programmausführung deutlich erschwert ist. Hier kann das Wissen über die Codegenerierung die entsprechenden Informationen liefern. Indem bekannt ist, wie aus der

graphischen Beschreibung Quellcode erzeugt wird, kann die Struktur des erzeugten Programmes ermittelt werden.

Unabhängig vom Programmablauf kann die Cache-Nutzung für jeden Funktionsblock einzeln betrachtet und diese Information mit in die Laufzeitabschätzung einbezogen werden. Wie frequentiert während der Ausführung eines Blocks auf den Cache zugegriffen werden muss, hängt dabei stark vom aktuellen Zustand des Caches ab (vgl. Kapitel 2.1.3). Da bei der Laufzeitmessung, so wie sie in dieser Arbeit dargestellt ist, jeweils nur Teilfunktionalitäten vermessen werden, ist die Cachebelegung während der Messung nicht repräsentativ für die tatsächliche Ausführung innerhalb eines Programms. Indem man den “Worst-Case” Fall annimmt, ist jedoch eine obere Schranke definierbar. Das “Worst-Case”-Szenario entspräche einem Zugriff auf den Hauptspeicher für jede auszuführende (Assembler-)Operation. Eine solche Annahme für alle Funktionen führt während der Laufzeitabschätzung für fast alle Betrachtungen zu einer deutlichen Überschätzung der Laufzeit und ist dementsprechend unbrauchbar. Unter der Annahme, dass ein Hauptspeicherzugriff äußerst selten im Gegensatz zur Trefferrate eines Caches ist, lässt sich das “Worst-Case”-Szenario hinsichtlich der Überschätzung abschwächen. Sind Trefferraten der Caches bekannt, so können diese zur Abschätzung der Cache-Zugriffe je Funktionsblock genutzt werden.

Da die Einflüsse der Speicherverwaltung und des Pipelinings stark von der tatsächlichen Struktur des auszuführenden Binärprogramms abhängen, können Abschätzungen auf Modellebene nur äußerst ungenau ausfallen. Aussagen über einzelne Funktionsblöcke sind ebenfalls nicht repräsentativ, da initiale Cache- und Pipelinebelegungen während der Ausführung des Programmes nicht bekannt sind. Wird die Codegenerierung nicht als “Black-Box-System” betrachtet, so lässt sich das Wissen über die Umsetzung des Modells zu Quellcode nutzen, um genauere Vorhersagen über hardwarenahe Vorgänge zu treffen.

4.5 Zusammenfassung

Das Konzeptkapitel umfasst die Entwicklung einer Methodik zur Laufzeitanalyse auf Modellebene. Dabei wird die konzipierte Prozesskette in zwei Abschnitte unterteilt. Zunächst wird auf der Grundlage der Anforderungsanalyse diskutiert, inwiefern Laufzeitaussagen auf abstrakter Ebene getroffen werden können. Ferner werden Methoden zur Laufzeitmessung untersucht und eine Modellanalyse entworfen. Auf einer abstrakten Beschreibung der Prozesskette aufbauend, werden die einzelnen Teilprozesse entwickelt. Dazu zählt die Beschreibung der Laufzeitmessung sowie die Integration in die Codegenerierung durch Simulink, die Entwicklung einer Lookup Table sowie die Konzeption einer in Matlab/Simulink eingebetteten Toolbox.

Die Modellanalyse untersucht die für die Laufzeitanalyse wichtigen Modelleigenschaften, wie Kontrollfluss-Mechanismen oder Mehrfachausführungen (Schleifen) und entwirft

entsprechende Algorithmen zur Auswertung dieser Informationen. Die über die Toolbox implementierte Modellanalyse kombiniert die Werte der Laufzeitmessung (Lookup Table) mit den Informationen, die für die Funktionsblöcke zur Verfügung stehen, zu einer abschließenden Laufzeitanalyse. Dabei besteht das besondere Moment darin, dass zur Ausführung einer Laufzeitschätzung keine Codegenerierung und Kompilierung mehr nötig sind. Das Gesamtschema des Konzepts ist in Abbildung 42 (Anhang) dargestellt. Neben der Laufzeitmessung für Simulink Funktionsblöcke und der Modellanalyse ist die Evaluierung als weiterer Abschnitt mit aufgenommen.

Das folgende Kapitel erläutert die prototypische Umsetzung des Konzepts. Diese Implementierung soll anschließend die Möglichkeit geben, die Umsetzung der definierten Anforderungen zu überprüfen und somit das Konzept zu evaluieren.

5 Implementierung

Dieses Kapitel dokumentiert die prototypische Umsetzung des zuvor erarbeiteten Konzepts. Die Implementierung umfasst die Laufzeitmessung von Simulink Funktionsblöcken auf einem Prototypensteuergerät, die Erstellung eines Lookup Tables sowie die Implementierung einer in Matlab/Simulink integrierten Toolbox zur Ausführung der Laufzeitanalyse.

5.1 Versuchsaufbau

Zur prototypischen Umsetzung des Konzepts wird als Entwicklungsplattform ein Computer mit Windows 7 als Betriebssystem und als Steuergerät die *MicroAutoBox*, ein Entwicklungssteuergerät (Prototypensteuergerät) von dSpace, verwendet. Typischerweise ist ein solches Steuergerät deutlich leistungsstärker als ein tatsächliches Motorsteuergerät (siehe Tabelle 2 u. 8). Zur Messung der Laufzeit wird auf dem Host-Rechner das Pro-

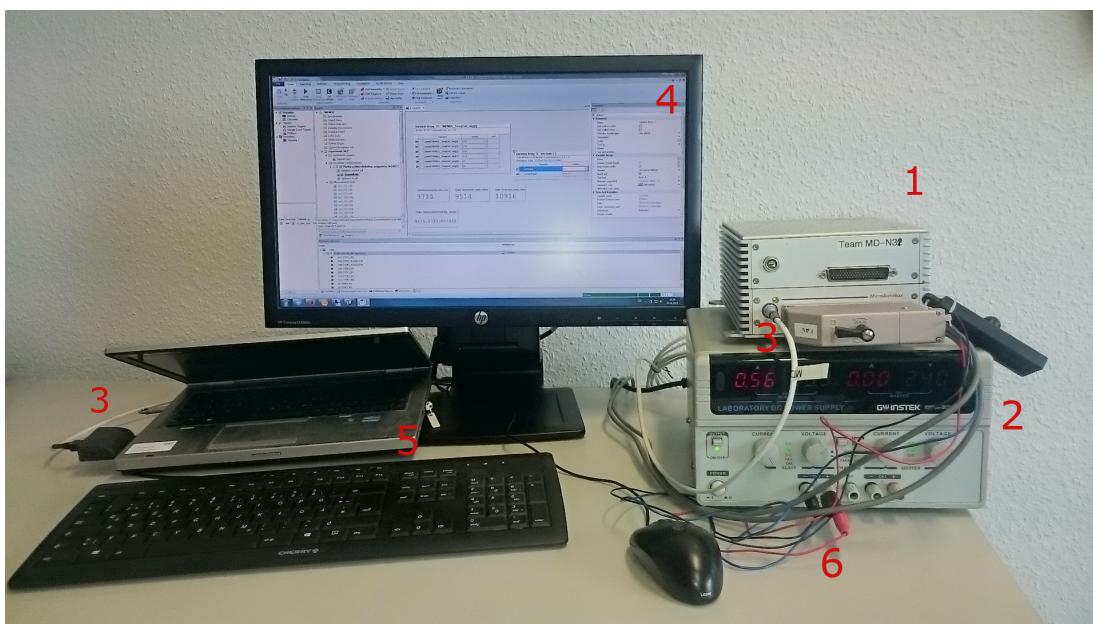


Abbildung 23: Versuchsaufbau zur Laufzeitmessung

1 - MicroAutoBox (Steuergerät), 2 - Netzgerät, 3 - dSpace DS821 ExpressCard Link Interface (High Speed Serial Link), 4 - ControlDesk, 5 - Host Computer, 6 - Stromversorgung: 24V (rot), Ground (schwarz), "Klemme 15" (blau)²⁸

gramm *ControlDesk*, ebenfalls ein Produkt von dSpace, genutzt. Es ermöglicht ein umfassendes Monitoring des Programmablaufs auf der MicroAutoBox. Als Dateiformat für die Lookup Table wird XLSX (Microsoft Excel) gewählt. Die gesamte Implementierung der Toolbox ist in Matlab/Simulink der Version R2013b umgesetzt. Die MicroAutoBox ist

²⁸(Im Fahrzeug) geschaltetes Plus vom Zündstartschalter (Zündungsplus)

über eine serielle Schnittstelle (dSpace DS821 ExpressCard Link Interface) mit dem Host-Computer (Laptop) verbunden. Über diese Verbindung kann die gesamte Funktionalität des Steuergeräts überwacht und der Programmablauf beeinflusst werden. Genutzt wird dazu das Programm *ControlDesk* in der Version 5.5. Die MicroAutoBox wird über ein Netzgerät mit 24V (Gleichstrom) versorgt. Damit die MicroAutoBox betrieben werden kann, ist eine weitere Leitung auf 24V zu schalten. Diese Leitung ist das Zündungsplus (Klemme 15). Abbildung 23 zeigt den beschriebenen Versuchsaufbau. Tabelle 8 listet noch einmal detailliert auf, welche Soft- und Hardware verwendet wird.

Tabelle 8: Versuchsaufbau

MicroAutobox 1404/1505/1507, Type: 800MHz 16 MByte (siehe Tabelle 2), BaseBoard: DS1401-18, I/O Boards: DS1505-01, DS1507-02
Netzgerät GW Instek GPS-2303
Host-Computer (Laptop), Intel Core i7-3520M CPU 2.90GHz, 8GB RAM, ExpressCard/54 Link Interface
Windows 7 Enterprise, 64 Bit-Betriebssystem, Service Pack 1
ControlDesk Version 5.5 (dSpace)
MATLAB/Simulink R2013b (8.2.0.701) 64-bit
Microsoft Excel (Microsoft Office Professional Plus 2013) 32-Bit

5.2 Umsetzung

Als erster Teil der Umsetzung wird im Folgenden anhand des Versuchsaufbaus (Abbildung 23) die Vermessung der einzelnen Simulink Funktionsblöcke erläutert. Dies umfasst die Nutzung des sogenannten “Performance Monitors” des PowerPC 750GL Prozessors in Kombination mit der ControlDesk-Experimentiersoftware für dSpace Steuergeräte. Auf den Messdaten aufbauend, wird ferner der Aufbau der Lookup Table dargestellt. Der zweite Teil der Umsetzung widmet sich der Implementierung der Modellanalyse in Matlab/Simulink, sowie der Entwicklung der Toolbox. Diese Toolbox wertet entsprechend des zu untersuchenden Modells die Einträge der Lookup Table aus und kombiniert diese mit den Ergebnissen aus der Modellanalyse abschließend zu einer Laufzeitabschätzung.

Als Randbedingung ist anzumerken, dass die Laufzeitanalyse, wie sie in dieser Arbeit implementiert ist, die Verwendung von “S-Funktion” Blöcken und “Matlab-Functions” ausschließt bzw. diese nicht in die Analyse mit einbezieht. Eine Laufzeitabschätzung anhand von Quellcode entspricht nicht dem Konzept dieser Arbeit (Analyse auf Modellebene). Um die Funktionalität jener Funktionsblöcke in der Laufzeitanalyse mit einzubeziehen, sind diese äquivalent mit Standardblöcken der Simulink-Bibliothek nachzubauen. Des Weiteren werden Einstellungen zu Code-Optimierungen während der Codegenerierung nicht mit in die Berechnung der Laufzeitanalyse einbezogen.

5.2.1 Laufzeitmessung der Funktionsblöcke

Um einzelne Funktionsblöcke hinsichtlich ihrer Laufzeit vermessen zu können, sind diese möglichst isoliert zu betrachten. Simulink bietet über die Einstellung “Treat as atomic Unit” die Möglichkeit, Subsysteme als atomare Ausführungsblöcke zu erstellen. Dies hat zur Folge, dass bei der Festlegung der Ausführungsreihenfolge diese Subsysteme als Einheit angesehen werden. Diese Einstellung überträgt sich ebenso auf die Codegenerierung. Das Subsystem enthält ausschließlich den zu vermessenen Block (nebst “System-Outputs Block”, vgl. Abbildung 17). Eingangssignale werden über Imports dem Funktionsblock zur Verfügung gestellt. Um ein möglichst breites Spektrum zu messen und etwaige Laufzeitschwankungen zu detektieren, sind die Eingangssignale möglichst stark zu variieren. Dies wird durch gleichverteilte Zufallszahlen erreicht (“Uniform Random Number”-Funktionsblock).

Wie in Abbildung 17 dargestellt, wird innerhalb des Subsystems durch die Nutzung des “System-Outputs”-Blocks das Einfügen von eigenem Quellcode während der Codegenerierung ermöglicht. Diese Funktionalität ist zum Starten und Auslesen der Zählregister zu nutzen. Der in der MicroAutoBox verbaute Prozessor *PowerPC 750GL* bietet vier dieser Zählregister (PMC1-4) an. Diese vier “Performance Monitor Counters” (PMC) zählen das Auftreten verschiedener Vorgänge auf der Hardware. Welche Vorgänge zu zählen sind, wird über zwei “Monitor Mode Control Registers” (MMCR) gesteuert. Sowohl PMC als auch MMCR sind 32-Bit Register und lassen sich über den Assemblerbefehl *mtspr*²⁹ beschreiben bzw. mit *mspr*³⁰ auslesen. Um die Laufzeit eines Programms zu messen, sind die MMCR Register so zu beschreiben, dass Prozessorzyklen gezählt werden. Die Bits 19-25 des MMCR0 geben an, welches Ereigniss im PMC1 Register aufgenommen wird. Damit die Prozessortakte gezählt werden, ist Bit 25 des MMCR0 auf 1 und der

<u>Performance Monitor Registers</u>	
Performance Counters	Sampled Instruction Address
PMC1 SPR 953	SIA SPR 955
PMC2 SPR 954	
PMC3 SPR 957	
PMC4 SPR 958	

Monitor Control	
MMCR0 SPR 952	
MMCR1 SPR 956	

Abbildung 24: Special Purpose Registers (SPR) des IBM PowerPC 750GL [IBM06]

MMCR0								
0..0	0	0	0	0	0	0	1	0..0
0.18	19	20	21	22	23	24	25	26..31

Rest des Registers auf 0 zu setzen. Eine solche Bit-Belegung des MMCR0 Registers entspricht der Dezimalzahl 64 (big-endian). Sobald das Control-Register beschrieben ist,

²⁹Move to special purpose register

³⁰Move from special purpose register

werden die Prozessortakte in PMC1 aufsummiert. Damit korrekte Wert gespeichert werden, ist das PMC1 Register vor dem Start zurückzusetzen.

