Rudolf Weißenbacher, Bsc

Model-based Scheduling of Safety-Critical Embedded Automotive Multi-Core Systems

Masterarbeit

Zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Telematik

eingereicht am

Institut für Technische Informatik

Technische Universität Graz

Vorstand: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Römer

Beurteiler:

Dipl.-Ing. Dr.techn. Christian Kreiner

Institut für Technische Informatik

Betreuer:

Dipl.-Ing. Georg Macher,

Institut für Technische Informatik,

Dipl.-Ing. Dr.techn. Jürgen Fabian

Institut für Fahrzeugtechnik

Graz, im November 2015

Abstract

Modern vehicles have more and more mechatronic functions such as lane detection, parking assistance or driving assistants. These functions are based on embedded electronic systems. The new functions, the complexity of the control unit and the communication of the various components are enormously increasing. The processing power of the actual control devices is no longer sufficient for this scope. Therefore, powerful multicore versions are necessary for future computing controllers. Because of the rapid development (e.g. expensive new features, consolidation of control units, the approached limit regarding thermal aspect), a general shift to multicore devices will be necessary in the future. To minimize the costs of the new developments and the integration of the existing features in future vehicles, reuse of existing functions is essential. This means that existing single-core applications have to be migrated to multi-core systems and therefore be divided into multiple cores. Considering safety-critical aspects, special attention has to be paid to shared resources; such as sensors, actuators or memory. In particular, the real-time capability of safety-critical applications or tasks and runnables has to be ensured. This can be accomplished by well-chosen offline scheduling. The analysis of resource workload (e.g. cores, tasks or periphery) might result in optimal scheduling and distribution of tasks to different cores. This analysis has to be executed during development and has to capture especially the functionalities and states that are critical for the system. In safety-critical functions, the Worst Case Execution Time Analysis should lead to satisfactory results. The aims of this work are to show the porting of the operating system from single-core to multicore, as well as to prepare the migration and to document its difficulties and necessary steps and tools. It is important to reuse the existing toolchain which is based on Enterprise Architect. To continue to work with the available tools, the integration of analysis tools into the existing toolchain is necessary. Attention is directed to the integration of the analysis tools SymTA/S into the automotive development framework and the analyses which are necessary for the multicore transition (e.g. communication overhead). By using SymTA/S, it is demonstrated, how an existing tool chain can be expanded by a timing analysis tool and how real-time behavior can be optimized.

Kurzfassung

Moderne Automobile verfügen über immer mehr mechatronische Funktionen wie Spurerkennung, Einparkhilfen oder Fahrassistenten. Diese Funktionen basierend auf eingebetteten elektronischen Systemen. Durch die neuen Funktionen werden die Komplexität der Steuergeräte und die Kommunikation der verschiedenen Komponenten massiv erhöht. Da die Rechenleistung der bisherigen Steuergeräte für diesen Umfang nicht mehr ausreicht, sind für zukünftige Steuergeräte rechenstarke Multicore-Ausführungen notwendig. Durch die rasche Entwicklung (aufwendigere neue Funktionen, Zusammenlegung von Steuergeräten und der aus thermischer Sicht bereits erreichten Takthöchstgrenze) wird in Zukunft ein genereller Umstieg auf Multicore-Geräte notwendig sein. Um die Kosten für Neuentwicklungen und Integration bereits vorhandener Funktionen gering zu halten, sollen bereits vorhandene, implementierte Funktionen auch in zukünftigen Fahrzeugen wiederverwendet werden können. Das bedeutet, dass vorhandene Singlecore-Anwendungen in Multicore-Systeme integriert und dadurch auch auf mehrere Cores aufgeteilt werden müssen. Da dies aus sicherheitskritischen Aspekten nicht trivial ist, muss besonderes Augenmerk auf gemeinsame Ressourcen, wie Sensoren, Aktuatoren oder Speicher genommen werden. Im Besonderen muss aber auch die Echtzeitfähigkeit sicherheitskritischer Anwendungen bzw. Tasks und Runnables unter allen Umständen sichergestellt werden. Dies muss durch gut gewähltes Offline-Scheduling gewährleistet werden. Eine Analyse der Ressourcenauslastung (z.B. Cores, Tasks bzw. der Peripherie) kann ein optimales Scheduling und eine optimale Aufteilung der Tasks auf die verschiedenen Cores erzielen. Diese Analyse muss zur Entwicklungszeit geschehen und vor allem systemkritische Funktionalitäten und Zustände erfassen. Bei sicherheitskritischen Funktionen muss die Worst Case Execution Time Analyse ein zufriedenstellendes Ergebnis bringen. Ziel der Arbeit ist es, die Schwierigkeiten und Probleme der Betriebs-systemportierung von Singlecore -auf ein Multicore-Systems zu zeigen, die Migration vorzubereiten und die, nötige Schritte und Tools zu dokumentieren. Wichtig ist, die bereits vorhandene Toolchain, deren Grundlage Enterprise Architect ist, weiterzuverwenden und Möglichkeiten zu finden, Analysetools in diese bereits vorhandene Toolchain zu integrieren, um mit den bestehenden Tools weiterarbeiten zu können. Augenmerk wird dabei auf die Integration des Analysetools SymTA/S in das automotive Entwicklungsframework und die für den Multicore-Umstieg nötigen Analysen (z.B. Kommunikationsoverhead) gelegt. Mit SymTA/S wird gezeigt, wie eine vorhandene Toolchain mit einem Timing-Analyse-Tool erweitert werden kann und Optimierungen im Echtzeitverhalten vorgenommen werden können.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am 1.11.2015 ……..

(Unterschrift)

Danksagung

Diese Diplomarbeit wurde am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt. Bedanken möchte ich mich bei meinen Betreuern Georg Macher und Christian Kreiner am Institut für Technische Informatik und Jürgen Fabian am Institut für Fahrzeugtechnik. Mit fachlicher und kollegialer Beratung standen sie mir während meiner Zeit am Institut für Fragen immer zur Seite. Ein besonderer Dank gilt auch der Firma Symtavision und Herrn Hagner, die mir das Analysetool zur Verfügung stellten und auch immer mit technischem Support kompetente Hilfestellung boten. Weiters möchte ich Eugen Brenner danken, der während meines Studiums immer ein offenes Ohr für Studierende hatte. In diesem Sinne möchte ich auch allen Menschen der Basisgruppe Telematik bedanken, die mich durch mein Studium begleitet und viel Zeit und Arbeitsstunden mit mir verbracht haben.

Meinen Eltern möchte ich für die Unterstützung während meiner beruflichen Ausbildung und meines Studiums danken. Sie haben mich meinen Weg gehen lassen und immer an mich geglaubt. Besonderen Dank gilt an meine Verlobte Barbara, die mich zum Durchhalten und Beenden meines Studiums motiviert und großartig während dem Verfassen meiner Arbeit unterstützt hat. Vielen Dank dafür! Der kleine Oliver hat es mir zwar schwer gemacht, mich zum Schreiben meiner Arbeit zu setzen, aber ohne ihn, hätte ich weniger Motivation gefunden, schnell fertig zu werden. Danke auch an Kübi, der mir als Babysitter und Freund immer mit einem offenen Ohr zur Seite stand und mir viel Zeit freischaufelte. Ein Dankeschön an alle, die mich in meinem Studium begleitet haben. Ein besonderes Danke aber an Hermann, Julia, Werner, Lemmi, Hannes, Thomas, Eva und Daniel.

Rudolf Weißenbacher

Graz, November 2015

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Innovation und Fortschritt haben im Automobilbereich deutlich die Komplexität der Elektronik und Elektrik erhöht. Dadurch ergibt sich eine Vielzahl an neuen Möglichkeiten, die Unterhaltung, den Komfort und schlussendlich die für Kunden und Gesellschaft wichtige Sicherheit zu verbessern. Gleichzeitig wird durch Funktionen wie Entertainment, ESP, ACC die Komplexität und Anforderung der Steuergeräte massiv erhöht [1].
Diese Innovationen, in vor allem Premiumfahrzeugen, werden bis zu 90% anhand von eingebetteter Software umgesetzt [47]. Dadurch belaufen sich auch 20 bis 30 Prozent des Fahrzeugwertes nur auf diese Komponenten [3]. Die momentanen Schwachstellen sind die “leistungsschwachen” acht-bis-32 Bit-ECUs, die sich 2010 auf mehr auf 70 in einem Fahrzeug beliefen [3]. Die Integration dieser bestehenden Systeme ist ein Muss, um die Kosten für zukünftige einzusparen. Eine Steigerung der Performance der Steuergeräte bzw. Prozessoren ist aber aus den zuvor genannten Gründen unausweichlich. Eine Steigerung der Prozessorleistung ist durch die besonderen Anforderungen an Echtzeitsysteme im Automotivbereich (Kühlleistung, Platzmangel, Einfluss der Umgebung) bereits ausgereizt. Mehrkernprozessoren mit zwei bis drei Cores und nahezu gleichbleibender Rechenleitung pro Core werden daher von vielen Herstellern zur Performancesteigerung angeboten. Hierbei gilt die Herausforderung, bestehende Software auf mehrere Kerne zu verteilen. Es sind die Intercore-Kommunikation, Freedom from Interference, gemeinsame Ressourcen und auch die steigenden Sicherheitsaspekte (sicherheitskritisch, sicherheitsrelevant und zuverlässig) zu beachten, welche die Zusammenlegung von Steuergeräten und deren Funktionen auf Multicore-Geräte zu einer Herausforderung machen [2].
Um eben diese Ziele zu erreichen, muss eine Form der Mikrocontroller gefunden werden, welche diese Herausforderungen der Vernetzung und Leistung erfüllen. Um die steigende Anzahl der Controller in einem Fahrzeug einzubremsen können Multicore-Controller eingesetzt werden. Durch die steigende Vernetzung der Fahrzeuge, der Verkehrsleitsysteme bzw. der Außenwelt, sind diese Multicore-Steuergeräte nur eine kleiner Schritt um den stetig steigenden Leistungsanforderungen gerecht zu werden. Altbewährte hydraulische oder mechanische Systeme werden zukünftig durch Softwarelösungen abgelöst. Daher muss nicht nur für die Hardware, sondern auch für die Software eine nachhaltige Lösung gefunden werden, um Kosten zu sparen.

Ein Grund zur Zunahme von Steuergeräten bzw. der Steigerung von Fahrzeugkosten ist die Steigerung der Standardausstattung in allen Fahrzeugklassen. Dies führt in kleineren Fahrzeugtypen jedoch zu Platzproblemen, welche wiederum durch Multicore-Systeme gelöst werden können. Dadurch können Steuergeräte, die räumlich in der Nähe liegen, zusammengefasst werden. Die steigenden Funktionen und Codezeilen pro Steuergerät werden dadurch aber massiv steigen [4]. Ein weiterer Aspekt, der für Multicore-Geräte spricht, ist, dass der Stromverbrauch und die damit verbundene Wärmeabgabe stärker als die Taktrate und der Performancegewinn steigen. Des Weiteren kann auch die Speichergeschwindigkeit nicht mit dem Entwicklungstempo mithalten [5].

Eine Möglichkeit, die Entwicklungskosten so gering wie möglich halten zu können, sind zukünftige Systeme, die auf Automotive Open Systems Architecture (AUTOSAR) basieren. Hierbei sollen die Schnittstellen seitens der Software vereinheitlicht werden, um Anpassungen und Adaptierungen so einheitlich wie möglich zu realisieren [6]. Diese rasante Entwicklung der Fahrzeugassistenzsysteme und der Elektronik benötigt ein Höchstmaß an Zuverlässigkeit der Multicore-Systeme in sicherheitskritischen Bereichen. Diese harte Echtzeitanforderung wird durch verschiedene Hardwareeigenschaften und einem statischen Scheduling erreicht. Die Erstellung des Schedulings muss, um ein deterministisches Echtzeitverhalten zu gewährleisten, bereits im Vorhinein erfolgen, um alle möglichen Zustände abzubilden. Um die vorhandenen Funktionen richtig aufteilen zu können, sind Periodenlängen, Abhängigkeiten und die Zugriffe auf Ressourcen vollständig mit allen Parametern nötig. Diese Methode des Offline-Schedulings verhindert jedoch ein Reagieren auf nicht vorhersehbare Ereignisse und schränkt auch die Adaptionsfähigkeit ein. Daher muss im Vorfeld schon jedes Szenario in Betracht gezogen werden. Auf jeden Fall muss für sicherheitskritische Bereiche eine Worst Case Execution Time (WCET) Analyse ein zufriedenstellendes und sicheres Ergebnis bringen [7], [8].

1.2 Zielsetzung

Ziel dieser Arbeit ist es, die nötigen Schritte für die Umstellung eines Systems von einem Singlecore- auf ein Multicore-Betriebssystem bzw. -Steuergerät zu analysieren und auch dessen Schwierigkeiten aufzeigen. Da die rasante Entwicklung der elektronischen Systeme sehr kostenaufwendig ist (40% der Entwicklungskosten 2015, [12], [13]) und ein Höchstmaß an Zuverlässigkeit in sicherheitskritischen Bereichen erfordert, wird angestrebt, vorhandene Systeme in diese Multicore-Systeme zu integrieren. Durch ein Weiterverwenden der vorhandenen Toolchain bzw. deren Erweiterung durch nötige Tools, sollen diese Kosten so

gering wie möglich gehalten werden. Bestimmte Steuergeräte, wie der in dieser Arbeit betrachtete Infineon Tricore, unterstützen z.B. auch die Lockstep-Funktion [67], die maßgeblich zur Kontrolle der richtigen Arbeitsweise des Steuergerätes beiträgt. Dadurch muss bei der Aufteilung der Funktionen dieser zusätzlich Rechenlast erzeugender Faktor weniger stark berücksichtigt werden. Es kann auf den zwei Cores parallel gearbeitet bzw. programmiert werden.

Die steigende Intercore-Kommunikation, die Verwaltung der gemeinsamen Ressourcen und die Blockier-Zeiten derer, stellen mit der optimalen Clusterung und den möglichst geringen Kontextwechselverlusten große Herausforderungen dar. Diese Herausforderungen können durch ein ausgeklügeltes Offline-Scheduling bewältigt werden. Diese harte Echtzeitanforderung wird durch verschiedene Hardwareeigenschaften und einem statischen Scheduling erreicht. Diese wird schon zum Entwicklungszeitpunkt festgelegt und muss kontinuierlich optimiert werden, um ein bestmögliches Scheduling zu erreichen. Um all diese Ziele zu erreichen, wird ein Tool gesucht, das diese Aufgaben meistert und gleichzeitig auf bereits vorhandenen Toolchains aufbaut und diese erweitern kann. Weiters wird analysiert, wie und unter welchen Bedingungen Timing Analysis Tool die gewünschten Resultate liefern können.

Anhand des Infineon Aurix Multicore-Systems wird versucht, die Aufteilung der verfügbaren Daten (Tasks, Runnables, Ressourcen) und die Migration vorzubereiten. Zu diesem Zwecke wird das Analysetool SymTA/S betrachtet: welche Möglichkeiten es zur Umsetzung der Anforderungen an eine Offline-Scheduling Analyse und die damit einhergehenden Probleme bietet. Besonderes Augenmerk wird auf die sicherheitskritischen Aspekte gelegt: Überwachung der richtigen Ausführung der Funktionen mittels Lockstep, Blockier-Zeiten feststellen und minimieren sowie Deadlocks ausschließen.

Um auch die Nachhaltigkeit bestehender Systeme und verwendeter Toolchains zu gewährleisten, wird in dieser Arbeit außerdem die Integrierbarkeit des Timinganalysetools SymTA/S mit verschiedenen Schnittstellen überprüft und auf deren Alltagspraxis getestet. So können bereits vorhandene Systeme und Erfahrungen im Entwicklungsbereich im Multicore-Bereich fortgeführt werden. Die Timing Analyse stellt eine kostbare Erweiterung in der Toolbridge dar und sollte so einfach wie möglich integriert werden können, um Kosten zu sparen und den Entwicklungsprozess und dessen Tools nicht neu aufstellen zu müssen.

1.3 Gliederung

In Kapitel 2 wird vor allem auf die grundlegenden Elemente eingegangen, die für das Verständnis notwendig sind, um eingebettete Systeme im Automobilbereich zu verstehen und auf deren Problematiken eingehen zu können. Aufgrund der Komplexität der Themen werden diese nur grundlegend erklärt und angeschnitten. Es werden Begriffe von eingebetteten Systemen erklärt, die für die Umsetzung eines effektiven Schedulings nötig sind. Die erläuterten Probleme können nicht alle durch ein Analysetool gelöst werden, sondern werden lediglich erkannt und müssen in anderen Entwicklungsschritten beachtet und gelöst werden. In Kapitel 3 werden die Tools betrachtet, die in Zusammenhang mit der Umstellung eines Single- auf ein Multicore-System auf dem Markt sind. Diese Tools sollen sich auch in eine bereits vorhandene Toolchain integrieren lassen und mit anderen nützlichen Entwicklungsumgebungen kompatibel sein. In Kapitel 4 wird die Analyse mittels SymTA/S erläutert, wie die Importierung vorhandener Systeme, mit Focus auf Remote Interface und Python Scripting, funktioniert, und welche Ergebnisse aus den zahlreichen möglichen Analysen gewonnen werden können. Timing Analysen im Bereich WCET, Kommunikationsoverhead, Speicherzugriffe und das Aufzeigen freier Ressourcen der Rechenkerne stehen im Mittelpunkt. Im Kapitel 5 werden die Ergebnisse noch einmal zusammengefasst und auf den Punkt gebracht. Weitere Themen und Problemstellungen werden in Kapitel 6 aufgezeigt.

2. Stand der Technik und Literaturrecherche

Um Embedded Automotive Systems analysieren zu können, ist es zuerst nötig, die Ausgangslage zu erfassen, wesentliche Merkmale festzuhalten und zu eruieren, welche Arbeit in diesem Bereich bereits geleistet wurde und welche Vorgehensweisen dazu nötig sind. In diesem Kapitel wird ein Überblick über die nötigen Voraussetzungen und die zu lösenden Probleme geschaffen. Jedes dieser Themen ist umfangreich und eine große Herausforderung, die es zu bewältigen gilt. All diese Voraussetzungen und Probleme im Detail zu beleuchten, würde jedoch den Umfang dieser Arbeit sprengen. Deshalb werden die Themen nur für das grundsätzliche Verständnis angeschnitten, um einen Überblick zu bekommen.

2.1 Embedded Automotive Systems

In Automobilbereich wird die Überwachung von zahlreichen Funktionen wie die optimale Verbrennung im Motor, Überwachung der Sensordaten, Steuerung des Antiblockiersystems oder einfach nur die Steuerung des Entertainmentbereiches immer umfangreicher. Diese Aufgaben werden durch Echtzeit-Systeme umgesetzt, welche zum jeweiligen Zeitpunkt, mit ihren vorhandenen Daten, korrekt laufen müssen. Hierbei wird in den meisten Fällen zwischen „harter" und „weicher" Echtzeitanforderung unterschieden.
"*Hard real-time tasks have critical deadlines that are to be met in all working scenarios to avoid catastrophic consequences. In contrast, soft real-time tasks (e.g., multimedia tasks) are those whose deadlines are less critical such that missing the deadlines occasionally has minimal effect on the performance of the system."* [B. Duwairi, G. Manimaran 2003], [9]

Deshalb sind vor allem Robustheit, Zuverlässigkeit und Sicherheit wichtige Punkte, die sich auf die Begriffe „harte und weiche Echtzeit" auswirken [10]. Vor allem sicherheitskritische Komponenten benötigen eine „harte" Echtzeit und dürfen ihre Deadlines auf keinen Fall versäumen. Da Fahrzeuge und deren Funktionen sehr komplex aufgebaut sind, müssen alle diese sicherheitskritischen Funktionen und Aufgaben erhoben und kontrolliert werden. Eine sicherheitskritische Funktion ist unter vielen anderen die Bremsanlage. Das Regelungssystem bzw. auch Überwachungssystem einer solchen Bremsanlage besteht aus mehreren Komponenten. Durch zusätzliche Funktionen, die in Verbindung mit der Bremsanlage, oder deren Funktionswerte verwenden, wie zum Beispiel ABS, wird der Zugriff auf Sensoren, Aktuatoren, Steuergeräte und auch die Sollwertgeber sehr komplex (siehe Abbildung 1 Vernetzte Steuergeräte in einem Fahrzeug [4]. Dies erfordert ein Überwachen

und Regeln aller zusammenhängenden Funktionen und benötigt auch eine Vernetzung der Steuergeräte untereinander.

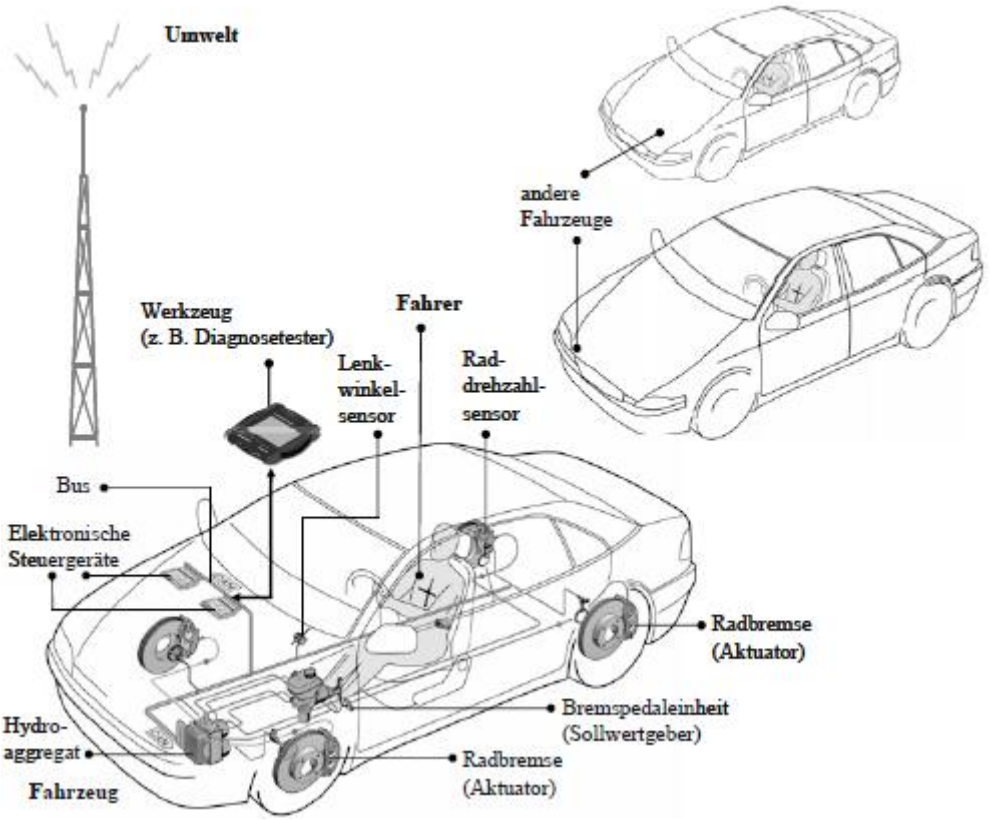


Abbildung 1 Vernetzte Steuergeräte in einem Fahrzeug [4]

In einem Fahrzeug ist zum einen der Fahrer direkt für die Sollwerte zuständig (Betätigen der Pedale), zum anderen kann das Fahrzeug selbst diese Sollwerte generieren (Tempomat, Motorsteuerkennlinien). Diese Faktoren beeinflussen ein Echtzeitsystem. Um auf sicherheitskritische Bereiche reagieren zu können, müssen alle erdenklichen Fälle durchdacht und mögliche Lösungen und sichere Zustände umgesetzt und erreicht werden [4].