Nach der Ausführung des zu messenden Quellcodes ist das Zählregister auszulesen (mfspr). Damit es möglich ist, die ausgelesenen Daten in ControlDesk zu analysieren und zu verarbeiten, ist im Modell ein *Data Store Memory* Block zu erstellen (*subsystem_exec_time*). Dieser Block erzeugt eine Variable, die beschrieben und ausgelesen werden kann. Sie wird genutzt, um die Werte des PMC1 Registers zu speichern und dem ControlDesk Programm auf dem Host-Computer zugänglich zu machen.

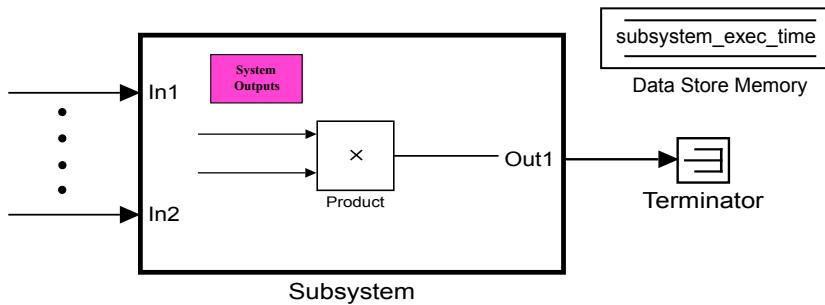


Abbildung 25: Aufbau des Messmodells

Abbildung 25 zeigt den Aufbau eines Messmodells. Als Eingangssignale für das Subsystem werden verschiedene Parameter und Wertebereiche gewählt, um ein möglichst großes Spektrum an Laufzeiten zu bestimmen (siehe Anhang Abbildung 37). Diese Signale sind als Blöcke außerhalb des Subsystems definiert. Es befindet sich nur der zu vermessene Funktionsblock innerhalb des Subsystems, um eine exakte Analyse der Laufzeit zu gewährleisten. Der “System-Outputs”-Block definiert die vor und nach der Ausführung der Funktion einzufügenden Quellcode-Sequenzen (siehe Anhang Abbildung 35). Für den PowerPC 750GL Prozessor der MicroAutoBox ergeben sich diese Sequenzen wie folgt.

Quellcode 5: Assembler-Sequenz vor Block-Quellcode

```

1 asm("li 3,0\n mtspr 952,3\n mtspr 953,3\n");
2 asm("li 3,64\n mtspr 952,3\n");

```

Zunächst wird das Register 3 auf 0 gesetzt (li³¹ 3,0). Anschließend werden die SPR 952 und 953 mit dem Wert aus Register 3 zurückgesetzt. Um die Zählung zu starten, ist das MMCR0 Register auf 64₁₀ zu setzen. Dies geschieht erneut über das Register 3. Nach der Ausführung dieser Sequenz wird bei jedem Prozessortakt der Wert des Registers PMC1 (SPR 953) um 1 erhöht. Die Assembler-Sequenz aus Quellcode 5 ist im “System-Outputs”-Block im Textfenster für *System Outputs Function Execution Code* anzugeben.

³¹load immediate

Die Folgende Assembler-Sequenz in Quellcode 6 ist im Textfenster für *System Outputs Function Exit Code* anzugeben (siehe Anhang Abbildung 35).

Quellcode 6: Assembler-Sequenz nach Block-Quellcode

```

1  asm("mfspr 3,953\n") ;
2  asm("addis 4,0,ha(subsystem_exec_time)\n") ;
3  asm("stw 3,lo(subsystem_exec_time)(4)\n") ;

```

Nach der Ausführung des Funktionsblocks wird das Zählregister 953 (PMC1) mit dem Befehl *mfspr* ausgelesen und in Register 3 gespeichert. Um den Wert in Register 3 dem ControlDesk Programm zugänglich zu machen, ist dieser in der Variable *subsystem_exec_time*, die zuvor im Simulink Modell durch den “Data Memory Store”-Block angelegt wurde, zu speichern. Dazu wird mit dem Befehl *addis*³² die “oberen” 16 Bit der Adresse der *subsystem_exec_time* Variable geholt und in Register 4 gespeichert. Die “unteren” 16 Bit des Registers 4 sind anschließend auf 0 gesetzt. Der letzte Befehl der Sequenz (*stw*³³) speichert den Wert aus Register 3 an die Adresse, die sich aus dem Wert in Register 4 ergibt, plus einem Offset von *lo(subsystem_exec_time)*. Dabei entspricht der Offset genau den “unteren” 16 Bit der Adresse von *subsystem_exec_time*. Da bei der Codegenerierung C-Code erstellt wird, sind die Assemblerbefehle zur Laufzeitmessung mit der Funktion *asm* einzubetten. Diese Funktion fügt an der entsprechenden Stelle im vom Compiler erzeugten Assemblercode die angegebenen Codezeilen ein.

Damit nicht für jeden Funktionsblock stets ein gesamtes Modell in ein Binärprogramm umgesetzt werden muss, kann das Messmodell wie in Abbildung 37 (Anhang) gezeigt aufgebaut werden. Über einen “Switch”-Block können verschiedene “Action-Subsystems” getriggert werden. Je nach Aufbau des Funktionsblocks können andere Eingangsparameter für das Subsystem gewählt werden. Über ControlDesk lässt sich während der Ausführung des Programms auf den steuernden “Constant”-Block *M_ID* zugreifen. Zur Laufzeit kann dementsprechend das Signal dieses Blocks verändert werden. So ist es möglich zwischen den Subsystemen zu wechseln. Dabei enthält jedes Subsystem einen “System-Outputs”-Funktionsblock mit den zuvor beschriebenen Assemblerbefehlen.

Ist der Host-Computer wie im Versuchsaufbau dargestellt mit der MicroAutoBox verbunden, so kann über das Programm ControlDesk auf den Programmablauf Einfluss genommen werden (siehe Anhang Abbildung 38). Zunächst ist das kompilierte Simulink Modell über jenes Programm auf das Steuergerät zu übertragen (flashen). Während der Ausführung kann anschließend die zuvor als “Data Memory Store”-Block angelegte Variable ausgelesen und angezeigt werden. Über “Recorder” können Signale angegeben und über definierte Zeit aufgenommen werden. Bei stark schwankenden Laufzeiten ermöglicht dies das Aufnehmen einer Messreihe, die anschließend ausgewertet werden kann. Ein

³²add immediate shifted

³³store word

Beispiel für eine solche Auswertung ist in Abbildung 36 (Anhang) für den “1-D Lookup Table” dargestellt. Die über “Recorder” aufgezeichneten Messreihen, lassen sich als “.mat”-Datei exportieren und somit in Matlab weiter weiterverarbeiten. Anhand des in Abbildung 38 (Anhang) abgebildeten Ausschnitts lässt sich der Aufbau von ControlDesk nachvollziehen.

Mit Hilfe der Laufzeitmessung und der individuellen Auswertung der Ergebnisse lässt sich für jeden Funktionsblock ein Laufzeitschätzer ermitteln. Als Besonderheit bei der Codegenerierung in Simulink ist zu beachten, dass die Funktionalität des Modells in mehrere C-Funktionen unterteilt wird. Unter anderem sind dies die Output-Funktion sowie die Update-Funktion. Für die Laufzeit von Bedeutung ist insbesondere die Output-Funktion. Sie umfasst die gesamte Funktionalität, die vom Simulink Modell in jedem Simulationsschritt abgebildet wird. In einigen Fällen wird eine Teifunktionalität eines Simulink Blocks ebenfalls in der Update-Funktion ausgeführt. Dies ist beispielsweise für “Delay”-Blöcke der Fall. In der Update-Funktion wird der vorige Signalwert mit dem aktuellen ersetzt. Um einen korrekten Laufzeitschätzer zu erstellen, ist dementsprechend die Messung in beiden Funktionen vonnöten.

Diese Schätzer werden anschließend in die Lookup Table aufgenommen. Die Lookup Table dieser Implementierung unterscheidet drei verschiedene Bereiche: “Datatyps”, “Common” und “DataConversion”. Diese Bereiche sind jeweils als “Sheets” (eigene Tabelle) einer Excel-Datei umgesetzt. Für jeden Datentyp, der in Simulink zur Verfügung steht, wird eine Tabelle angelegt. In diesen Tabellen werden die Blöcke abgespeichert, deren Laufzeit vom verwendeten Datentyp der Eingangssignale abhängt (siehe Anhang Abbildung 39). In die Tabelle des Bereichs “Common” werden alle anderen Funktionsblöcke aufgenommen (siehe Anhang Abbildung 41). Der dritte Bereich listet die Laufzeitschätzungen für die jeweiligen Datentypumrechnungen auf. Die Umrechnungsangaben sind als reguläre Ausdrücke der Form “*src_dt* → *dest_dt*” angegeben (siehe Anhang Abbildung 40). Das Excel-Dateiformat wird, aufgrund der Möglichkeit mehrere Tabellen in einer Datei speichern zu können und der bereits bestehenden Lese- und Schreibfunktionalitäten die durch Matlab zur Verfügung gestellt werden, gewählt.

Das folgende Kapitel erläutert die Implementierung und den Aufbau der Toolbox. Die als Excel-Datei aufgebaute Lookup Table wird in dieser Implementierung genutzt, um mit Hilfe der Modellanalyse eine Laufzeitabschätzung zu berechnen.

5.2.2 Implementierung der Toolbox

Die Toolbox zur Laufzeitabschätzung von Simulink Modellen besteht aus verschiedenen Matlab-Funktionen, sowie einem Simulink Funktionsblock. Durch den Funktionsblock wird festgelegt, welcher Teil eines Modells untersucht wird. Während der Simulation eines Modells ruft dieser Block die entsprechenden Matlab-Funktionen zur Auswertung der Modellanalyse und der Lookup Table auf.

Die Toolbox ist wie folgt aufgebaut: Alle Funktionen der Laufzeitschätzung sind unter *rte* (runtime estimation) gruppiert. Hier befinden sich auch die zwei einzigen Funktionen, die direkt vom Nutzer aufgerufen werden müssen. Zum einen ist dies *rte.init(<table.xlsx>)*. Diese Funktion initialisiert die Laufzeitanalyse indem sie die angegebene Excel-Tabelle ausliest und die Informationen in einem globalen Objekt speichert. Während der Analyse wird dieses Objekt genutzt, um die spezifischen Laufzeitwerte für die einzelnen Funktionsblöcke abzufragen. Die zweite Funktion ist *rte.runtime_estimate(<system>)*. Diese Funktion startet die Analyse für ein gegebenes System und wird durch den Funktionsblock der Toolbox während der Simulation aufgerufen.

Hilfsfunktionen befinden sich unter *rte.util*. Sie übernehmen Aufgaben, wie das Auslesen von Blockparametern (*get_block_parameter*), das Berechnen von Schleifendurchläufen (*get_iterations*), oder überprüfen, ob die Simulation gestartet ist. Diese Implementierung wird von den Funktionen der einzelnen Blöcke genutzt, um Informationen über das Simulink Modell abzufragen. Für jeden Funktionsblock existiert unter *rte.blocks* eine eigene Funktion. Diese berechnet individuell die Laufzeit für einen gebenen Block. Dazu beziehen sie die Informationen aus dem Modell und der Lookup Table mit ein. Zum Auslesen und Evaluieren von Loop-Table-Einträgen umfasst die Implementierung unter *rte.xls* weitere Funktionen. Diese geben die Werte “Prozessortakte” und “Standardabweichung” des entsprechenden Loop Table Eintrags zurück bzw. führen die angegebene Laufzeitfunktionen aus. Die Laufzeitfunktionen für den PowerPC 750GL Prozessor sind unter *rte.target* gespeichert. Als letzte Gruppierung existiert *rte.estimate*. Diese umfasst zwei Funktio-

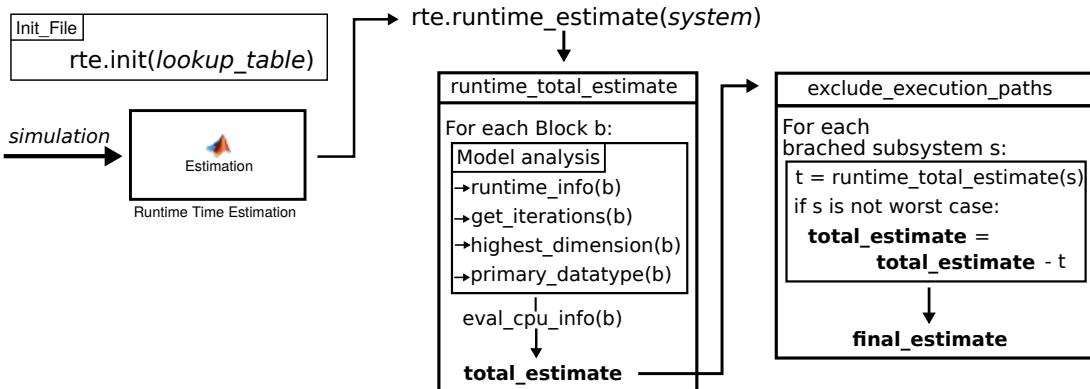


Abbildung 26: Ablauf der Laufzeitanalyse (Funktionsaufruf-Schema)

nen, die die gesamte Laufzeitanalyse ausmachen (siehe Abbildung 26). Beide Funktionen werden durch *rte.runtime_estimate* aufgerufen. Dabei wird durch *runtime_total_estimate* die Summe aller Blocklaufzeiten gebildet. Anschließend entfernt *exclude_execution_paths* die Laufzeiten aus der Berechnung, die nicht zu einer “Worst-Case”-Ausführung gehören (vgl. Kapitel 4.3.3). Aufgeführt ist stets nur die Laufzeitabschätzung *total_estimate* bzw. *final_estimate*. Es ist zu beachten, dass bei der Berechnung der Schätzung ebenfalls die Standardabweichungen der Messungen aufsummiert werden.