2.1.1 Harte und weiche Echtzeitanforderungen

Echtzeitsysteme verarbeiten ihre Eingabesignale in einer bestimmten Zeit zu einem Ausgabewert. Durch die zeitliche Anforderung an diese Bearbeitung können Echtzeitsysteme in zwei Klassen eingeteilt werden. Weiche Echtzeitanforderungen bestehen darin, dass sie die Signale meist schnell genug abarbeiten und das rechtzeitige Erfüllen der Deadline eher eine Richtlinie ist. Das Überschreiten der Zeitvorgabe zieht keine schweren Konsequenzen nach sich. Harte Anforderungen werten ein Überschreiten jedoch als

Versagen und ziehen schwerwiegende Konsequenzen nach sich. Aus diesem Grund wird bei harten Echtzeitanforderungen garantiert, dass diese zu jedem Zeitpunkt eingehalten werden. Dies muss natürlich auch von den Entwicklern gewährleistet werden. In [4] werden die Echtzeitanforderungen wie folgt definiert:
„*Eine Echtzeitanforderung an einen Task ist hart und die Task ist eine harte Echtzeit-Task, wenn die Validierung gefordert wird, dass die spezifizierten Echtzeitanforderungen der Task immer erfüllt werden. Unter Validierung wird ein Nachweis mittels einer nachprüfbaren und korrekten Methode verstanden. Entsprechend werden alle Echtzeitanforderungen an eine Task als weich bezeichnet und die Task ist eine weiche Echtzeit-Task, wenn ein Nachweis dieser Art nicht gefordert wird*" [Schäuffele, 2000], [4]. Echtzeitanforderungen an einen Task haben also keinen Zusammenhang mit der Geschwindigkeit oder der Sicherheitsrelevanz eines solchen [4].

2.1.2 Definition von Echtzeitanforderungen

Um ein Echtzeitsystem planen und steuern zu können, ist es notwendig, die zeitlichen Anforderungen der Tasks so genau wie möglich zu beschreiben. Ein wesentlicher Punkt ist der Unterschied zwischen Zeitpunkt und Zeitspanne. Zwei wichtige Zeitpunkte sind der Aktivierungszeitpunkt und der Deadline-Zeitpunkt.

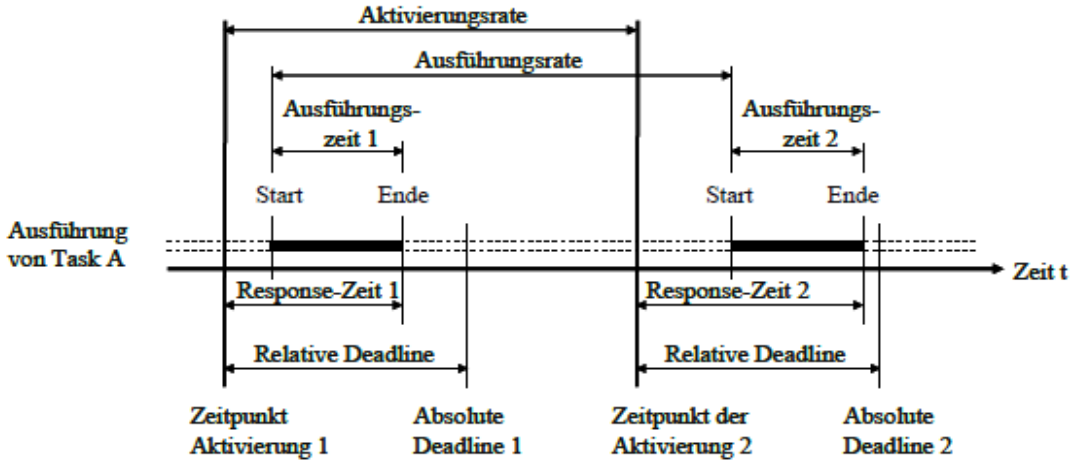


Abbildung 2 Zeitpunkte und Zeitspannen eines Task

- Aktivierungszeitpunkt: Der Task wird angestoßen und zur Ausführung freigegeben
- Deadline-Zeitpunkt: Spätester Zeitpunkt, an dem der Task abgeschlossen sein muss

- Response-Zeit: Zeitspanne oder Dauer von der Aktivierung eines Tasks, bis zu dessen Beendigung, bzw. Beendigung der Ausführungszeit
- Maximale Response-Zeit: Relative Deadline
- Aktivierungsrate: Zeitspanne zwischen den Aktivierungszeitpunkten eines Tasks
- Ausführungsrate: Zeitspanne zwischen zwei Ausführungszeitpunkten

Festzuhalten ist, dass es zwischen der Echtzeitanforderung (Zeitschranke) und der wirklich benötigten Ausführungszeit (Execution Time) der Tasks zu unterscheiden gilt. Wird der Task, wie in Abbildung 2, nicht unterbrochen, so ist die Ausführungszeit leicht abzulesen. Wird der Task jedoch unterbrochen, so ist die Ausführungszeit die Summe aller Zeitspannen zwischen den Start und Endzeitpunkten eines Tasks. Die Unterbrechung eines Tasks kann z.B. vom Scheduler erfolgen, der die Rechenzeit einem höher priorisierten Task zur Verfügung stellt oder der unterbrochene Task stellt seine Rechenzeit „freiwillig", weil er z.B. auf einen anderen Task warten muss, zur Verfügung [4].

2.1.3 Korrektes Zeitverhalten von sicherheitskritischen Tasks

Eines der Hauptprobleme bei der Zuweisung der Software auf verschiedene Cores ist der Beweis, dass die Konfiguration alle Echtzeiteigenschaften des Systems erfüllt. Besonders für große, umfangreiche Systeme ist dieser Beweis sehr schwierig. Da Standardprozessoren die Ausführungszeit im durchschnittlichen Fall optimieren, ist das im sicherheitskritischen Bereich nicht genügend. Die Echtzeitfähigkeit muss in jedem Fall garantiert werden. Um systemkritische Prozesse vor unwichtigen oder unnötigen Interrupts zu schützen, können diese zum Beispiel auf einem Core mit höherer Priorität isoliert werden. Durch Laufzeitmessungen, wie z.B. „traceGURU" [49], werden Events aufgezeichnet, die für die Einhaltung der Laufzeit wesentlich sind. Tasks, Runnables, IR werden zu allen möglichen Ereignissen aufgezeichnet. Meist wird zwischen Start, Ende und Aktivierung unterschieden. Weitere Messpunkte können vom Benutzer noch eingefügt werden, um Codeabschnitte zu analysieren. Die Messung der WCET der Tasks und anderer zu messenden Größen (z.B. IR, Runnables) und etwaiger Unterbrechungen derer kann so festgestellt werden. Wichtig bei solchen Analysen ist auch noch die Nicht-Beeinträchtigung der Software durch die Messung, da diese ansonsten das Messergebnis verfälschen würde. Durch die Zustände „ready",
„running" und „suspended" oder vergleichbarer States kann das OS ausgemessen werden. Durch geeignete Scheduling-Analysen können Engpässe erfasst und aufgezeigt werden, auch wenn diese bei der Messung nicht entstanden sind. Durch sogenannte "Event Models"

können dynamische Variablen erzeugt werden, die das System in kritische Situationen bringt und somit die Laufzeiten und die Auslastung maximal werden lässt. Solche simulierten Engpässe, die in Testfällen nicht aufkommen, könnten jedoch in WC-Szenarien auftreten und somit Deadlines kritischer Tasks beeinträchtigen oder hinauszögern, was bei sicherheitskritischen Fällen auf keinen Fall auftreten darf. Weitere Methoden sind unter [33] zu finden.

2.2 Embedded Automotive Multicore-Systems

Bei der Umstellung auf ein Multicore-System muss neben der anzupassenden Software vor allem Augenmerk auf den Aufbau der Hardware gelegt werden, um diese beiden Punkte von Anfang an aufeinander abzustimmen. Die harten Echtzeitanforderungen, Zuverlässigkeit und Sicherheitsanforderungen stellen die größten Herausforderungen im Multicore-Bereich dar. Was für Single Core-Systeme schon eine große Herausforderung war, ist natürlich für Multicore-Systeme noch schwieriger. Weitere Herausforderungen und Schwierigkeiten sind die des zur Verfügung stehenden Personals und der Tools. Wenige Programmierer im Multicore-Bereich, die kaum zur Verfügung stehenden Tools für Debugging, Simulation und Optimierung bedeuten eine zusätzliche Herausforderung.
Bei der Wahl des Multicore-Systems gibt es noch zwischen homogenen und heterogenen Systemen zu unterscheiden. Homogene Systeme arbeiten mit identischen CPUs und heterogene mit verschieden aufgebauten CPUs. Wobei homogene System meist im Lockstep Modus betrieben werden, können divergente zusätzlich zur funktionalen Sicherheit beitragen. Dies erfordert jedoch meist eine komplexere Entwicklung [38], [39]. Durch den Aufbau und die Zuordnung der Speichermodule kann noch zwischen „non-uniform memory architecture“ (NUMA) und „uniform memory architecture“ (UMA) wie in [68] beschrieben, unterschieden werden. Die meisten Multicore-Anbieter stellen auf ihren Mikrokontrollern zwei Rechenkerne zur Verfügung. Der TMS570 von Texas Instrument, MPC574xP von Freescale und AURIX von Infineon stellen einige der zur Verfügung stehende Mikrocontroller dar. Unter [68] sind noch weitere Multicore-Systeme und Betriebssysteme wie „ERIKA Enterprise“ oder „Vector“ beschrieben. Viele Sicherheitsaspekte können durch Multicore-Systeme besser und effektiver umgesetzt werden. Eine Trennung von Hard- und Software oder Redundanz ist einfacher umzusetzen. Treibende Kräfte sind in diesem Bereich die Automobilhersteller, die große Anforderungen an Echtzeitsysteme stellen. Um diese zu erreichen, wird der AUTOSAR Standard genutzt, um diese Systeme zu entwickeln. Die Zuteilung der Task zu den Cores erfolgt offline zur Entwicklungszeit. Empfohlen wird auch die Festlegung der Basissoftware auf einem Core [6]. Eine wesentliche Aufgabe von

Embedded Automotive Multicore-Systems stellt die funktionale Sicherheit des Mikrocontrollers dar. Da sicherheitsrelevante Funktionen auf diesem ausgeführt werden, bedarf es speziellen Anforderungen, die das richtige Abarbeiten der Funktionen auf Soft- und Hardwareebene gewährleisten. Die Überwachung der Register, Frequenzfehler, Takt- und Spannungsüberwachung, Temperaturüberwachung und vieles mehr stellen eine große Herausforderung an die Hardware. Besonders die Speicher und deren Inhalte sowie Peripheriemodule müssen gesondert überwacht werden. Meist besitzen solche Bausteine eigene Überwachungsfunktionen, die ein richtiges Arbeiten garantieren. Zur Überprüfung seitens der Software müssen die AUTOSAR-Spezifikationen eingehalten werden. Die Überwachung der Funktionsabläufe kann mittels eines Watchdog-Managers erfolgen. Hierbei werden die Ausführungsdauer und die Häufigkeit der Funktionsaufrufe, die Speicherauslastung, Datenübertragungen auf dem Bussystem oder auch Hardwarekomponenten durch Plausibilitätsabfragen auf Richtigkeit überprüft. Ein Auftreten von „deadlocks“, Aushungern von Tasks, „race conditions“ oder Prioritätsumkehr muss ausgeschlossen und „freedom of interference“ und ein Einhalten der Ausführungszeiten garantiert werden [68].

2.3 Aufbau von Mikrocontrollern

Um zu verstehen auf welchen Systemen die Software lauffähig sein muss, wird in den folgenden Punkten die Hardware näher gebracht. Mikrocontroller zeichnen sich im Unterschied zu Mikroprozessoren dadurch aus, dass ihnen Peripherie und deren Funktion angehören und dieses System speziell zur Steuerung von anderen Geräten oder eingebetteten Systemen verwendet wird. Auszeichnend für Mikrocontroller ist weiters, dass diese programmierbare Speicherblöcke beinhalten, über welche der Controller konfiguriert wird. Diese Peripherie kann im Controller Interrupts auslösen, die dann weitere Ereignisse steuern. Kontrolliert werden die Mikrocontroller über oft schon integrierte Watchdog-Schaltungen.

2.3.1 Grundlagen von Singlecore Mikrocontrollern

Auf einfachen Mikrocontrollern sind meist die nötigen Peripheriegeräte wie Speicher, Schnittstellen, -Ausgänge, AD-Wandler und viele weitere Komponenten vorhanden oder möglich. Auf komplexere Module bzw. Controller wird im nächsten Punkt genauer eingegangen. Die einfache Programmierung und Ausführung der Controller lässt auch einen flexiblen Einsatzbereich in allen möglichen Alltagsbereichen zu und ermöglicht eine günstige Produktion in großer Stückzahl.