Wie die jeweiligen Funktionen in Beziehung stehen, ist in Abbildung 26 in einem Ablaufschema der Toolbox dargestellt. Bevor die Analyse gestartet wird, ist die Initialisierung durchzuführen. Gestartet wird die gesamte Analyse durch den Start der Simulation. Die sich im Modell befindende Matlab-Funktion “Estimation” ruft die Funktion `rte.runtime_estimate(system)` auf. Dabei beschreibt der Übergabeparameter “*system*” das Subsystem, in dem sich der “Runtime Estimation” Block befindet. Während der Laufzeit-

```
----- RESULT -----
The execution time estimate for 'TMPMDL' is:
10818.45 clock cycles
with error of 600.06
-----
fx >>
```

Abbildung 27: Matlab-Konsolenausgabe nach der Laufzeitanalyse

analyse wird in der Matlab-Konsole zu jedem untersuchten Funktionsblock ausgegeben, wie viele Prozessortakte die Berechnung benötigt, sowie die entsprechenden Schwankung um diesen Wert. Das Endresultat ist ebenfalls aus der Matlab-Konsole zu entnehmen (Abbildung 27).

5.3 Zusammenfassung

In diesem Kapitel wird die Umsetzung, der im Kapitel 4 entwickelten Prozesskette zur Bestimmung einer Laufzeitanalyse auf Modellebene, erläutert. Die konzipierte Laufzeitmessung für Simulink Funktionsblöcke wird mit dem Prototypen-Steuengerät “MicroAutoBox” sowie der Experimentier-Software ControlDesk umgesetzt. Mit Hilfe des “Performance Monitors” können Prozessortakte exakt gezählt werden. Darüber hinaus lassen sich mit dem “System-Outputs”-Block Assembler-Befehlssequenzen so in den generierten Code einfügen, dass eine Steuerung der Zählregister möglich ist. “Memory Data Store”-Blöcke im Messmodell ermöglichen es, die Registerwerte in ControlDesk einzusehen und auszuwerten. Die vollständig in Matlab-Code geschriebene Toolbox liest zu Beginn der Analyse die Werte der Lookup Table aus und verarbeitet mit Hilfe der implementierten Matlab-Funktionen die individuellen Informationen jedes Funktionsblocks zu einer Laufzeitabschätzung. Je nach Bedarf kann die Analyse durch Versetzen des “Runtime Estimation”-Blocks angepasst werden.

Die in diesem Kapitel erläuterte Umsetzung, der zuvor konzipierten Prozesse, soll im folgenden Kapitel die Grundlage für eine abschließende Betrachtung und Evaluierung des Konzepts bieten. Dabei wird untersucht, inwieweit das Konzept sowie die Implementierung die aus der Analyse ermittelten Anforderungen erfüllen.

6 Evaluierung

Das in dieser Arbeit erarbeitete Konzept zur Laufzeitanalyse von Simulink Modellen soll in diesem Kapitel evaluiert werden. Anhand drei verschiedener Modelle wird der Ansatz der Laufzeitschätzung auf Modellebene untersucht. Die Modelle unterscheiden sich hinsichtlich ihrer Komplexität, der Laufzeit sowie der verwendeten Funktionsblöcke.

6.1 Testmethodik

Welche Güte eine Laufzeitschätzung anhand des in dieser Arbeit vorgestellten Konzepts erreicht werden kann, soll im Folgenden durch den Vergleich zwischen der geschätzten und tatsächlichen Laufzeit geschehen. Dabei werden drei verschiedene Simulink Modelle hinsichtlich ihrer Laufzeit auf der *MicroAutoBox* untersucht. Dies geschieht mit dem selben Aufbau und der selben Vorgehensweise, wie sie in Kapitel 5 erläutert und in Abbildung 23 dargestellt ist. Die drei verwendeten Versuchsmodelle sind in Tabelle 9 aufgeführt und beschrieben. Dabei ist das LABMDAGAS-Modell ein nicht nach MAAB-Regeln erstelltes Modell.

Tabelle 9: Versuchsmodelle

Name	Blockanzahl	Beschreibung
TMPMDL	160	Ein Temperaturmodell des Diesel-Abgasstrangs eines Fahrzeugs. Dabei wird der Abgasstrang in Abschnitte (Bricks) aufgeteilt, wobei für jeden Teilabschnitt eine Temperatur berechnet wird. In je mehr “Bricks” der Abgasstrang unterteilt wird, desto aufwändiger ist die Berechnung.
DPFMON	366	Ein Dieselpartikelfilter-Monitor. Dieser überwacht den Zustand den Dieselpartikelfilters durch messen des Differenzdrucks vor und nach dem Filter. Die Differenzdruckmessungen werden durch eine Parameterschätzung mit Hilfe des Kalman-Filters angepasst.
LAMBDA GAS	756	Eine Lambda-Regelung für Gasmotoren. Abhängig vom Sauerstoffanteil im Abgas regelt dieses Modell die Verbrennung und ermöglicht die Abgasreinigung durch einen Drei-Wege-Katalysator.

Durch die Codegenerierung wird aus den Modellen C-Code erzeugt. Anschließend wird der Code für die MicroAutoBox kompiliert und über ControlDesk auf das Steuergerät geladen. Die Codegenerierung und die Kompilierung werden mit den selben Einstellungen, wie sie auch bei Vermessung der Funktionsblöcke verwendet werden, durchgeführt. Funktionsblöcke, die das Vorhandensein von technischen Systemen modellieren und somit nur für die Simulation notwendig sind, werden zuvor aus den Modellen entfernt

und mit entsprechenden “Dummy-Werten” ersetzt. Dies gilt insbesondere für den “From Workspace”-Funktionsblock, der in jedem Zeitschritt Werte zur Verfügung stellt, die zuvor am Prüfstand gemessen wurden. Verbleibt dieser Block im Modell hat dies signifikante Auswirkungen auf die Laufzeit des Programmes.

Jedes Programm wird auf dem Steuergerät mehrfach ausgeführt. Die Laufzeiten werden dabei als Messreihe in ControlDesk aufgenommen. Anhand dieser Daten wird die Güte der Laufzeitschätzung ermittelt. Im Folgenden sind die ermittelten Testergebnisse auszuwerten. Darüber hinaus ist neben der Betrachtung der Laufzeit zu überprüfen, inwieweit das Konzept und die Implementierung dem in der Analyse erstellten Anforderungskatalog entsprechen.

6.2 Testdurchführung

Als erstes Modell wird das TMPMDL-Modell untersucht. Die Laufzeitverteilung für dieses Modell ist im Diagramm der Abbildung 28 dargestellt. Für diese Messreihe ist das Modell mit sechs “Bricks” eingestellt. Dies bedeutet, dass der modellierte Abgasstrang in sechs Abschnitte unterteilt ist, für die jeweils die Temperatur berechnet wird.

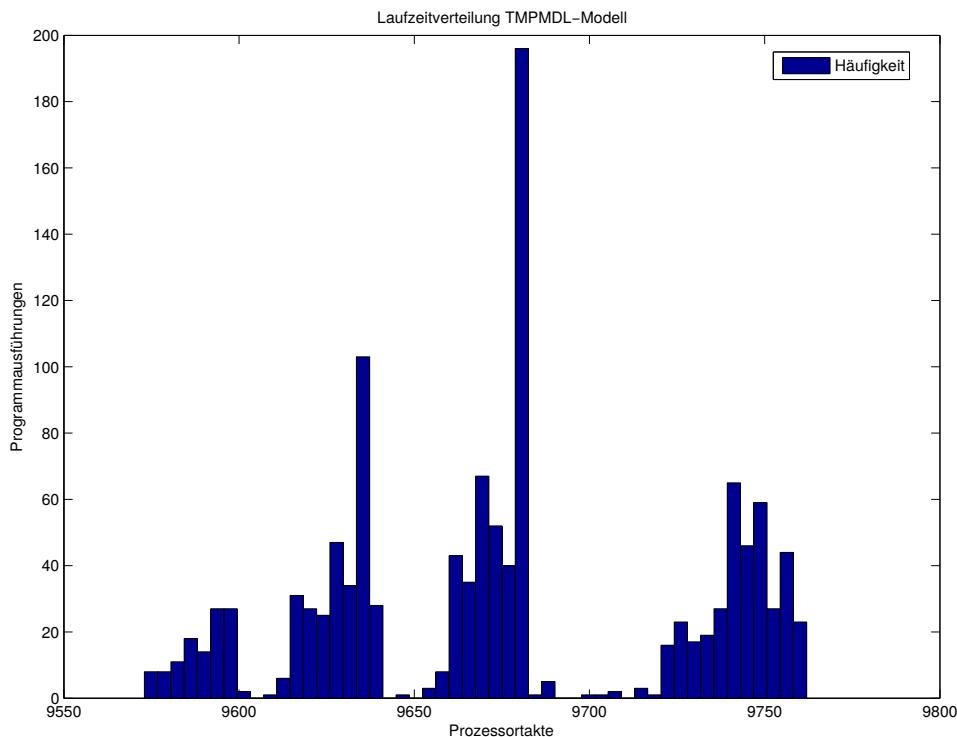


Abbildung 28: Verteilung der Laufzeiten für das TMPMDL-Modell

Auffällig bei dieser Verteilung ist, dass die Laufzeiten sich auf vier Bereiche aufteilen. Die Laufzeit steigt während der Durchläufe immer weiter an. Nachdem die Ausführungszeit im Bereich von 9700 Prozessortakten angelangt ist, steigt sie nicht weiter an. Eine Erklärung für dieses Laufzeitverhalten kann nicht angegeben werden.

Um die Schätzung der Laufzeit für dieses Modell zu berechnen, wird der “Runtime Estimation”-Funktionsblock der Toolbox in der obersten Ebene des TMPMDL-Modells eingesetzt. Nach der Platzierung des Blocks wird die Simulation gestartet. Die Informationen zu den einzelnen Funktionsblöcken können in der Matlab-Konsole eingesehen werden. Die Laufzeitschätzung für das TMPMDL-Modell ergibt 10818,45 Prozessortakte mit einer Standardabweichung von 600,06 Takten. Aus der Verteilung in Abbildung 28 ergibt sich eine maximale Laufzeit von 9761 Prozessortakten. Zu dieser Ausführungsduer ist die Zeit, die zur Ausführung der Update-Funktion des TMPMDL-Modells benötigt wird, zu addieren. Die Laufzeitverteilung ist in Abbildung 43 (Anhang) dargestellt. Als maximale Laufzeit ergibt sich hier ein Wert von 55 Takten. Das Ergebnis ist in Abbildung 29 aufgetragen (vgl. Abbildung 48 (Anhang)). Es zeigt sich, dass die Laufzeitschätzung die tatsächliche Ausführungsduer mit 1002,45 Takten überschätzt. Dies entspricht einer Überschätzung von 10,2%. Dabei liegt die Messung in der $1,67\sigma$ -Umgebung der Schätzung, mit $\sigma = 600,06$. Verantwortlich für eine solche Überschätzung sind mehrstufige Optimierungsprozesse, die nicht in die Laufzeitanalyse mit einfließen. Dazu zählen Optimierungen während der Codegenerierung, die jedoch zu einem Großteil durch die Laufzeitmessung der Funktionsblöcke abgebildet werden kann. Weitere Optimierungen des Programmcodes entstehen während der Kompilierung des C-Codes, sowie aktiv während der Ausführung auf der Hardware. Da eine Laufzeitanalyse hinsichtlich der zu erwartenden Ressourcenausnutzung erstellt wird, sind Optimierungen weniger kritisch, da sie das Einhalten von Echtzeitanforderungen nicht beeinflussen.

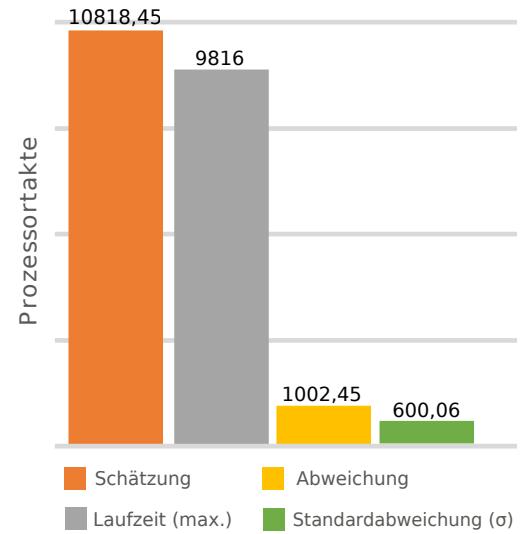


Abbildung 29: Laufzeitwerte des TMPMDL-Modells

Da die Laufzeit des TMPMDL-Modells durch eine höhere Anzahl an “Bricks” ansteigt, kann über diesen Parameter festgestellt werden, inwieweit die Laufzeitschätzung ebenfalls skaliert. Für eine spätere Anwendung der Modelfunktionalität kann hierbei die Frage beantwortet werden, wie viele “Temperatur-Bricks” tatsächlich innerhalb der vorgegebenen maximalen Ausführungsduer bzw. Abtastzeit berechnet werden können. Die Abtastzeit, für die in dieser Arbeit betrachteten Modelle, liegt bei 0,01 Sekunden. In Abbildung 47 (Anhang) sind die Laufzeiten des Modells in Abhängigkeit der Anzahl an “Bricks” angegeben. Die entsprechende Laufzeitschätzung ist ebenfalls aufgetragen. Zu sehen ist, dass die Schätzung der Ausführungsduer stets höher als die tatsächlich gemessene Laufzeit ausfällt. Ebenso skaliert die absolute Abweichung, wobei die Abweichung

ab etwa 20 “Bricks” etwas ansteigt. Bei 100 Abschnitten beträgt die prozentuale Abweichung 13,6%. Diese Steigerung ist durch die optimierte Abarbeitung des Programms auf der Hardware zu erklären. Es zeigt dich ferner, dass eine Einteilung in 100 Abschnitte in der Schätzung $212,6\mu s$ benötigt und eine tatsächliche Laufzeit von $187,1\mu s$ aufweist. Damit liegen beide Werte deutlich unter der Abtastzeit von 0,01 Sekunden. Um die Berechnung zu verfeinern, lässt sich das Modell somit in weitere Abschnitte einteilen.