Mikrocontroller bestehen im Allgemeinen aus folgenden Komponenten (siehe Abbildung 3):

- Mikroprozessor (CPU) mit Steuer- und Rechenwerk
- Ein- und Ausgabeeinheiten (I/O), wie Schaltungen für Interrupts, Bussysteme oder AD-Wandler
- Programmspeicher (meist Lesespeicher) und Datenspeicher (Lese- und Schreibspeicher)
- Bussystem (CAN, LIN, FlexRay)
- Taktgenerator (Oszillator)
- Watchdog (Überwachung).

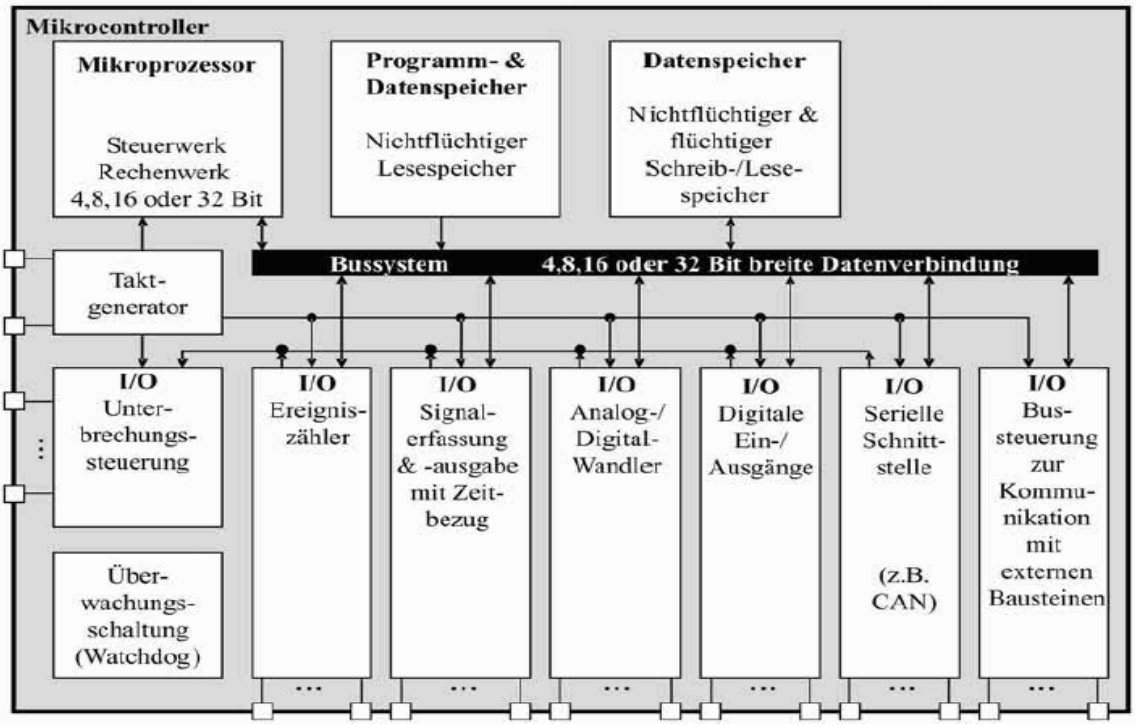


Abbildung 3 Aufbau eines Mikrocontrollers [4]

Optional und je nach Komplexität können noch folgende Komponenten in Controllern eingesetzt werden:

- Getrennte Bussysteme für Daten und Adressspeicher
- Kryptographie
- Touchpads.

Die Umsetzung von Controllern erfolgt meist auf einem Chip. Durch Erweiterungen können dann auch externe Speicher oder andere Peripherien hinzugefügt werden. [4]

2.3.2 Aufbau eines Multi-Core Controllers

Diese Arbeit bezieht sich auf ein Multicore-System, das mittels des Infineon Aurix Application Kit [11] realisiert wird. Der Controller selbst besteht aus einem Infineon TC275C mit eben drei Kernen. Diese sind als 32-bit-Tricore-CPUs mit 200 MHz spezifiziert, welche im gesamten Automobil-Temperaturbereich arbeiten. Dafür stehen auch jedem Kern eigene Speichermodule zur Verfügung. Zusätzlich ist für das verwendete Application Kit ein Touchscreen (320x240 Pixel) vorhanden. Diverse Anschlüsse z.B. für externe Speicher (SD-Karte) sind ebenfalls verfügbar, siehe Abbildung 4

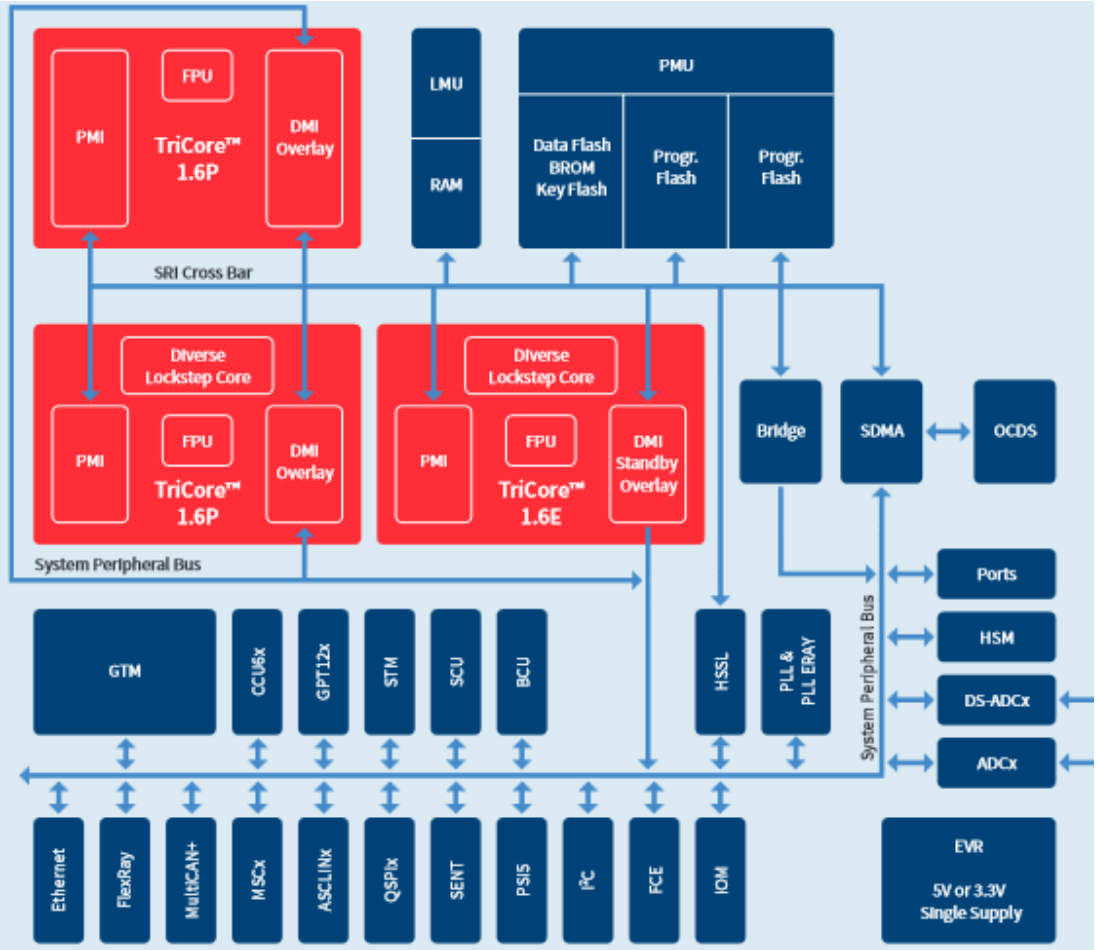


Abbildung 4 Blockschaltbild eines MC Mikrocontroller [11]

Um als Mehrkern-Steuergerät den Anforderungen im Automobilbereich gerecht zu werden, sind spezielle Funktionen realisiert, die den sicherheitskritischen Anforderungen in einem Fahrzeug gerecht werden. Die drei Rechenkerne arbeiten unabhängig voneinander und können auch im Lockstep-Modus betrieben werden. Die Verbindung des ROM- und RAM-Speichers für den Daten- und Programmspeicher und der DMA-Controller sind mittels eines

sicheren Buses realisiert. Ein eigenes Sicherheitsmodul sorgt für die Fehlererkennung der sicherheitskritischen Elemente des Controllers. Bei Fehlern werden Aktionen wie Interrupts, Abschalten von Modulen oder das Zurücksetzen des Mikrocontrollers aktiviert. Gefunden werden können Fehler auf mehrere Weisen wie dem Lockstep Komparator (CPU Fehler), verschiedene SRAM und Flash Prüfungen (Bitfehler, Buffer, Adressierungsfehler, …) oder auch Shared Resource Interconnect (SRI) (Schutz des Core-Busses). Die im Tricore vorgesehene Lockstep-Architektur erleichtert die nötige Umsetzung des ISO 26262-konformen Systems. Dies wird auch durch den diversen Lockstep-Kern mit Taktverzögerung und die redundanten Timer Module gewährleistet. Durch Konfiguration der Sicherheitseinstellungen im Zugriffberechtigungssystem und der Sicherheits-Managementeinheit kann eine Sicherheitsanforderung bis ASIL-D erreicht werden [11].

2.4 RTOS

Um Mikrocontroller, die in jedem Steuergerät vorhanden sind, zu konfigurieren, ist es nötig, diese Steuergeräte mittels eines Betriebssystems zu steuern. Alle zur Verfügung stehenden Systemressourcen wie Speicher, Ein- und Ausgabeschnittstellen oder Rechenzeit durch die CPU, werden vom Betriebssystem organisiert. Die Verwaltung der Tasks und somit die Aufgabenverteilung erfolgt über das Betriebssystem, das somit einen der wichtigsten Teile eines Echtzeitsystems darstellt. Das OS gewährleistet, dass definierte Rechenzeiten und deren Verzögerungen nicht überschritten, sondern eingehalten werden. Für sicherheitskritische Aspekte muss das Betriebssystem immer in einem bekannten und sicheren Status sein. Dies ist ein wesentliches Merkmal von Echtzeitbetriebssystemen. Das Scheduling wird bereits im Vorhinein festgelegt und ist somit statisch.
Die meisten dieser Systeme verwenden TASK-Priorisierung, falls mehrere Tasks „ready" sind. Durch das präemptive Scheduling wird gewährleistet, dass immer der Task mit der höchsten Priorität, auch wenn andere dadurch unterbrochen werden müssen, ausgeführt wird und das System somit richtig und sicher läuft. Sind die Prioritäten gleich, wird meist die FIFO-Warteschlange umgesetzt, um sicherzugehen, dass jeder Task bis zu einer gewissen Zeit ausgeführt wird. Das Scheduling-Verfahren wird statisch festgelegt bzw. kann die Priorität des Tasks anpassen (z.B. Earliest Deadline First). Durch die individuelle Einsatzmöglichkeit von Echtzeitbetriebssystemen und deren geringen Platzbedarf gibt es hunderte solcher im Einsatz. Oft bestehen einfache Systeme nur im Abarbeiten von definierten Schleifen und Interrupts. Das Betriebssystem, auf das in dieser Arbeit Bezug genommen wird, ist ERIKA Enterprise [16] von Evidence. OSEK OS, COM und OIL werden hierbei unterstützt [4], [16], [17], [18].