Als zweites Modell wird das DPFMON-Modell untersucht. Das Diagramm der Laufzeitmessung ist für dieses Programm in Abbildung 30 dargestellt. Auffällig ist bei dieser Messung, dass sich die Laufzeiten auf zwei deutlich unterschiedliche Bereiche aufteilen. Dies lässt sich durch den Einsatz des Kalman-Filters erklären. Das Modell ist so aufgebaut, dass nicht in jedem Programmdurchlauf eine Berechnung des Filters angestoßen

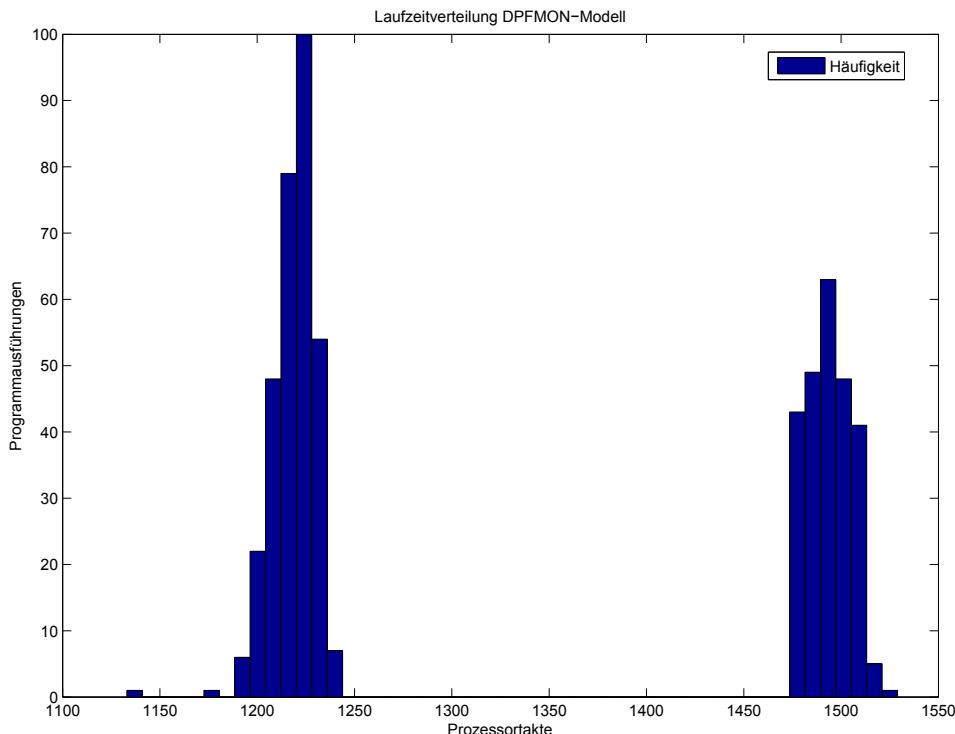


Abbildung 30: Verteilung der Laufzeiten für das DPFMON-Modell

wird. Nur in jedem zehnten Durchlauf wird das Kalman-Filter ausgeführt. Bei einer Abtastzeit von 0,01 Sekunden wird das Kalman-Filter dementsprechend jede 100ms aufgerufen. Implementiert wird dies in Simulink durch ein “Triggered Subsystem”. Da die Funktionalität des Kalman-Filters als Subsystem gekapselt ist, lässt sich die Laufzeitanalyse auf dieses System anwenden und so die Implementierung des Filters genauer untersuchen. Die Laufzeitanalyse für das Kalman-Filter-Subsystem ergibt 289,00 Prozessortakte mit einer Standardabweichung von 20,00 Takten. Diese Schätzung entspricht mit sehr guter Näherung dem Abstand der zwei Laufzeitbereiche der Messung. Für das gesamte DPFMON-Modell ergibt die Analyse einen Wert von 1762,93 Prozessortakten und eine Standardabweichung von 253,96 Takten. Die Laufzeitverteilung der Update-Funktion

ist in Abbildung 44 (Anhang) für dieses Modell dargestellt. Die maximale Laufzeit der Update-Funktion ergibt sich zu 54 Prozessortakten. Die Gesamtlaufzeit beträgt für dieses Modell 1583 Takte (siehe Abbildung 31). Im Unterschied zum TMPMDL-Modell werden im DPFMON-Modell keine Schleifen eingesetzt. Somit akkumulieren sich die Fehler der Laufzeitmessung bzw. Laufzeitschätzung weniger stark. Dies ist ebenfalls an der Standardabweichung der beiden Laufzeitanalysen ablesbar. Trotz der deutlich geringeren Anzahl an Funktionsblöcken ergibt die Schätzung für das TMPMDL-Modell eine höhere Standardabweichung. Dies ist damit zu begründen, dass sich ein Großteil der Funktionsblöcke des TMPMDL-Modells in einer Schleife befinden. Diese For-Schleife berechnet für jeden “Brick” die entsprechende Temperatur. Aufgrund dieser Schleife werden insgesamt mehr Berechnungen als bei der Abarbeitung des DPFMON-Programms ausgeführt. Ein weiterer Unterschied ist, dass im Gegesatz zum TMPFMDL-Modell das DPFMON-Modell Verzweigungen in Form von “if/else”-Bedingungen enthält. Insgesamt lässt sich die Laufzeitanalyse für dieses Modell in die gleiche Genauigkeitsklasse wie die Schätzung für das TMPMDL-Modell einordnen. Mit einer Schätzung von 1762,93 Prozes-

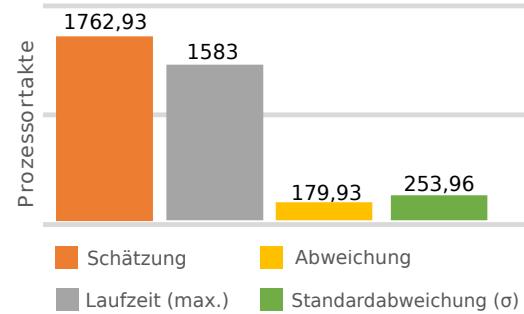


Abbildung 31: Laufzeitwerte des DPFMON-Modells

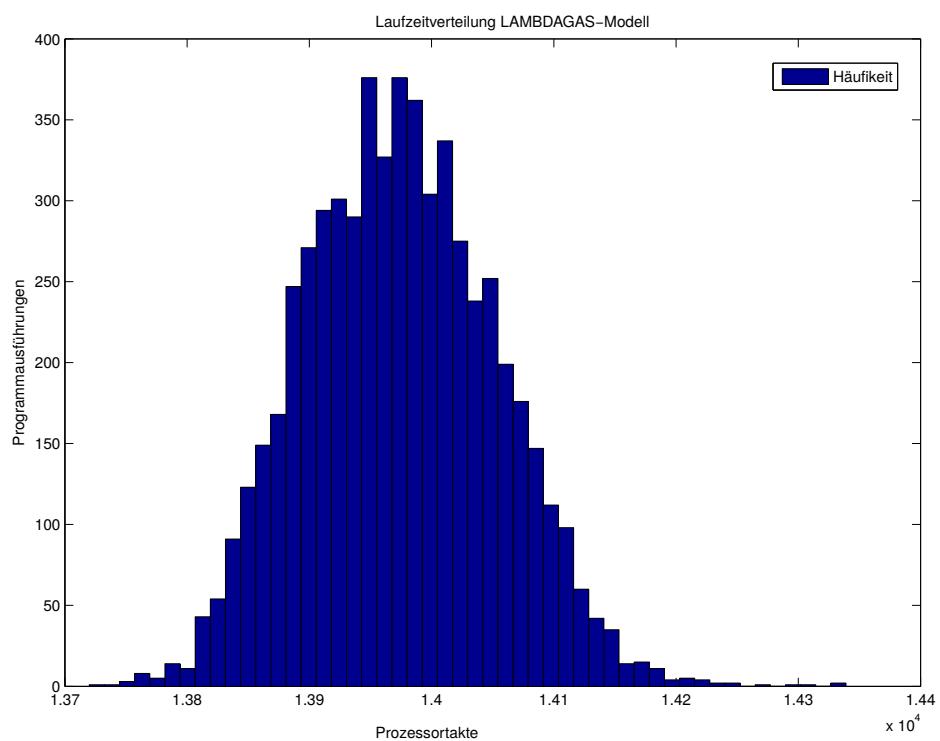


Abbildung 32: Verteilung der Laufzeiten für das LAMBDA GAS-Modell

sortakten wird die tatsächliche Laufzeit mit 11,4% überschätzt. Dabei liegt die gemessene Ausführungsduer in der $0,71\sigma$ -Umgebung der Schätzung.

Das dritte Modell, an dem die Implementierung evaluiert wird, ist das LAMBDA GAS-Modell. Hierbei handelt es sich um ein nicht MAAB-konformes Modell bzw. um ein Modell in der Konzeptphase. Die Laufzeitmessungen für dieses Modell sind in Abbildung 32 angegeben. Die Laufzeiten für dieses Programm sind normalverteilt. Es gibt keine Sprünge oder gesonderten Bereiche. Die maximale Laufzeit ergibt sich für dieses Modell zu 14339 Prozessortakten. Die Laufzeitverteilung für die Update-Funktion des LAMBDA GAS-Modells ist in Abbildung 45 (Anhang) dargestellt. Die Update-Laufzeit für dieses Modell sind maximal 607 Takte. Insgesamt summiert sich die maximale Laufzeit somit zu 14946 Prozessortakten. Die Laufzeitanalyse für das LAMBDA GAS-Modell ergibt: 13721,42 Prozessortakte und eine Standardabweichung von 544,16 Takten. Für dieses Programm wird die Ausführungsduer durch die Laufzeitanalyse mit 8,2% unterschätzt.

Dies zeigt deutlich, dass die Laufzeitanalyse, wie sie in dieser Arbeit konzipiert ist, keine WCET-Analyse darstellt. Sie stellt ein Werkzeug zur ersten Abschätzung der Laufzeit in einer frühen Phase der Funktionsentwicklung und auf sehr abstrakter Ebene dar. Ohne das Wissen über die konkrete Codegenerierung lassen sich hardwarenahe Prozesse nur schwer approximieren. Dies ist ebenfalls der Grund für eine solche Unterschätzung der Laufzeit.

Grund für diese Unterschätzung ist der Zugriff der CPU auf den Level 2 (L2) Cache. Bei der Ausführung des LAMBDA GAS Modells geschieht es, dass benötigte Daten nicht im Level 1 (L1) Cache zur Verfügung stehen. Anders als bei den Modellen zuvor passt das Programm des Modells nicht komplett in den L1 Cache. Dies führt zu sogenannten "Cache Misses" auf der L1 Ebene und damit zu längeren Ausführungszeiten. Mithilfe des "Profiling Monitors" des PowerPC 750GL lassen sich die "Cache Misses" des L1 Caches zählen. Dazu sind die Bits 0 bis 4 des MMCR1 Registers (SPR 956) wie folgt zu besetzen.

MMCR1					
0	0	1	0	1	0..0
0	1	2	3	4	5..31

Die Anzahl der "Cache Misses" wird durch diese Einstellung im Zählregister PMC3 (SPR 957) gespeichert (vgl. Kapitel 5.2.1). Da die Programme des TMPMDL- und DPFLMON-

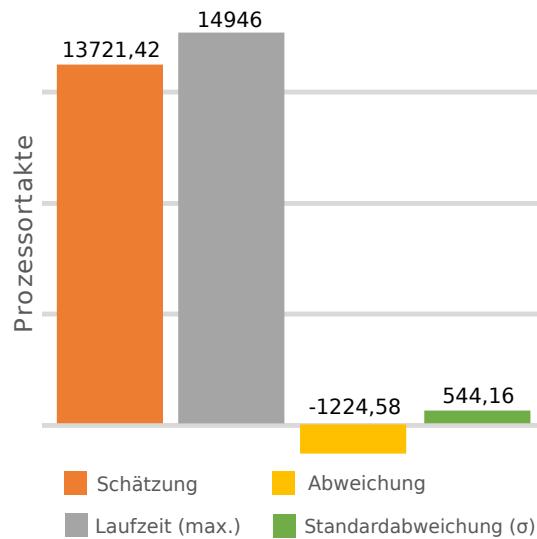


Abbildung 33: Laufzeitwerte des LAMBDA GAS-Modells

Modells vollständig in den L1 Cache passen, gibt es hier keine Zugriffe auf den L2 Cache. Die Messung der “L1 Cache Misses” für das LAMBDA GAS-Modell ist im Diagramm der Abbildung 46 (Anhang) dargestellt. Die maximale Anzahl an “L1 Cache Misses” und damit die maximale Anzahl an Zugriffen auf den L2 Cache ist 264. Die Messung für “L2 Cache Misses” ergibt, dass während der Ausführung nicht auf den Hauptspeicher zugegriffen werden muss. Das Datenblatt für den PowerPc 750GL Prozessor [IBM06] gibt für einen “L1 Cache Miss” mit anschließendem Treffer im L2 Cache eine “Estimated Latency” von 13 Prozessortakten an. Die Laufzeit für das LAMBDA GAS-Modell ergäbe sich bei einer Ausführung des Programmes ohne “Cache Misses” somit näherungsweise zu $14946 - 264 \cdot 13 = 11514$ Prozessortakten. Für die Laufzeitanalyse ergibt sich somit eine Überschätzung von 19,2% (siehe Anhang Abbildung 49). Die häufigen Speicherzugriffe des LAMBDA GAS-Programms lassen sich neben der Größe des Modells ebenfalls mit der Nutzung von vergleichsweise vielen bzw. größeren Kennfeldern zurückführen (im Vergleich zu TMPMDL und DPFMON). Diese werden durch “Lookup_n-D”-Funktionsblöcke modelliert. Die Werte der Kennfelder müssen dabei bei jeder Berechnung geladen und im Cache vorhanden sein.