2.4.1 Scheduling

Scheduling ist die Bezeichnung für die Ablaufreihenfolge von Tasks in einem System. Dabei werden begrenzte Ressourcen (z.B. Rechenzeit) auszuführenden Tasks zugeteilt. Um ein gutes Scheduling zu erreichen, müssen alle auszuführenden Aufgaben und deren Eigenschaften bekannt sein. Ein effizientes Scheduling zeichnet sich durch einen großen Durchsatz, effiziente Ressourcenverwendung, fairer Zuteilung und der Termineinhaltung aus. Kooperative Verfahren stellen die Ressourcen zur Verfügung und warten, bis diese wieder verfügbar ist. Um jedoch alle Deadlines zu gewährleisten, können auch präemptive Verfahren eingesetzt werden, die bereits zugeteilte Ressourcen wieder entziehen und anderen höher priorisierten Tasks zuteilen. Der ursprüngliche Task bekommt die Rechenzeit dann wieder vom Scheduler zugeteilt, wen dieser an der Reihe ist. Scheduling-Verfahren können, wie in Abbildung 5 dargestellt, eingeteilt werden:

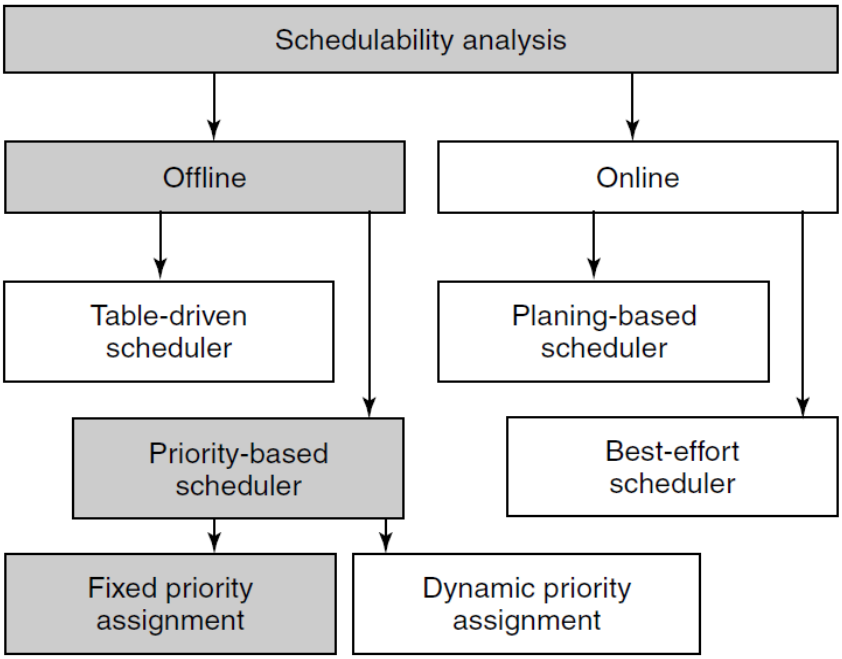


Abbildung 5 Übersicht Scheduling Analysis [31]

Es gibt zwei grundlegende Verfahren für Offline-Scheduling:

- Die auf die Auslastung der Prozessoren basierende Analyse
- Die Analyse der Antwortzeit jedes einzelnen Tasks.

Mittels eines Tests der Prozessorauslastung kann im Vorhinein eine Schedule-Tabelle erstellt und überprüft werden, wie hoch der Prozessor ausgelastet ist. Die Analyse der Tasks liefert eine WCET. Diese darf für den jeweiligen Task dann nicht über dessen Deadline liegen. Vorteil der Auslastungsanalyse ist eine schnelle “Machbarkeitsüberprüfung”, ob das

System das Scheduling fristgerecht umsetzt. Es liefert jedoch keine Details über die Umsetzung der jeweiligen Tasks. Die WCET Analyse für jeden Task kann sehr rechenintensiv sein, bringt jedoch einen guten Überblick für jeden Task und gegebenenfalls darüber, wie das Scheduling verbessert werden kann [40]. Unter [52] sind weitere Scheduling-Methoden für sicherheitskritische Systeme zu finden. Genauer wird auf das Scheduling-Verfahren noch in Kapitel 2.4.2 und 2.4.4 eingegangen.

2.4.2 Offline Scheduling

Um ein sicheres und effizientes Scheduling in echtzeitkritischen Systemen zu erreichen, sind einige Anforderungen an das System und deren Elemente zu stellen. Vor allem die WCET aller systemkritischen und sicherheitsrelevanten Tasks müssen bekannt sein. Dies ist auch eine große Herausforderung an Timing-Tools. Die Startreihenfolge und die Priorität der Task müssen schon zur Entwicklungszeit festgelegt werden. In OSEK Time erfolgt der Start mittels eines Dispatchers. Zu Beginn befinden sich alle Tasks im „suspended“ Zustand, siehe Abbildung 6).

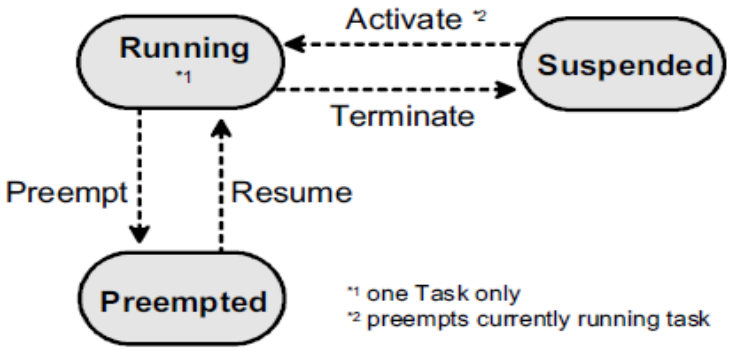


Abbildung 6 Task Status für Scheduler

Der Dispatcher in OSEK Time startet nur die Tasks, nimmt aber keine wirklichen Scheduling-Aufgaben wahr. Ist der Startzeitpunkt erreicht, wird der jeweilige Task gestartet und verdrängt die gerade laufenden Tasks (siehe Abbildung 7).

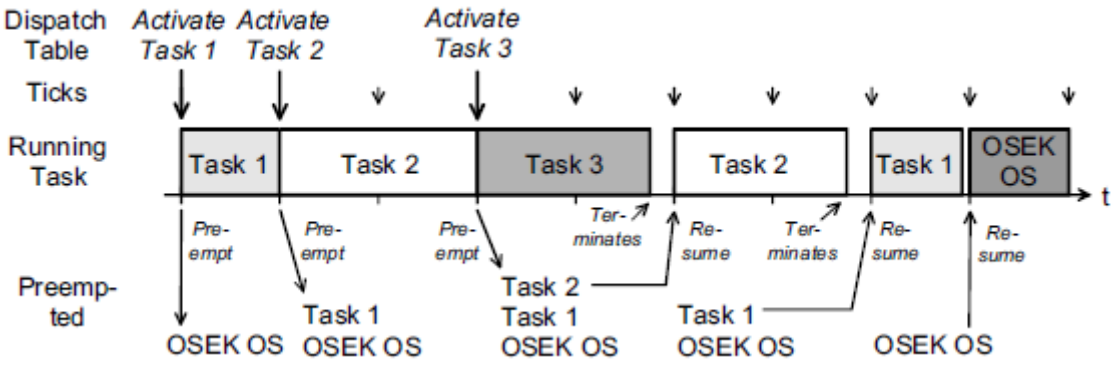


Abbildung 7 Ablaufreihenfolge

Der Verdrängte wird so in den „preempted" Status gesetzt. Hat ein gestarteter Task seine Ausführung beendet, wird der zuletzt beendete Task wieder aktiviert (Stack Based). In den Phasen, in denen kein OSEK Time Task läuft, kann das Betriebssystem seine Tasks ausführen, bis alle Tasks vom Dispatcher gestartet wurden und dieser sich beenden kann. Diese Grundidee entspricht auch der Idee der Schedule-Tabellen, die von der Initialisierung des OSEK OS gestartet werden.

2.4.3 Tasks

Ein oder auch mehrere Prozessoren können mehrere Aufgabenstellungen abarbeiten. Jede dieser Aufgaben, die im Multicore-Bereich auch parallel ausgeführt werden kann, wird in Folge als Task bezeichnet. Auf einem Prozessor wird ein Task nach dem anderen ausgeführt. Als Beispiel dient Abbildung 8, in der die Tasks *Task A „Zündung" Task B „Einspritzung" und Task C „Erfassung Pedalwert"* angeführt werden.

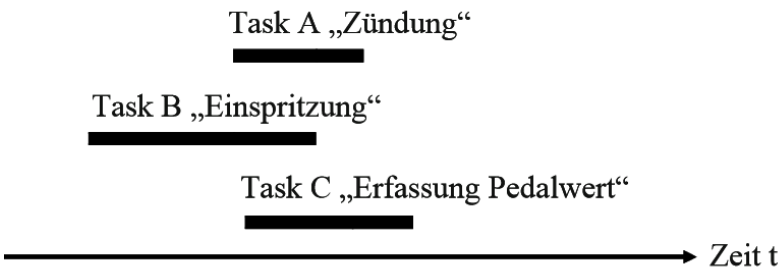


Abbildung 8 Beispiele für Tasks

Sollen, wie im angeführten Beispiel, mehrere Tasks auf einem Prozessor oder Core laufen, müssen die Tasks zeitlich aufgeteilt werden. Dies erfordert eine Aufteilung zu bestimmten Zeitpunkten, wie zum Beispiel in Abbildung 9 ersichtlich.

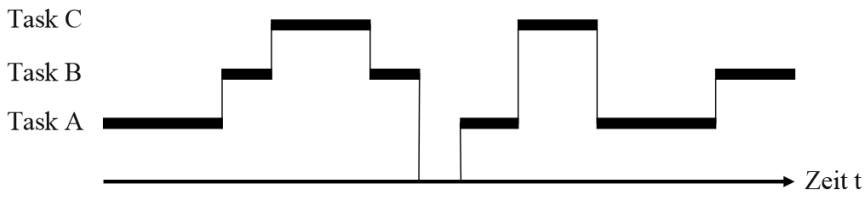


Abbildung 9 Aufteilung der Rechenzeit zu Tasks

Ist der Task dem Prozessor zur Ausführung zugeteilt, wird er als „running" bezeichnet. Weitere Zustände von Task sind unter 3.3 zu finden. Es kann pro Core immer nur ein Task im Zustand „running" sein. Die Aufteilung der Tasks auf die Rechenzeit verwaltet der Scheduler. Dieser benötigt für die Zuteilung die Priorität des Tasks, nach der der Task aus einer Warteschlange der Rechenzeit zugeteilt wird. Weitere wichtige Begriffe im Zusammenhang mit Tasks sind der Aktivierungszeitpunkt und die Deadline. Diese Punkte, die für die Echtzeitanforderung erforderlich sind, wurden bereits in Kapitel 2.1.2 erläutert [4], [6], [15].

2.4.4 Task Scheduling

Der Scheduler (Zeitplaner) bestimmt die Abarbeitungsreihenfolge von Tasks und kennt zwei grundlegende Arten von Prozess-Scheduling. Sie können präemptiv (unterbrechend) und non-präemptiv (nicht unterbrechend) sein. Wenn ein Task von einem nicht unterbrechendem Scheduling Rechenzeit zugeteilt bekommt, kann diese ihm nicht wieder „weggenommen" werden. Er muss sie „freiwillig" wieder freigeben, indem er entweder blockiert oder eben "fertig" ist. Bei unterbrechendem Scheduling wird jedem Task nur eine bestimmte Zeit zugeteilt und diese dem Task dann auch wieder „weggenommen".
OSEK kennt für Tasks drei Zustände:

- Running: Der Task wird gerade ausgeführt.
- Suspended: Der Task steht nicht zur Ausführung zur Verfügung. Er muss erst wieder explizit aktiviert werden
- Ready: Der Task ist bereit, ausgeführt zu werden. Seine Priorität entscheidet, wann er ausgeführt wird.