Als abschließende Betrachtung sind im Folgenden die am häufigsten verwendeten Funktionsblöcke in der gleichen Reihenfolge wie in Abbildung 13 aufgetragen und mit den, durch die Analyse ermittelten, durchschnittlichen Laufzeiten verbunden. Dabei ist

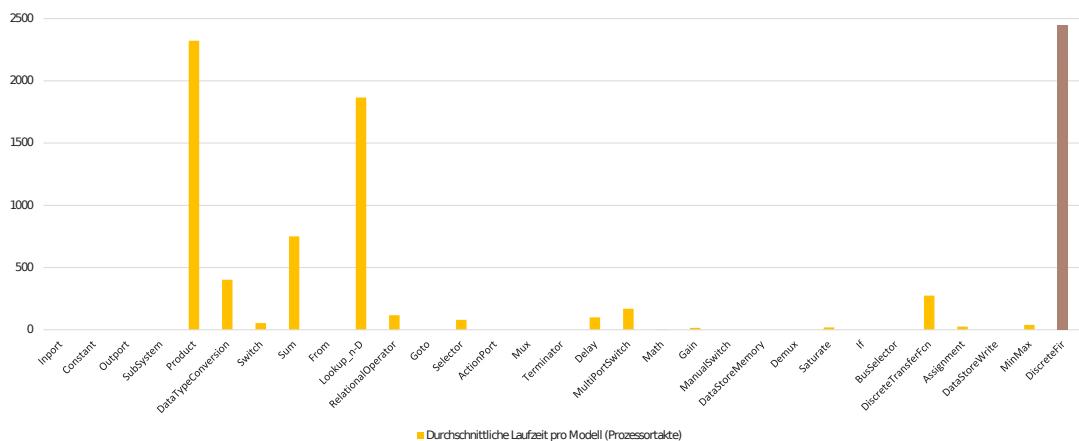


Abbildung 34: Durchschnittliche Blocklaufzeiten pro Simulink Modell

in Abbildung 34 zu erkennen, dass die am häufigsten verwendeten vier Funktionsblöcke keinen Einfluss auf die Laufzeitschätzung haben. Es zeigt sich, dass die Grundrechenarten sowie die Lookup Tables durchschnittlich die meiste Laufzeit ausmachen. Für die “Product” und “Sum”-Blöcke ist neben der häufigen Verwendung ebenfalls die Kombination verschiedener Datentypen der Eingangssignale für eine höhere Laufzeit verantwortlich. Das “Casten” von Datentypen ist demnach ebenso eine Ursache für eine längere Ausführungsduer. Aufgrund des häufigen Einsatzes von Kennfelder in Simulationen bei IAV, sind ebenfalls die “Lookup Table”-Funktionsblöcke in den meisten Modellen zu finden.

Ganz besonders sticht in Abbildung 34 der “DiscreteFir”-Funktionsblock hervor. Zum einen, weil er nicht zu den am häufigsten verwendeten, MAAB-konformen Standardblöcken gehört und zum anderen, weil er im Durchschnitt trotzdem eine sehr hohe Ausführungszeit besitzt. Eingesetzt wird dieser Funktionsblock im LAMBDA GAS-Modell, um dort einen gleitenden Mittelwert der letzten 1000 Werte zu berechnen. Die Laufzeitanalyse ermöglicht es, diese Designentscheidung bereits in der Konzeptphase ausfindig zu machen und gibt so die Möglichkeit bereits früh in der Entwicklung performantere Lösungen zu finden, falls dies erforderlich ist.

6.3 Fazit

Durch die Untersuchung der drei Modelle kann gezeigt werden, dass die Umsetzung des Konzepts die Anforderungen an eine frühe, auf Modellebene ausgeführte Laufzeitanalyse erfüllt. Es ist möglich, eine Laufzeitabschätzung auf Modellebene anhand einer zuvor erstellten Lookup Table sowie gegebenen Modellinformationen zu erstellen, dabei ist eine durchschnittliche Abweichung von 9,9% im Toleranzbereich für eine erste abschätzende Laufzeitanalyse. Es kann folglich gezeigt werden, dass für eine erste Abschätzung keine aufwändigen Prozesse vonnöten sind. Damit bieten die in dieser Arbeit entworfenen Prozesse eine ideale Ergänzung zu den bereits in der Industrie eingesetzten detaillierten Laufzeitanalysewerkzeugen.

Das Konzept erfüllt die Anforderung an eine Prozesskette, die es erlaubt ohne steile Codegenerierung und Kompilierung eine Laufzeitschätzung zu erstellen. Nach dem Aufbau der Lookup Table ist keine Hardware, Codegenerierung oder Kompilierung des Simulink Modells mehr vonnöten, um die Laufzeitanalyse auszuführen. Durch die Lookup Table ist die Analyse ebenfalls für andere Systeme konfigurierbar. Abhängig davon, für welche Zielplattform eine Lookup Table aufgebaut ist, ergibt die Laufzeitanalyse eine auf die Plattform zugeschnittene Laufzeitschätzung. Da der Speicherzugriff nur auf L1-Cache-Ebene während der Laufzeitmessung der Simulink Blöcke mit abgebildet wird, kann es bei großen Modellen und häufigen Speicherzugriffen (insbesondere auf die unteren Speicherebenen) zur Unterschätzung der Laufzeit kommen. Durch die Möglichkeit, die Laufzeitmessung für Funktionsblöcke zu automatisieren und die Lookup Table Einträge durch individuelle Laufzeitfunktionen zu ergänzen, kann die Prozesskette kosten- und zeiteffektiv umgesetzt werden. Darüber hinaus ist die gesamte Toolbox in Matlab-Code umgesetzt und damit eine vollständige Einbettung in die Matlab/Simulink Umgebung gegeben. Um die Laufzeitanalyse nutzen zu können, sind keine Kenntnisse über die Zielhardware, Laufzeitanalysekenntnisse oder tiefergehende Programmierkenntnisse vonnöten. Die Toolbox ist über “Set Path” der Matlab-Umgebung hinzuzufügen. Danach kann bereits der “Execution Estimation”-Funktionsblock im Modell eingesetzt und eine Laufzeitabschätzung durchgeführt werden.

Als prototypische Umsetzung konnten die gestellten Anforderungen erfüllt werden. Damit eine Laufzeitanalyse für beliebige Modelle und Modellumfänge realisierbar ist, sind weitere Simulink Funktionsblöcke in die Lookup Table aufzunehmen und insbesondere das “Caching” stärker in die Analyse mit einzubeziehen. Darüber hinaus macht das Wissen über die konkrete Codegenerierung und Kompilierung eine genauere Laufzeitschätzung möglich.

7 Zusammenfassung und Ausblick

Unter Berücksichtigung der Aufgabenstellung sowie der gestellten Anforderungen soll in diesem Kapitel die vorliegende Arbeit abschließend zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben werden.

7.1 Zusammenfassung

Zielstellung dieser Arbeit ist die Konzeption sowie Spezifikation eines Laufzeitanalysewerkzeugs. Dieses Werkzeug soll Aussagen zum Zeitverhalten von Simulink Modellen bzw. seiner Module ermöglichen. Dies erlaubt es, bereits in der frühen Phase der Funktionsentwicklung, detailliert Auswirkungen bestimmter Designentscheidungen nachvollziehen zu können.

Lautzeitanalysen werden zumeist auf Ebene des Assembler- oder Binärprogramms ausgeführt, um möglichst sichere obere Zeitschranken zu ermitteln. Dabei sind die verwendeten mathematischen Methoden insbesondere für größere Programme sehr aufwändig. Das in dieser Arbeit vorgestellte Konzept definiert eine alternative Vorgehensweise, die es erlaubt bereits auf Modellebene erste Laufzeitabschätzungen vorzunehmen. Dabei lässt sich die entwickelte Prozesskette in zwei Teile aufteilen: Zunächst wird durch die Laufzeitvermessung einzelner Simulink Funktionsblöcke auf der Zielhardware eine Lookup Table aufgebaut, aus der die Laufzeitinformationen in der späteren Analyse ausgelesen werden können. Der zweite Prozess beinhaltet die Modellanalyse. Neben den Laufzeitwerten aus der Lookup Table werden die für eine Abschätzung notwendigen Informationen in diesem Prozessschritt ausgelesen und verarbeitet. Je Funktionsblock und Parametrisierung kann so individuell die Ausführungsdauer abgeschätzt werden. Die Summe der Einzelschätzungen ergibt die Laufzeitschätzung für das gesamte Modell. Dabei lässt sich die Laufzeitanalyse über die Lookup Table für verschiedene Steuergeräte und Prozessorarchitekturen konfigurieren. Überdies ist die Toolbox ausschließlich in Matlab-Code entwickelt und lässt sich somit vollständig in den Software-Entwicklungsprozess unter Simulink einbetten.

Anhand verschiedener Simulink Modelle wird die Umsetzung des Konzepts evaluiert und getestet. Dabei kann für das Prototypensteuergerät MicroAutoBox gezeigt werden, dass die Abweichung der Laufzeitschätzung bei durchschnittlich 9,9% von der tatsächlichen Laufzeit liegt. Bei umfangreichen Modellen können die häufigen Speicherzugriffe durch die Schätzung jedoch nicht abgebildet werden, was zu einer Unterschätzung der Laufzeit führt. Aus diesem Grund ist das Konzept, um eine tiefergehende “Cache-Analyse” zu erweitern.

7.2 Ausblick

Die Evaluierung des Konzepts zeigt, dass L2-Cache- oder Hauptspeicherzugriffe durch das Konzept nicht abgedeckt werden und es so zu Fehleinschätzung der Programmlaufzeit bei größeren Programmen kommen kann. Dies bietet die Möglichkeit in weiterführenden Arbeiten zu untersuchen, inwieweit Speicherzugriffe auf abstrakter Modellebene abgebildet bzw. vorhergesagt werden können. Erweiterbar ist die Laufzeitanalyse ebenfalls durch das Vermessen weiterer Simulink Funktionsblöcke und dem Wissen über die Regeln der Codegenerierung. Ist genau bekannt, wie ein Funktionsblock in C-Code übersetzt wird, kann mit diesen Informationen die Laufzeitanalyse präzisiert werden. Eine statische Analyse der Generierungsregeln könnte dabei die Laufzeitmessung auf der Hardware ersetzen. Neben der Analyse der “If/else”- und “Switch Case”-Verzweigungen können ebenfalls die häufig verwendeten “Switch”-Blöcke in weiterführenden Arbeiten betrachtet werden. Dabei ist auf Modellebene zu entscheiden, welche Funktionsblöcke durch optimierende Einstellungen ausgeführt werden und welche nicht.

Diskutiert werden könnte ebenfalls eine Erweiterung der Lookup Table. Um eine größere Anzahl an die Laufzeit beeinflussenden Blockparametern aufzunehmen, ist eine alternative Form der Strukturierung denkbar. Genutzt werden könnte dabei ein auf XML aufbauendes Format. Ferner könnte über eine automatisierte Vermessung der Funktionsblöcke eine Tabelle in entsprechendem Format direkt erstellt werden. Um die Nutzerfreundlichkeit zu erhöhen ist zudem die Entwicklung einer graphischen Nutzeroberfläche sowie die Anzeige der Laufzeitanalyse-Ergebnisse an jedem Block mithilfe von Annotations möglich.

Literaturverzeichnis

- [ABRW14] ANGERMANN, Anne ; BEUSCHEL, Michael ; RAU, Martin ; WOHLFARTH, Ulrich: *Matlab - Simulink - Stateflow - Grundlagen, Toolboxen, Beispiele*. 8. Auflage. De Gruyter Oldenbourg Verlag, 2014. – ISBN 3-486-57719-0
- [BR06] BURGUIÈRE, Claire ; ROCHANGE, Christine: History-based Schemes and Implicit Path Enumeration. In: MUELLER, Frank (Hrsg.): *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)* Bd. 4. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006 (OpenAccess Series in Informatics (OASIcs)). – ISBN 978-3-939897-03-3
- [FHW⁺06] FERDINAND, Christian ; HECKMANN, Reinhold ; WOLFF, Hans-Jörg ; RENZ, Christian ; PARSHIN, Oleg ; WILHELM, Reinhard: Towards Model-Driven Development of Hard Real-Time Systems. In: BROY, Manfred (Hrsg.) ; KRÜGER, Ingolf H. (Hrsg.) ; MEISINGER, Michael (Hrsg.): *Model-Driven Development of Reliable Automotive Services* Bd. 4922. Berlin, Heidelberg : Springer, Oktober 2006 (Lecture Notes in Computer Science), S. 145–160
- [FL10] FALK, Heiko ; LOKUCIEJEWSKI, Paul: A Compiler Framework for the Reduction of Worst-case Execution Times. In: *Real-Time Syst.* 46 (2010), Oktober, Nr. 2, S. 251–300. – ISSN 0922-6443
- [FW97] FERDINAND, Christian ; WILHELM, Reinhard: Fast and Efficient Cache Behavior Prediction / Universität des Saarlandes. 1997. – Forschungsbericht
- [Gus16] GUSTAFSSON, Jan ; MÄLARDALEN UNIVERSITY, SWEDEN (Hrsg.): *SWEET Manual*. Västerås, Sweden: Mälardalen University, Sweden, 2016. [www.mrtc.mdh.se/projects/wcet/sweet/manual/SWEET_manual.pdf](http://mrtc.mdh.se/projects/wcet/sweet/manual/SWEET_manual.pdf). – Version 2016-06-27
- [HF06] HECKMANN, Reinhold ; FERDINAND, Christian: Worst-Case Execution Time Prediction by Static Program Analysis / AbsInt Angewandte Informatik GmbH. 2006. – Forschungsbericht. – www.absint.com/aiT_WCET.pdf
- [HSDZ⁺09] HÖHN, Holger ; SECHSER, Bernhard ; DUSSA-ZIEGER, Klaudia ; MESSNARZ, Richard ; HINDEL, Bernd: *Software Engineering nach Automotive SPICE*. dpunkt.verlag, 2009. – ISBN 978-3-89864-578-2
- [IBM06] IBM (Hrsg.): *IBM PowerPC 750GX and 750GL RISC Microprocessor (User's Manual)*. 1.2. IBM, März 2006
- [Kel15] KELTER, Timon: *WCET Analysis and Optimization for Multi-Core Real-Time Systems*, Technischen Universität Dortmund, Diss., März 2015

- [Kir00] KIRNER, Raimund: *Integration of Static Runtime Analysis and Program Compilation*, Technischen Universität Wien, Diplomarbeit, 2000
- [Kir02] KIRNER, Raimund: The Programming Language wcetC / Technische Universität Wien, Institut für Technische Informatik. Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002 (2/2002). – Research Report
- [KLFP02] KIRNER, Raimund ; LANG, Roland ; FREIBERGER, Gerald ; PUSCHNER, Peter: Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In: *Proceedings of the 14th Euromicro Conference on Real-Time Systems*. Washington, DC, USA : IEEE Computer Society, 2002 (ECRTS '02). – ISBN 0-7695-1665-3
- [Lis14] LISPER, Björn: SWEET - A Tool for WCET Flow Analysis (Extended Abstract). In: *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, 2014, 482–485
- [LLMR07] LI, Xianfeng ; LIANG, Yun ; MITRA, Tulika ; ROYCHOUDURY, Abhik: Chronos: A Timing Analyzer for Embedded Software. In: *Science of Computer Programming* 69 (2007), Nr. 1-3, S. 56–67. – www.comp.nus.edu.sg/~rpembed/chronos
- [LM10] LOKUCIEJEWSKI, Paul ; MARWEDEL, Peter: *Worst-case execution time aware compilation techniques for real-time systems*. Dordrecht, Heidelberg, New York : Springer-Verlag, 2010 (Embedded systems). – ISBN 978-90-481-9928-0
- [Lun02] LUNDQVIST, Thomas: *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. Göteborg, Sweden, Department of Computer Engineering - Chalmers University of Technology, Diss., 2002
- [maa12] MathWorks Automotive Advisory Board (MAAB). Website (pdf). www.mathworks.com/solutions/automotive/standards/maab.html. Version: 3.0 (August 2012). – Control algorithm modeling guidelines using MATLAB®, Simulink®, and Stateflow®
- [Men12] MENGI, Cem: *Automotive Software : Prozesse, Modelle und Variabilität*. Aachen, RWTH Aachen, Diss., Juni 2012. – Aachener Informatik-Berichte, Software Engineering ; 13
- [MF06] McHOES, Ann ; FLYNN, Ida M.: *Understanding Operating Systems, Fourth Edition*. 4th. Boston, MA, United States : Course Technology Press, 2006. – ISBN 0-534-42366-3

- [MPPU10] MIN, Sang L. (Hrsg.) ; PETTIT, Robert (Hrsg.) ; PUSCHNER, Peter (Hrsg.) ; UNGERER, Theo (Hrsg.): *Software Technologies for Embedded and Ubiquitous Systems: 8th IFIP WG 10.2 International Workshop, SEUS 2010 Waidhofen/Ybbs, Austria, October 13-15, 2010: Proceedings.* Springer-Verlag, Oktober 2010 (Lecture Notes in Computer Science 6399). – ISBN 3–642–16255–X
- [MZK11] MALONE, Corey ; ZAHRAN, Mohamed ; KARRI, Ramesh: Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. In: *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing.* New York, NY, USA : ACM, 2011 (STC '11). – ISBN 978–1–4503–1001–7, S. 71–76
- [Rei08] REINEKE, Jan: *Caches in WCET Analysis - Predictability, Competitiveness, Sensitivity,* Universität des Saarlandes, Diss., November 2008. – ISBN 978-3-941071-69-8
- [San04] SANDELL, Daniel: *Evaluating Static Worst-Case Execution-Time Analysis for a Commercial Real-Time Operating System,* Mälardalen University, Masterarbeit, Juli 2004
- [SEG⁺06] SEHLBERG, Daniel ; ERMEDAHL, Andreas ; GUSTAFSSON, Jan ; LISPER, Björn ; WIEGRATZ, Steffen: Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems. In: *Leveraging Applications of Formal Methods, Second International Symposium, ISoLA 2006, Paphos, Cyprus, 15-19 November 2006,* 2006, 212–219
- [SKHX15] SANDER, Thomas ; KRÜGEL, Karsten ; HUFNAGEL, Thorsten ; XIE, Tao: Virtuelles Bypassing - Mit frühen Funktionstests zu besserer Steuergerätesoftware. In: *ATZ Elektronik* Jahrgang 10 (2015), Nov., Nr. 6, S. 38–43
- [SZ06] SCHÄUFFELE, Jörg ; ZURAWKA, Thomas: *Automotive Software Engineering.* Wiesbaden : Springer Vieweg, 2006 (ATZ/MTZ-Fachbuch). – ISBN 3–8348–0051–1
- [Tan06] TAN, Lili: The Worst Case Execution Time Tool Challenge 2006: Technical Report for the External Test. In: *In Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA '06),* 2006
- [Tei97] TEICH, Dr.-Ing. J.: *Digitale Hardware/Software-Systeme - Synthese und Optimierung.* 4. Auflage. Springer-Verlag, 1997. – ISBN 3–540–62433–3
- [Zöb08] ZÖBEL, Dieter: *Echtzeitsysteme: Grundlagen der Planung.* Berlin Heidelberg : Springer-Verlag, 2008 (examen.press). – ISBN 978–3–540–76396–3

A Anhang — Abbildungen

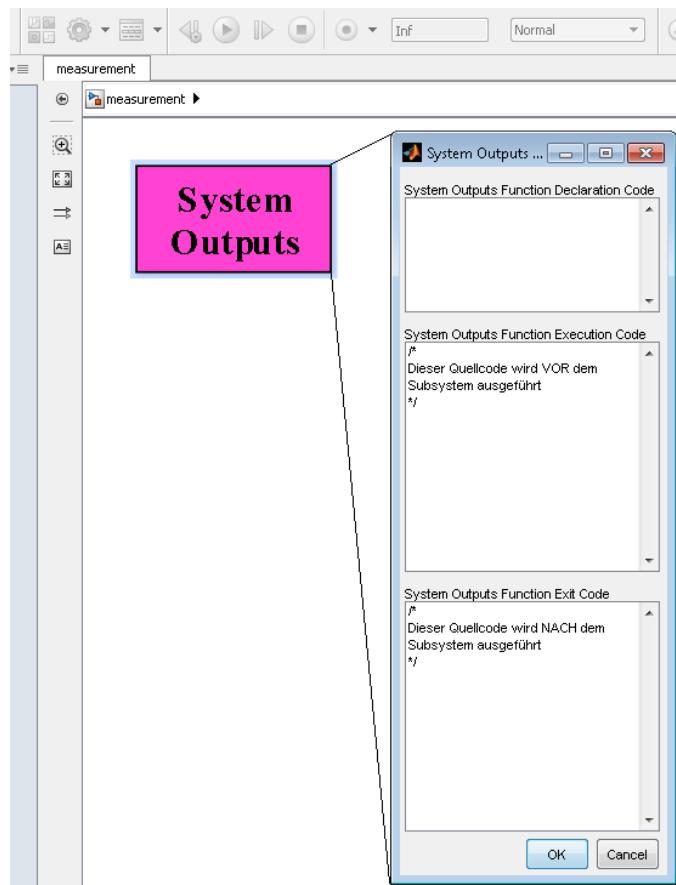


Abbildung 35: System Outputs Block mit GUI in Simulink

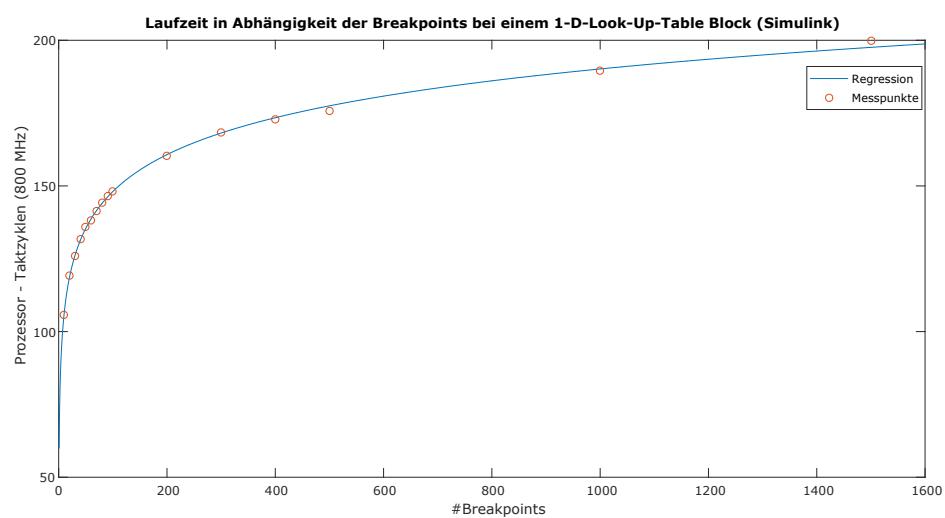


Abbildung 36: Ausführungszeit der 1-D Lookup Table in Abhängigkeit der Stützstellen

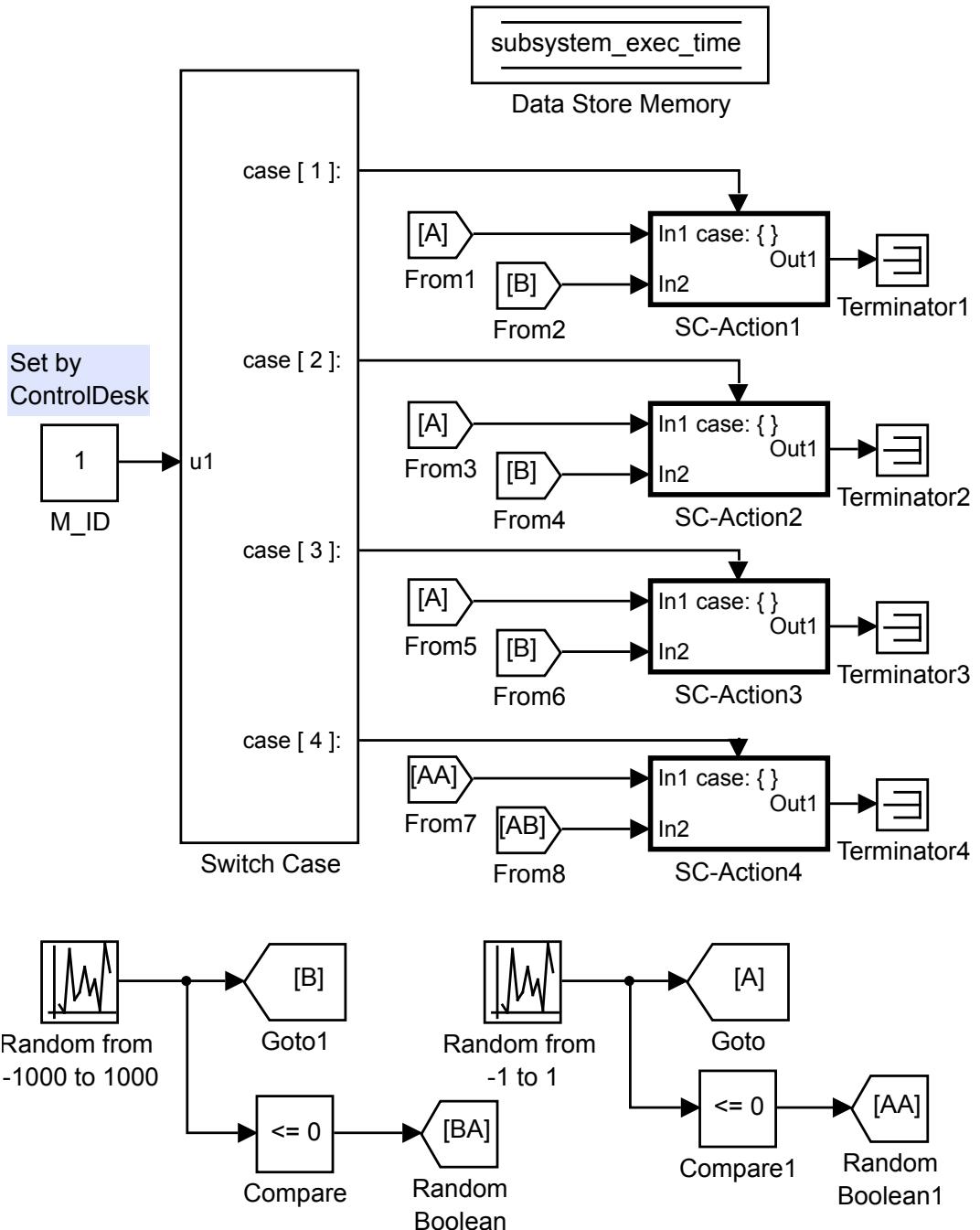


Abbildung 37: Simulink Modell zur Vermessung mehrerer Funktionsblöcke

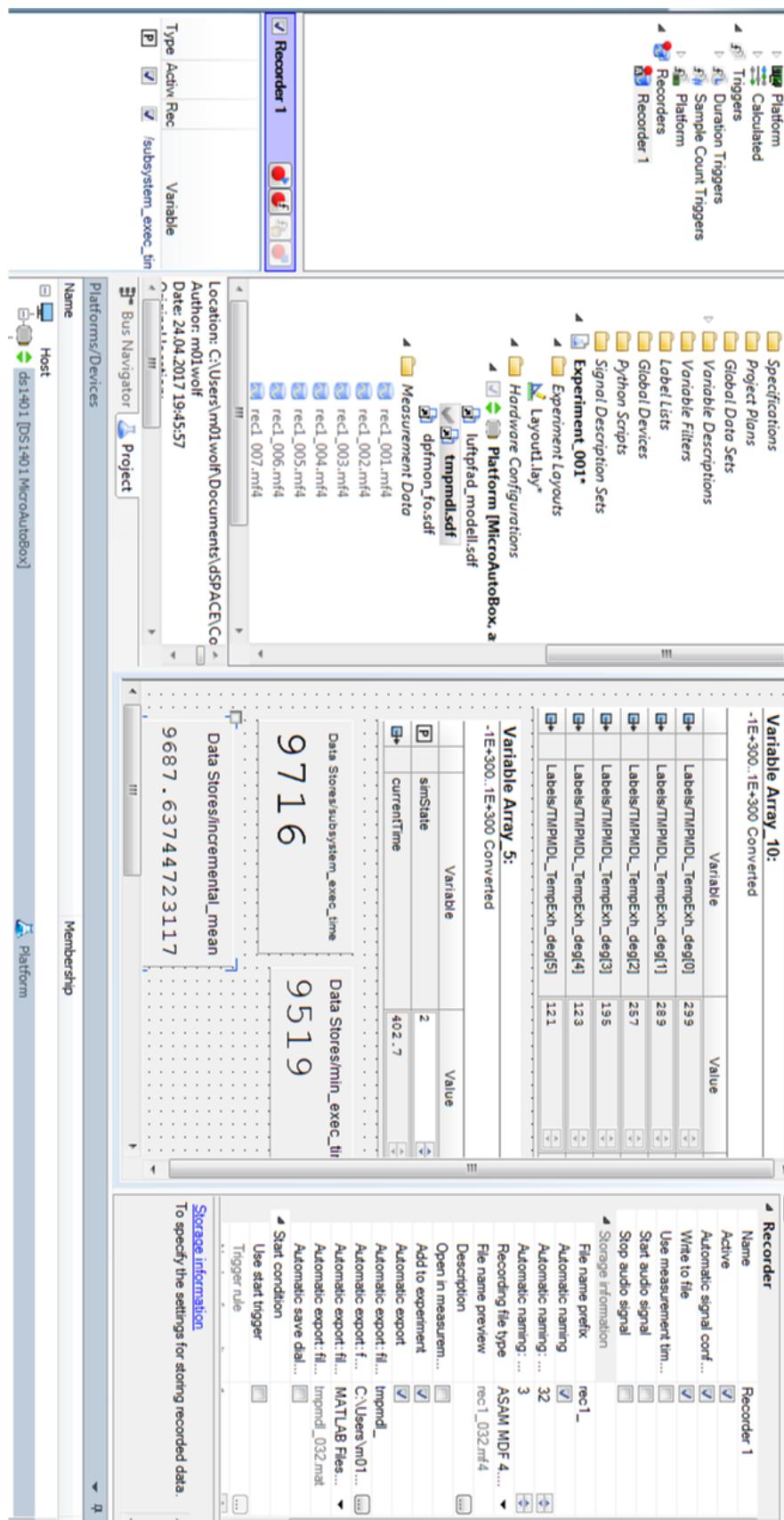


Abbildung 38: Aufbau von ControlDesk

	A	B	C
1	Block type	Processor cycles	Deviation
2	Product	7	0
3	Product (Division)	35,00	0,00
4	Sum	4,00	0,33
5	RelationalOperator	11,50	1,50
6	Saturation	11,84	2,50
7	Sqrt	74,00	1,00
8	Gain	8,00	1,50
9	DiscreteIntegrator	11,00	1,00
10	DiscreteFir	5,59	13,50