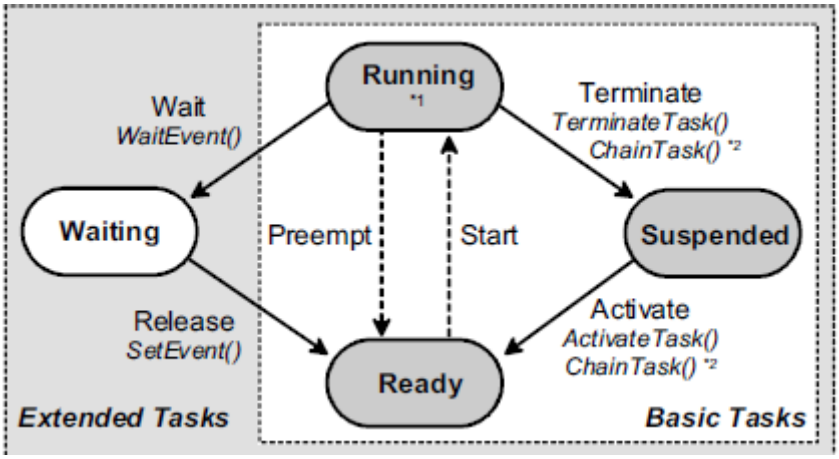


Abbildung 10 Zustände eines Tasks [14]

Es gibt auch noch erweiterte Tasks, die den Zustand „Waiting" besitzen. In Echtzeitsystemen kann es aufgrund der nötigen „harten Echtzeitfähigkeit" nur ein präemptives Scheduling geben, welches die Deadlines einhält. Weiters müssen nicht nur vordefinierte, bekannte Aufgaben abgearbeitet werden, sondern auch ereignisbasierte. Die Taskprioritäten werden statisch vor deren Ausführung festgelegt und können vom Benutzer nicht mehr geändert werden. Nur das Betriebssystem kann unter besonderen Bedingungen, zum Beispiel Zugriff auf Ressourcen, die Priorität kurzfristig für einen Task erhöhen. Tasks, die den Zustand „Ready" haben, werden nach ihren Prioritäten in einer FIFO-Warteschlange gruppiert. Wird ein Task von einem anderen unterbrochen, wird er wieder neu, als ältester Task, eingereiht. Wechselt ein Task von „waiting" in den „ready" Status, ist dieser der letzte der jeweiligen Prioritätenliste. Ein nicht unterbrechbarer Task kann nur in die Warteschlange neu aufgenommen werden, wenn dieser sich selbst beendet, auf ein anderes Event wartet oder die Ressource bzw. Rechenzeit von sich aus freigibt. Unterbrochene Tasks können von einem anderen Task aktiviert werden, durch die Freigabe einer Ressource oder durch Änderung des Status.

2.4.5 Runnables

Verschiedene Aufgaben mit gleichen Echtzeitanforderungen können entweder als eine Ansammlung von Tasks bezeichnet werden bzw. zu einem Task zusammengefasst werden. In diesem Fall werden einzelne Funktionen/Aufgaben als Runnables oder Prozesse bezeichnet, die durch ihre gemeinsame Echtzeitanforderung als Tasks zusammengesammelt werden. Diese einzelnen Prozesse oder Runnables werden dann meist in einer festgelegten Abarbeitungsreihenfolge statisch abgearbeitet [4], [6], [15].

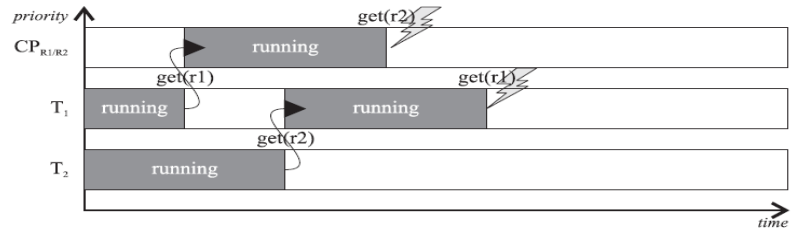
2.4.6 Ressourcen

Das Ressourcen-Management von OSEK verwaltet den Zugriff der Tasks auf gemeinsame Ressourcen. Hauptaufgabe ist die Behandlung von folgenden Punkten:

- Mutual Exclusion (Gegenseitiger Ausschluss; nur ein Task kann die Ressource halten)
- Priorität Inversion
- Verhindern von Deadlocks
- Nach Zugriff auf eine Ressource kann ein Task nie in den „waiting"-Status kommen.

Mittels des OSEK Priority Ceiling Protokolls [21] können die Probleme bzw. zu garantierenden Punkte umgesetzt werden. Deadlocks [55] sind auch eine andere Schwierigkeit, die bei Mehrkernprozessoren auftreten können [21]. Hierbei warten je zwei oder mehrere Task auf die Freigabe von Ressourcen, die der/die andere/n Task gerade blockieren, wie in Abbildung 11 zu sehen.

(a) Single-threaded: only one task running. Deadlocks are prevented by priority ceiling



(b) Multithreaded: two tasks sharing the processor can lead to a deadlock!

Abbildung 11 Ressourcenzugriff auf Single- und Multicore-Systemen mit Deadlock in (b) [21]

Das Problem der Prioritäteninversion ist unter Abbildung 12 zu sehen. Hierbei muss Task 3 unnötig lange warten, da Task 1 die Ressource hält und durch die höhere Priorität von Task 2 auch noch unterbrochen wird.

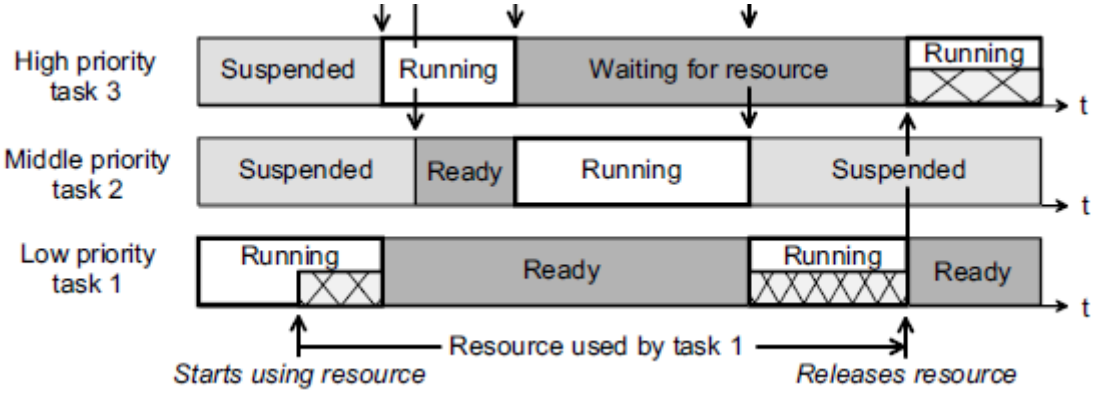


Abbildung 12 Prioritäteninversion [14]

Durch dynamische Prioritätenerhöhung kann dieses Problem, wie in Abbildung 13 ersichtlich, gelöst werden.

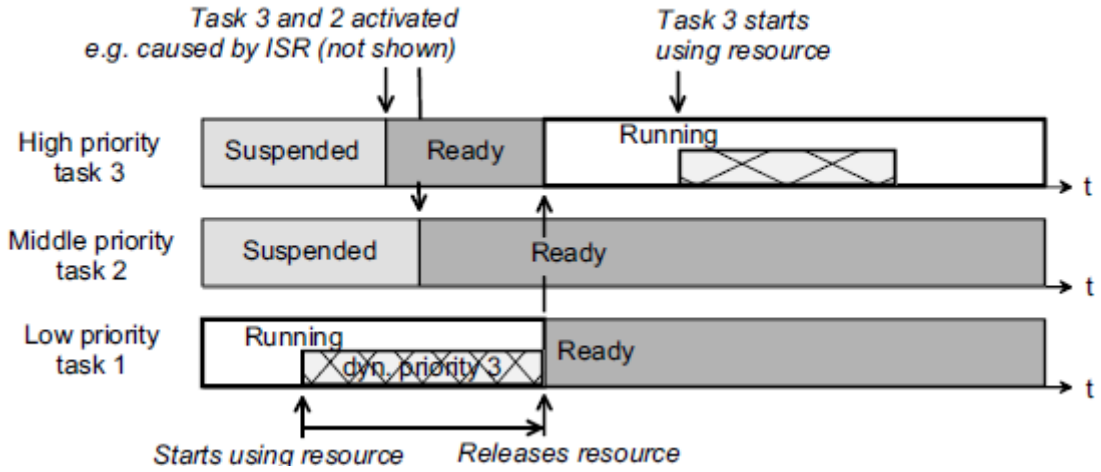


Abbildung 13 Prioritätserhöhung bei Ressourcenzugriff [14]

Hier wird die Priorität des Task 1 erhöht, solange er die Ressource verwendet. So kann Task 1 auf die Ressource zugreifen, bis er fertig ist. Danach wird die Priorität wieder verringert und Task 3 bekommt die Rechenzeit [4], [14]. Unter [21] sind weitere Lösungsvorschläge zu finden.

2.5 OSEK

OSEK ("**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug") ist ein Industriestandard, der von verschiedenen KFZ-Herstellern (u.a. BMW, Daimler-Benz, VW, Opel, Bosch, Siemens) und deren wichtigsten Zulieferern in einem Gremium im Jahre 1993 gegründet wurde. Ein Jahr später schloss man sich mit der französische **V**ehicle **D**istributed E**x**ecutive zur offiziellen Bezeichnung **OSEK/VDX zusammen.** Einer der wichtigsten

Standards von OSEK ist der das Echtzeit- Betriebssystem OSEK-OS. Weitere wichtige Standards, die im Gremium definiert wurden, sind:

- OIL (OSEK Implementation Language): Definition von den OS-Objekten wie Tasks, Ressourcen oder Alarme.
- ORTI (OSEK Runtime Interface): Vereinheitlicht das Debugging
- COM: Regelt die Kommunikation unter dem oder den Steuergerät(en).
- NM (Networkmanagement): regelt das Ein- und Ausschalten von Steuergeräten
- TIME: Funktionen von OSEK Time werden höher priorisiert als OS Tasks.

Aus diesen Standard (siehe Abbildung 14) wurden die ISO-Norm 17356 und weiterführend das AUTOSAR-Projekt gestartet. Gegliedert werden die Softwaremodule funktionsorientiert z.B. Fahrgeschwindigkeitsregler, komplexe Sensoren (z.B. von ESP) oder andere komplexe Funktionen und Anforderungen. Die Implementation in eigenen Softwaremodulen gewährt eine Unabhängigkeit von anderen Funktionen und erleichtert die Wiederverwendung und die Kommunikation mit anderen Komponenten. Dies gewährleistet eine Unabhängigkeit der Software in Bezug auf die Hardware. Der Datenaustausch wird über die RTE (Run Time Environment) geregelt. Die Basissoftware kann mit dem Betriebssystem eines gewöhnlichen PCs verglichen werden, wobei das eigentliche Betriebssystem das Taskmanagement bzw. die Rechenzeitverwaltung der CPU regelt. OSEK beinhaltet jedoch (noch) nicht weitere Betriebssystemaufgaben wie Kommunikation, Fehlerspeicherverwaltung oder die Schnittstelle zu Hardware. Abstrahiert wird die Hardware durch den HAL (Hardware Abstraction Layer) und ist somit von der Software abgekoppelt. Diese Aufteilung hat den Vorteil, dass bei Änderungen der Peripherie dieser Layer angepasst werden muss. Wo jedoch genau die Grenze zu der in der Basissoftware festgelegten Ein- und Ausgabefunktionen liegt, ist eine zentrale Aufgabe des AUTOSAR-Projekts [14], [15].
Durch den Integrierten Flash Loader ist es möglich, den Programm- und Datenspeicher auch im Nachhinein zu ändern, individuelle Anpassungen und Erweiterungen vorzunehmen.
Das zentrale Element von OSEK ist OSEK OS, das die Ressourcen verwaltet und als Multitasking-Echtzeitbetriebssystem die Tasks synchronisiert. Durch OSEK COM kann der Datenaustausch innerhalb eines oder auch mehrerer Steuergeräte erfolgen. Die Überwachung dieses externen Austausches erfolgt mittels OSEK NM (Network). Ein Merkmal der effizienten Umsetzung solcher oft sicherheitskritischen Systeme ist die statische Festlegung dieser Aufgaben. Die Konfiguration des Systems ist im später beschriebenen OIL (OSEK Implementation Language) File festgelegt. Durch diese Datei kann später mittels geeigneter Tools die Task Verwaltung erstellt werden [14], [15].