Abbildung 39: Lookup Table Ausschnitt (Laufzeiten für *double*)

	A	B	C
1	Conversion pattern	Processor cycles	Deviation
2	(single double)->(u?)int(8 16 32)	197,91	10,21
3	double->single	3,00	0
4	double->boolean	6,00	0
5	single->boolean	10,00	0
6	(u?)int(8 16 32)->double	13,00	0
7	(u?)int(8 16 32)->boolean	3,00	0
8	(u?)int(8 16 32)->single	16,00	0
9	(u?)int(8 16 32)->(u?)int(8 16 32)	1,00	0
10	boolean->(u?)int(8 16 32)	1,00	0
11	boolean->double	13,50	0,5
12	boolean->single	17,50	0,5

Abbildung 40: Lookup Table Ausschnitt (Laufzeiten für “Data Conversion”)

A	B	C	D
Block type / Mask type	Processor cycles	Deviation	Note
1 Block	9	0	
2 Switch	5,5	1	
3 Logic	2	0	Delay is the same as Unit Delay
4 Delay	0	0	
5 Terminator	0	0	
6 Selector	4	0	
7 For Loop	0,2419*loops+25,3655		
8 Assignment	rte.targetPPC750GL.runtime_assignment(block_handle)+2*loops		
9 Mux	0	0	
10 Demux	0	0	
11 Lookup Table	rte.targetPPC750GL.runtime_1DLookupTable(block_handle)	10,10	
12 Compare To Zero	11		For loop subsystem overhead. This value is calculated once, not for each loop.
13 Compare To Constant	11		Value for processor cycles is not multiplied when block is in iteration-subsystem. This value is calculated once, not for each loop.
14 MultiPortSwitch	9,5	14,96	
15 Merge	2	0	
16 Data Store Write	2,00	0,00	2 Subsystem (Mask); Same as 'Compare To Constant'
17 MathTranspose	rte.targetPPC750GL.runtime_math_transpose(block_handle)	3,15	2 Subsystem (Mask); Same as 'Compare To Zero'
18 Concatenate	rte.targetPPC750GL.runtime_matrix_concatenate(block_handle)	10,00	
19 MinMax	rte.targetPPC750GL.runtime_minmax(block_handle)	7,99	
20 DiscreteTransferFcn	rte.targetPPC750GL.runtime_discrete_fcn(block_handle)	0,00	
21 DigitalClock	6	0,00	
22 PulseGenerator	31	2,00	
23 Step	13	0	
24 Saturation Dynamic	30	2,00	Subsystem (Mask)