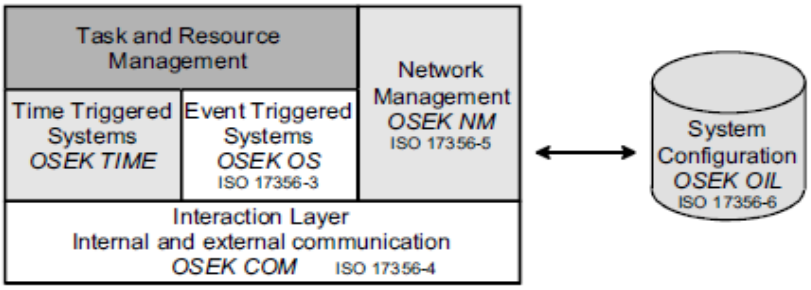


Abbildung 14 Grundkomponenten eines OSEK/VDX-Systems [14]

2.5.1 OIL-File

Um ein bestehendes System in SymTA/S abbilden zu können, gibt es die Möglichkeit, diese zu importieren. Da nicht alle Formate unterstützt werden, bietet SymTA/S auch die Möglichkeit an, mittels Python alle möglichen Formate einzulesen. Dies wird natürlich erleichtert, wenn eine einheitliche Formatierung oder ein Standard wie unter OIL vorliegt.
OIL steht für OSEK Implementation Language und ist ein OSEK Standard. In der OIL-Datei werden Einstellungen für das Betriebssystem und die Anwendungen festgelegt. Die vollständige Struktur der Datei und der Syntax kann unter der offiziellen OSEK-Seite [36] nachgeschlagen werden. In der, auf diese Arbeit bezogenen, Entwicklungsumgebung (RT-Druid bzw. Erika Enterprise wird auch die Konfiguration durch eine OIL-Datei vorgenommen (siehe Abbildung 15). Die Tasks, Alarms, Ressourcen usw. werden statisch vor dem Kompilierzeitpunkt festgelegt [36].

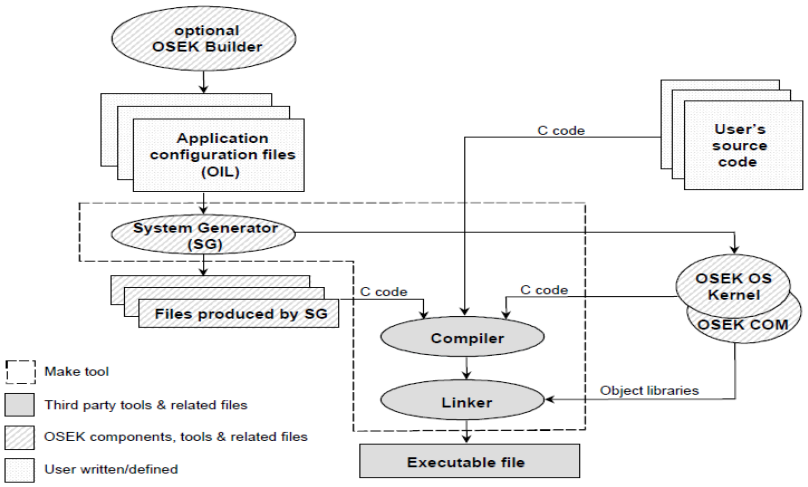


Abbildung 15 Beispiel für Softwareentwicklungsprozess

Die OIL-Datei wird von der Entwicklungsumgebung eingelesen. Für jede CPU ist eine eigene OIL-Datei notwendig. Die Datei besteht grundlegend aus zwei Teilen. Im einen

werden Definitionen und im anderen Deklarationen festgelegt. In den Definitionen werden Datentypen, Konstanten und Kernel-Objekte, die im Deklarationsteil für die Konfiguration eines spezifischen Kernel zur Verfügung gestellt werden müssen, festgelegt. Aufgebaut ist eine OIL-Datei dann wie folg, wobei nicht alle möglichen Elemente aufgelistet werden.
CPU ist ein Container, der die jeweiligen, im OIL definierten Objekte beinhaltet. Hierbei kann für jeden Core der CPU ein Betriebssystem definiert werden. In den Eigenschaften des Betriebssystems sind globale Eigenschaften enthalten, die sich auch auf die Einstellungen vom RTOS von Erika Enterprise auswirkt. Weitere wichtige Elemente eines OIL-Files sind:

- TASKS
- Resourcen
- Counter
- ALARM
- Event
- ISR
- COM.

Für Erika Enterprise können außerdem auch noch folgende Flags gesetzt werden, welche im Erika Enterprise Makefile benötigt werden:

- CFLAGS (Compiler-Optionen)
- ASFLAGS (Assembler Optionen
- LDFLAGS (Linker Parameter)
- LDDEPS (Bibliotheksabhängigkeiten für Makefile
- LIBS (Liste der zusätzlichen Bibliotheken).

Ein Beispiel für Deklarationen eines OIL-Files:

```
“CPU mySystem {
        OS myOs {
                EE_OPT = " DEBUG ";
                CPU_DATA = PIC30 {
                        APP_SRC = " code1.c";
                        APP_SRC = " code2.c";
                        MULTI_STACK = FALSE ;
                        ICD2 = TRUE;
                        };
                MCU_DATA = PIC30 {
                        MODEL = 33 FFJ256GP710;
```

```
                    };
                BOARD_DATA = EE_FLEX {
                    USELEDS = TRUE;
                    }
                };
        TASK myTask {
                PRIORITY = 1;
                STACK = SHARED ;
                SCHEDULE = FULL;
                };
        TASK myTask {
        PRIORITY = 1;
        STACK = SHARED ;
        SCHEDULE = FULL;
        };
}; “[37]
```

Wie bereits erwähnt, können die Objekte im Betriebssysteme (OS, myOS) für einen einzelnen Kern oder auch für ein Mehrkernsystem (auch einzeln) spezifiziert werden. Im Bereich MCU-Data wird die Konfiguration von den Peripheriegeräten (Speicher, Schnittstellen…) des Mikrocontrollers vorgenommen. Dazu kann die erweiterte Peripherie (Display, LEDs, Tasten….) des ganzen Bordes (z.B. Tricore) im BOARD_DATA Abschnitt des Betriebssystems definiert werden. Des Weiteren ist noch zu erwähnen, dass die Bibliotheken, die nicht von der Entwicklungsumgebung (Erika Enterprise) bereitgestellt werden, direkt im Linker Flag (LDFLAGS) hinzugefügt werden können. Der OIL-Eintrag eines Tasks könnte wie folgt ausschauen [nach 37]:

```
TASK TaskCpu0_MotorControl {
        CPU_ID          = "cpu0";
        PRIORITY        = 2;
        AUTOSTART   = TRUE;
        STACK           = PRIVATE { SYS_SIZE = 256; };
        SCHEDULE        = FULL;
        EVENT           = MCOnEvent;
        EVENT           = MCOffEvent;
        RESOURCE        = ResCpu0_MCEnable;
}
```

Die CPU_ID gibt an, welchem Core sie in diesem Fall zugeordnet ist. Die eigentliche CPU wird auf drei Core aufgeteilt. Die Priorität regelt, inwieweit der Task gegenüber anderen bevorzugt wird. Autostart auf TRUE bedeutet, dass der Task automatisch bei (Neu-) Start aktiviert wird. Der Stack ist exklusiv für den Task und hat eine definierte Größe (256). Schedule gibt an, wie der Scheduler mit dem Tasks umgeht und wie er ihn unterbrechen bzw. „einteilen" darf. Unter Event und Ressource wird ein Event ausgelöst, bzw. eine Ressource verwaltet.

2.6 AUTOSAR

AUTOSAR steht für AUTomotive Open System ARchitecture und ist ein Zusammenschluss von Automobil-, Steuergeräte- und Entwicklungswerkzeugherstellern bzw. anderen Zulieferern, wie die von Steuergeräten. Ziel dieser Partner ist es, einen gemeinsamen Standard zu definieren, um die Kommunikation der Software der verschiedenen Bauteile, vor allem von Steuergeräten, zu vereinheitlichen. Weiters ist durch die Vereinheitlichung eine Weiterverwendung und ein Austausch von Softwareelementen möglich. Durch diesen gemeinsamen Standard können mehrfache Neuentwicklungen von Softwarekomponenten von verschiedenen Herstellern reduziert und grundlegende Basisfunktionen geschaffen werden. Das Motto von AUTOSAR lautet: Zusammenarbeit bei Standards – Wettbewerb bei der Umsetzung [6]. Die Ziele von AUTOSAR sind:

- Standardisierung von Funktionen der Basissoftware von automobilen ECUs
- Skalierbarkeit für verschiedene Fahrzeug- und Plattformvarianten
- Übertragbarkeit der Software
- Unterstützung von verschiedenen funktionellen Domänen
- Definition einer offenen Architektur
- Die Zusammenarbeit zwischen den verschiedenen Partnern
- Entwicklung von hochverfügbaren Systemen
- Support der geltenden internationalen Kfz- Standards und state-of-the-art Technologien.

Durch diese gemeinsamen Ziele ist es für Unternehmen möglich, Steuergerätenetzwerke flexibler zu integrieren bzw. deren Funktionen einfacher austauschen und verschieben zu können. Die Unterstützung von sogenannter COTS (Commercial off-the-shelf) macht auch die Wartung der Steuergeräte über den ganzen Produktlebenszyklus möglich. Das funktionale Verhalten der Softwarekomponenten ist im AUTOSAR Standard jedoch nicht geregelt.

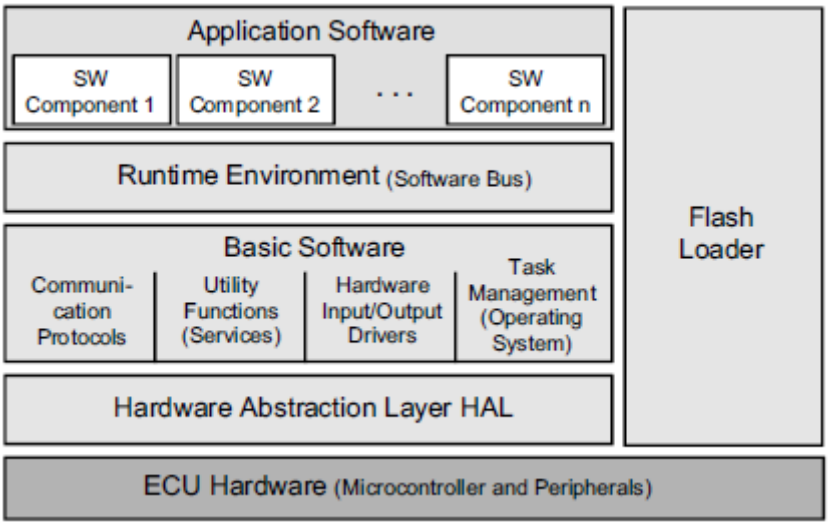


Abbildung 16 Grundaufbau von Softwarearchitektur [14]

In AUTOSAR wird die Steuergerätesoftware abstrahiert und in Basissoftware, Laufzeitumgebung und Anwendungsschicht, wie in Abbildung 16, unterteilt. Die Einteilung der Software erfolgt in verschiedene Layer wie z.B. dem Mikrocontroller Abstraction Layer (MCAL), der die Kommunikation, den Speicherzugriff bzw. die Ein. und Ausgabeschnittstellen (IO) definiert. Durch die Abstraktion und die Einteilung in Layer wie in Abbildung 17 abgebildet, ist die Software mikrocontrollerunabhängig und kann auf beliebigen Systemen verwendet werden. Dadurch wird die Kommunikation, der Speicherzugriff oder die Verwaltung der IO-Schnittstellen vereinfacht. Die Laufzeitumgebung ist für die Abstrahierung der Anwenderschicht der Basissoftware verantwortlich und stellt wiederum Dienste wie die Kommunikation für Anwendungssoftware bereit, die dann untereinander, oder mit anderen Diensten über die RTE kommunizieren [6].