Abbildung 41: Lookup Table Ausschnitt mit eingebetteten Funktionen

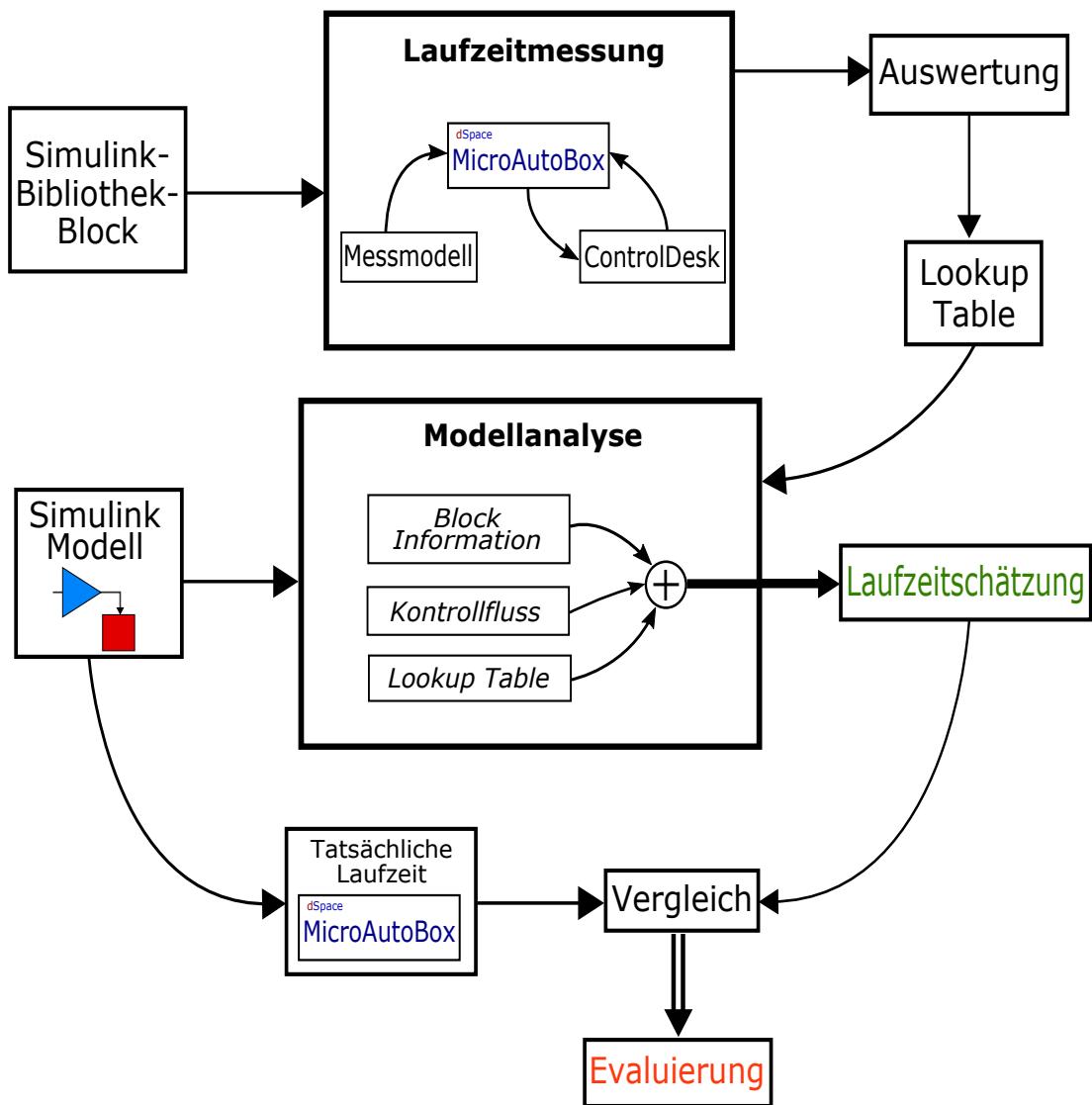


Abbildung 42: Gesamtschema der Konzeptumsetzung mit Evaluierung

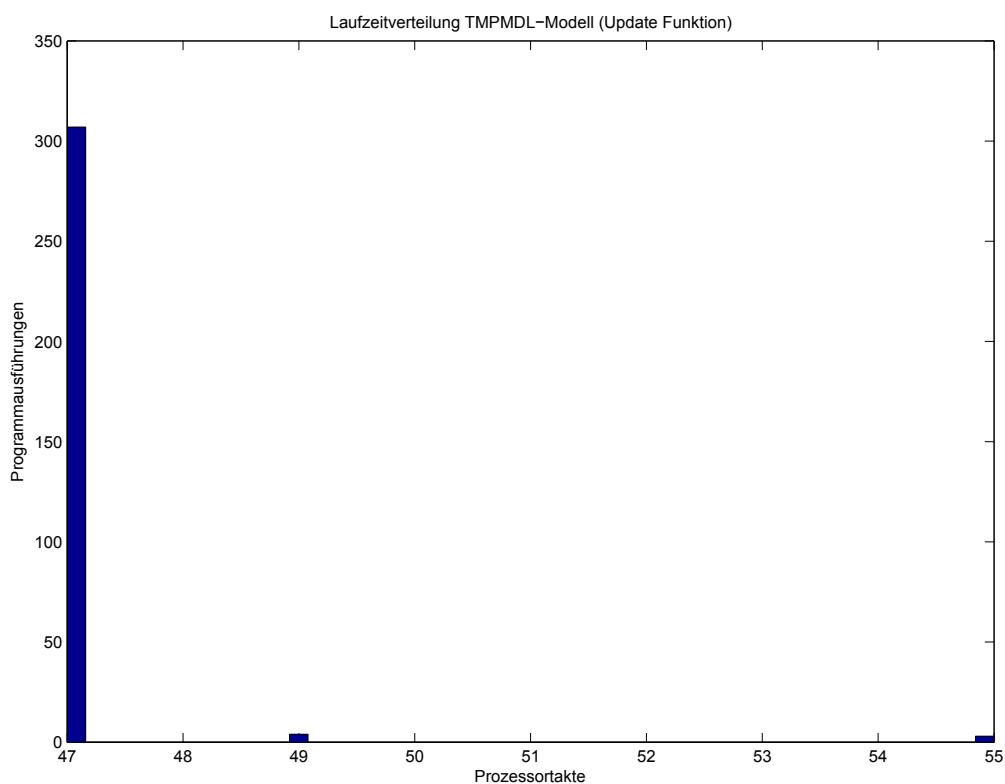


Abbildung 43: Laufzeit der Update-Funktion des TMPMDL-Modells

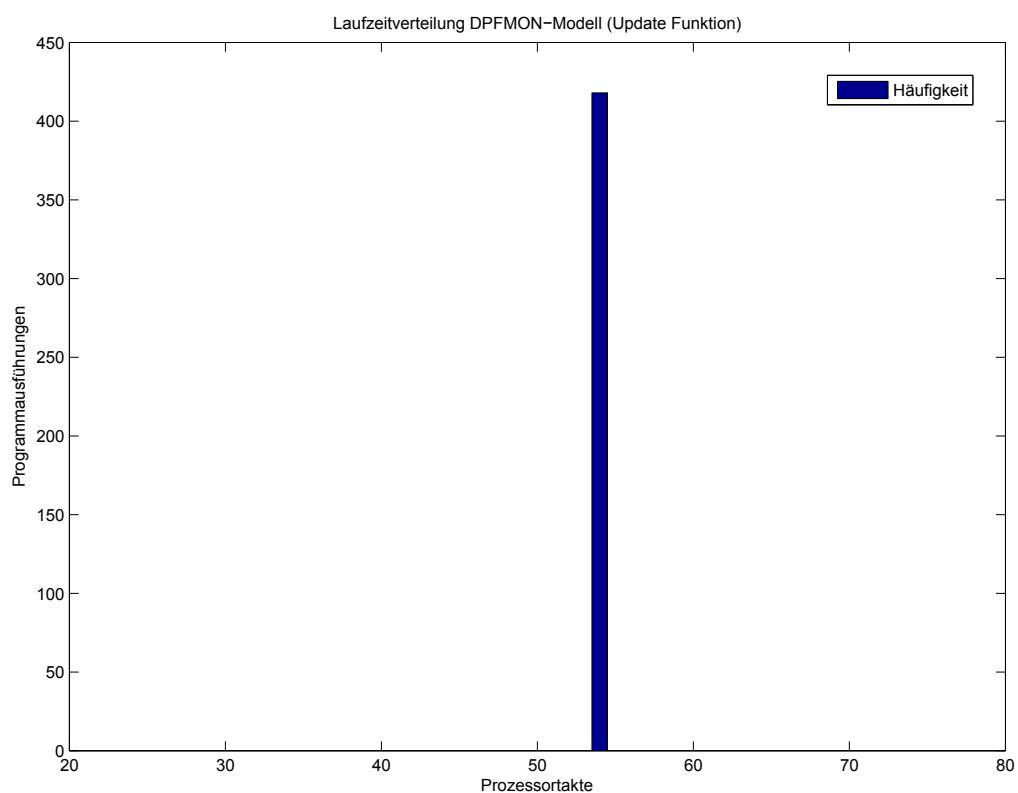


Abbildung 44: Laufzeit der Update-Funktion des DPFMON-Modells

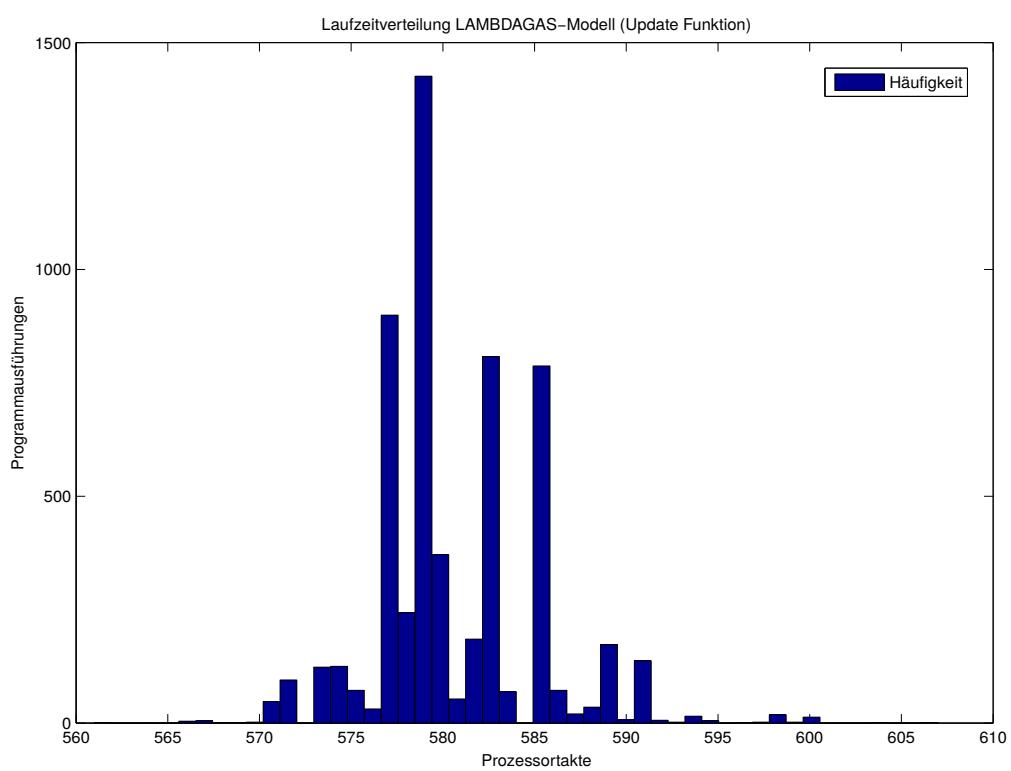


Abbildung 45: Laufzeit der Update-Funktion des LAMBDA GAS-Modells

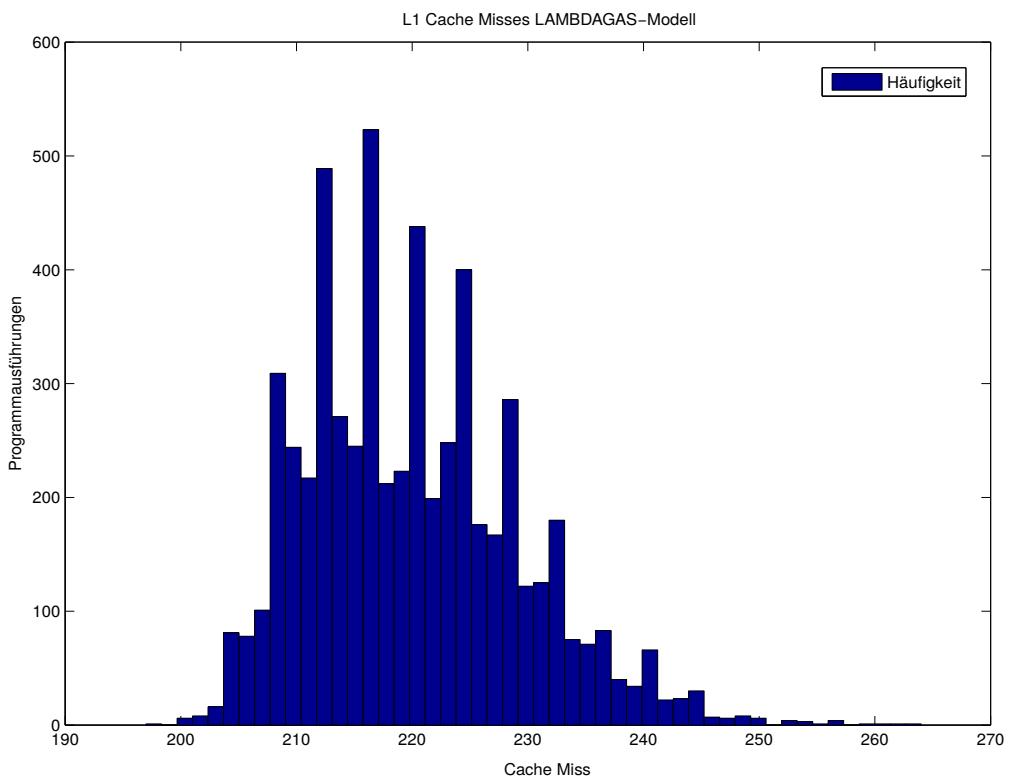


Abbildung 46: Anzahl der Cache-Misses des LAMBDA GAS-Modells

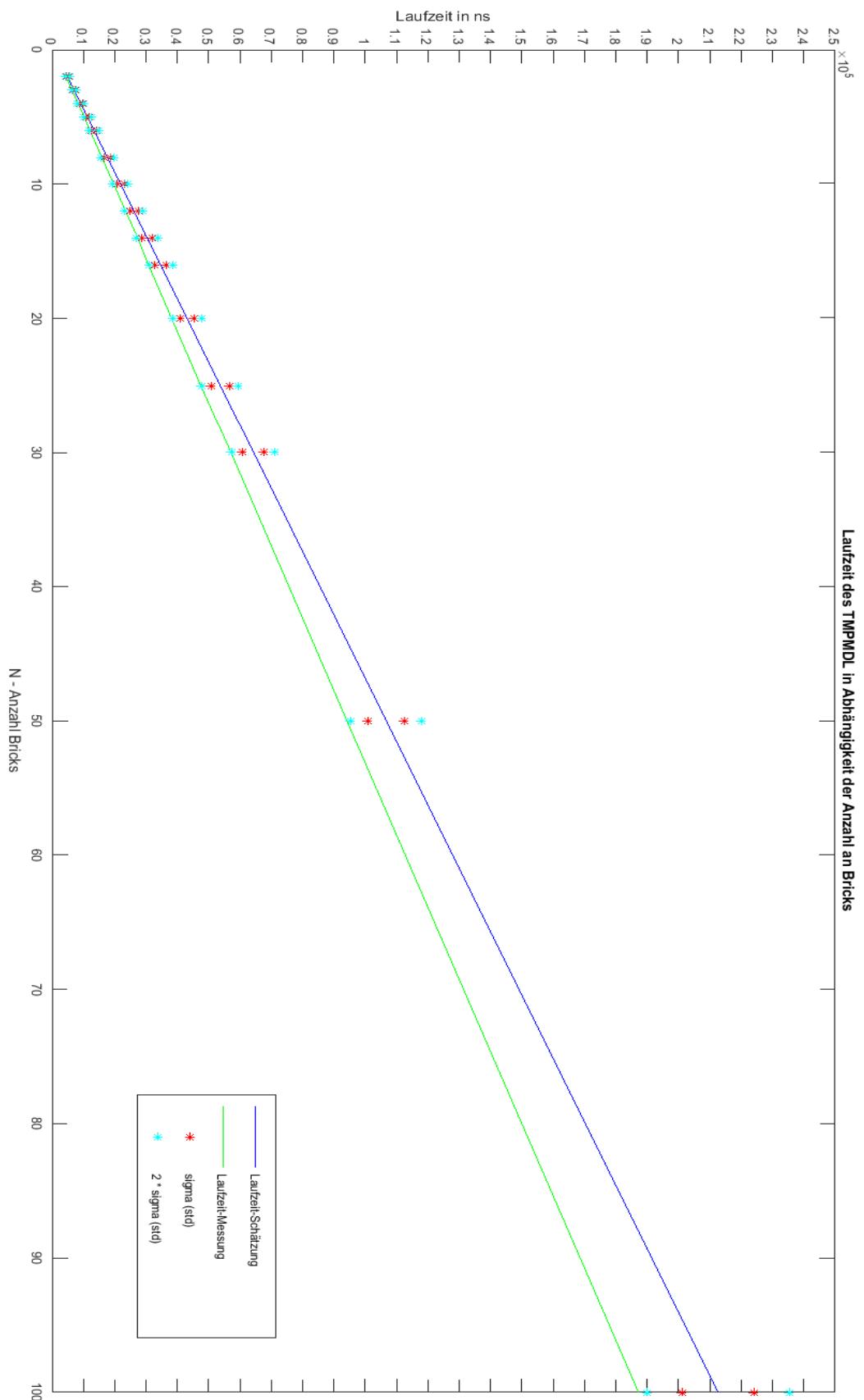


Abbildung 47: Laufzeit des TMPMDL Modells in Abhängigkeit der "Bricks"

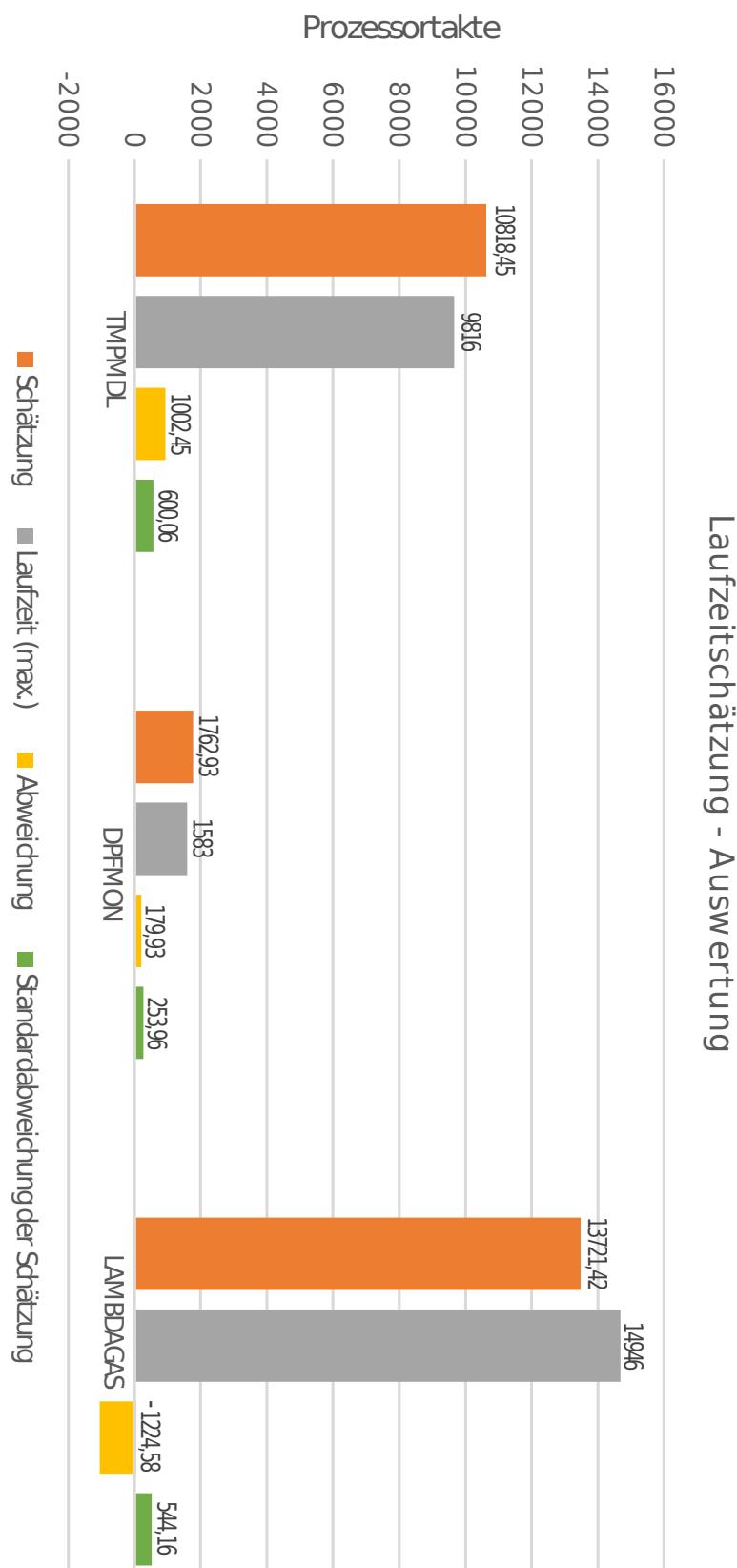


Abbildung 48: Laufzeiten der Simulink Modelle mit Laufzeitschätzung

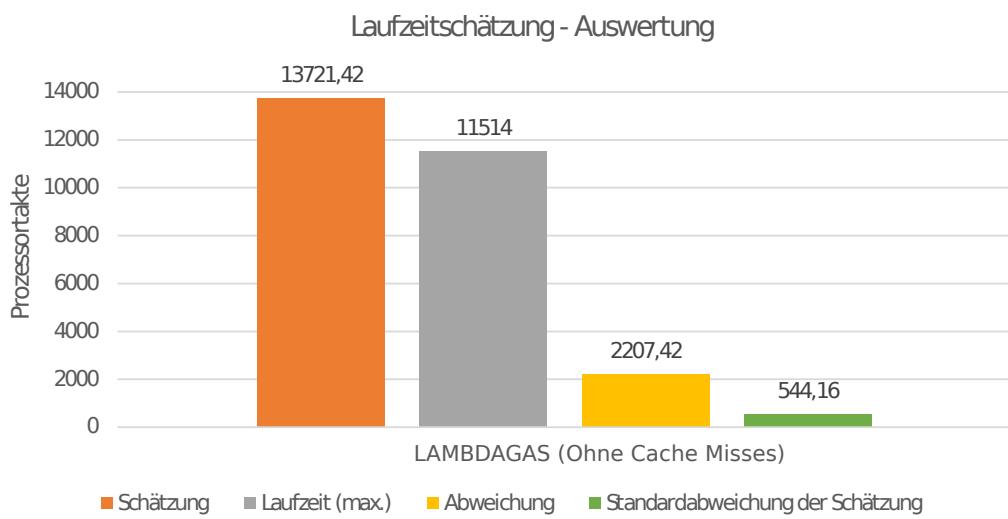


Abbildung 49: Laufzeitauswertung für LAMBDA GAS Modell ohne “Cache-Miss-Zeit”

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit "Konzeption einer Prozesskette zur Analyse der Ausführungsduer von Simulink Modellen auf eingebetteten Systemen" selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, sowie alle Zitate entsprechend kenntlich gemacht habe.

Magdeburg, den 16. Mai 2017

Markus Wolf