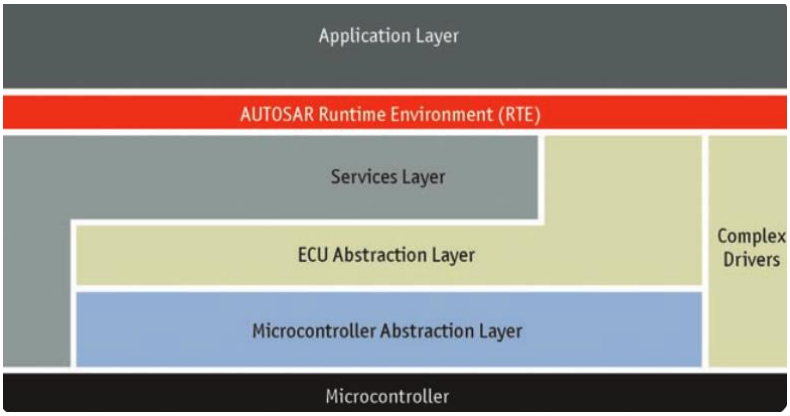


Abbildung 17 AUTOSAR-Schichtenmodell [64]

Wie bereit in Kapitel 2.5 OSEK beschrieben, sind die Schnittstellen zur BSW (Basissoftware) durch die Softwarekomponenten formal definiert. Die RTE steuert die Verbindung dieser Softwarekomponenten. Der Virtual Functional Bus (VFB) ist die Grundlage für die Kommunikation und damit wieder unabhängiger von der Hardware. Für konkrete Fahrzeuge wird die RTE und BSW konfiguriert und an das jeweilige Steuergerät angepasst. Diese Vorgehensweise kann zur besseren Aufteilung von Rechen- und Kommunikationslast sowie Speicherbedarfsoptimierung dienen. Die Definition der Schnittstellen (API) erfolgt mittels SWC Beschreibungen in XML. Diese abstrakte Strukturierung ermöglicht eine vorzeitige Implementierung, ohne den genauen Aufbau des Systems zu kennen. Eine weitere wichtige Spezifikation von AUTOSAR ist die des Metamodells (Templates) und der Methoden. Diese decken die Eingabe der Laufzeitumgebung (.arxml) oder auch die Dokumentation ab. Signale aus den Baugruppen Karosserie, Fahrgestell, Personenschutz, Antriebsstrang oder Komfort stellen einen weiteren Bereich der AUTOSAR Spezifikation dar [6].

2.7 Wiederverwendung von Software-Funktionen

Die oftmalige Weiterverwendung von bereits vorhandener Software ist nicht nur aus Kostengründen sinnvoll, sondern ermöglicht auch eine schnellere Entwicklung von neuen Fahrzeugen, die auf bereits vorhandenen Systemen aufbauen. Besonders im Motor und Getriebebereich hat diese Wiederverwendung Einfluss auf die Architektur der Elektronik. Dadurch sind einheitliche Getriebe und Motorsteuergeräte gefordert, die sich dann nur durch Daten und Programmstand unterscheiden. Weiters kann durch bereits verwendete Module das Risiko von Fehlern minimiert werden, da diese schon erfolgreich getestet sind. Hierbei ist wieder die Einhaltung der AUTOSAR Spezifikationen bei der Modellierung nötig. Für die Softwareentwicklung auf Quellcode-Ebene ist das von Vorteil, kann aber die Entwicklung von neuer Software, Funktionen und die Umstellung auf neue Hardware bremsen.

Daher ist eine Wiederverwendung von bereits vorhandenen Modellen günstiger. Die überprüften Spezifikationsmodelle der Funktionen sowie Design, Testfälle und Kalibrierdaten für die Simulation können vom Laborbetrieb bis zu den ersten Fahrversuchen zur Risikominimierung von Fehlern beitragen [4].

2.8 Verteilung von Software

Ein Problem, das es bei bereits vorhandener Software oder auch bei Neuentwicklungen für Multicore-Systeme zu lösen gilt, ist die optimale Aufteilung der Software-Komponenten auf die zur Verfügung stehenden Kerne. Softwarekomponenten konnten auf einem Kern noch gut optimiert werden um Zugriffe auf Ressourcen zu regeln. Werden diese jedoch auf verschiedenen Cores aufgeteilt, kann es zu drastischen Veränderungen des Zeitverhaltens kommen. Durch Interrupts, Antwortzeiten der Aufrufe und der Intercore-Kommunikation kommt es zu Einschränkungen der Verteilung. Auch Runnables, die auf gleiche Pointer zugreifen, sollten auf einem Core, bzw. in gemeinsamen Softwarekomponenten realisiert werden. Eine Lösung ist die Umstellung der Kommunikationsparadigmen.

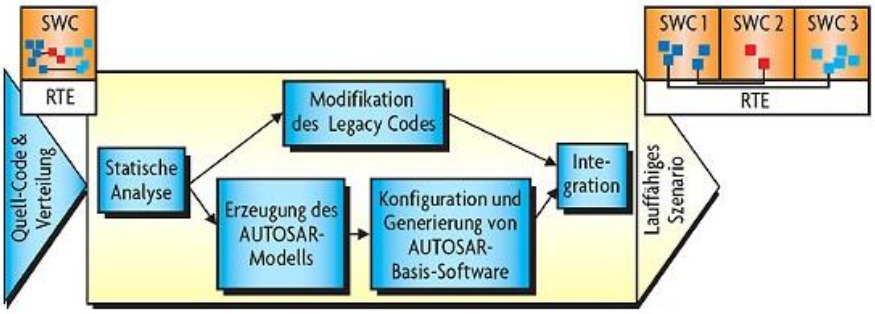


Abbildung 18 Zerlegung von Softwarekomponenten [32]

Durch die werkzeuggestützte Zerlegung der Software zu neuen kleineren Komponenten, siehe Abbildung 18, muss auch deren Kommunikation durch von AUTOSAR definierten Paradigmen umgesetzt werden. Um die Komponenten sinnvoll zu zerlegen, ist eine Analyse der Interaktion der Runnables-Gruppen nötig. Dadurch kann eine Übersicht über Aufrufe von Funktionen, Zugriff auf globale Variablen sowie Datentypen erreicht werden. Mittels z.B. Artop-Funktionen[66] kann ein AUTOSAR Systemmodell erstellt werden. Die modifizierten RTE-Aufrufe wurden so anstelle des Ursprungscodes eingesetzt. Garantien, die für das alte Single Core System galten, können für das neue Multicore-System nicht gegeben werden. Bei der Aufteilung der Software muss auf jeden Fall sehr flexibel reagiert werden. Laut [32] stehen folgende drei Strategien zur Verfügung:

- Freie Verteilung von Runnables
- Gruppierung von Runnables nach Datenfluss
- Barrier architecture.

Freie Verteilung von Runnables

Hierbei muss die Ausführungsreihenfolge von Runnables gleicher Priorität egal sein und nur ein schreibendes Runnables ist möglich. Besonders geeignet ist diese Verteilung, wenn viele Runnables auf gemeinsamen Daten arbeiten, die jedoch wenige miteinander kommunizierende Daten haben [32].

Gruppierung von Runnables nach Datenfluss

Voraussetzung für diese Strategie ist, dass nur eine Runnables-Gruppe als Schreiber jedes Datums fungiert. Die Abhängigkeiten der Runnables-Datenflüsse dürfen nicht zu groß werden, da ansonsten die Aufteilung schwierig wird. Die zu kommunizierenden Daten dürfen nicht zu groß werden, da ansonsten der Datenerhaltungs-Overhead zu groß wird [32].

Barrier architecture

Um die Runnables parallel ausführen zu können, sollte die Datenabhängigkeit möglichst gering sein. Die Lösung ist auf das Worst-Case-Szenario optimiert, was bei unregelmäßigen Laufzeiten große Wartezeiten verursacht [32].

Ist die Software erst einmal zerlegt, kann diese auf die Kerne aufgeteilt werden:
Echtzeitkritische- und sicherheitsrelevante Anforderungen haben in modernen Steuergerät-Systemen große Auswirkung auf die Aufteilung von Funktionen. Diese Aufteilung auf die verschiedenen Mikrocontroller muss auch unter Beachtung der Controller-Auslastung, des Kommunikations-Overheads und anderen Rahmenbedingungen wie der Beschränkung des Bauraumes vollzogen werden. Die verschiedenen Aufteilungsmöglichkeiten können in der Entwicklungsphase schon Kosten und Entwicklungszeit einsparen. Die Aufteilung der einzelnen Funktionen (Runnables) und Signalen kann z.B. wie in Abbildung 19 erfolgen.

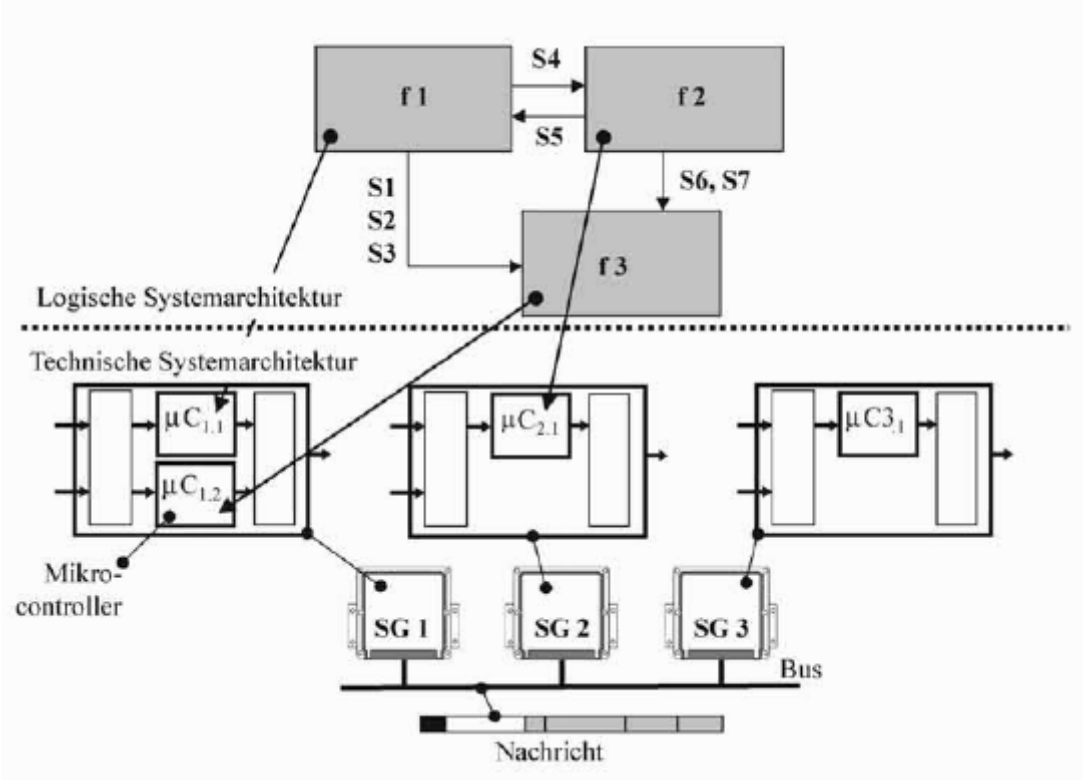


Abbildung 19 Aufteilungsmöglichkeiten auf Cores und Controller [4]

Ist die Zuordnung der Funktionen getroffen, müssen auch noch die Signale bzw. Nachrichten zugeteilt werden. Aus Sicht der Systemarchitektur sind die Nachrichteninhalte, die gebildet werden müssen, interessant, um diese für die jeweiligen Controller zur Verfügung zu stellen. Diese Nachrichten beinhalten dann mehrere verschiedene Signale, wie in Abbildung 20 zu sehen

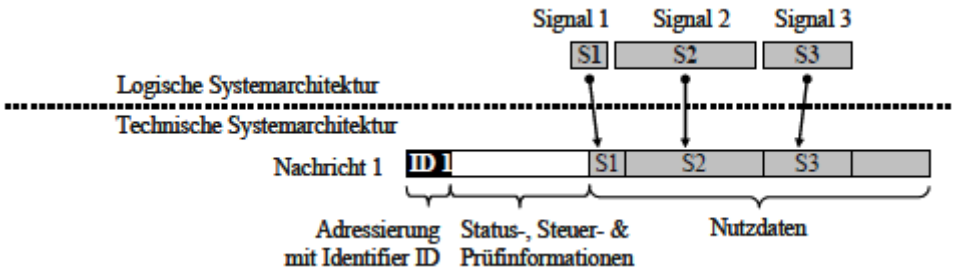


Abbildung 20 Aufteilung von Signalen in eine Nachricht [4]

Nach der Entwicklung und Aufteilung der Untersysteme erfolgt noch die stufenweise Integration in das elektronische System. Auch hier ist eine ständige Überprüfung der richtigen Aufteilung und Durchführung über die Systemgrenzen hinweg wichtig. Dieser Eingliederungsprozess kann durch Verfahren wie dem V-Modell [25] Software-Process-Improvement- and Capability-Determination (SPICE) [24], oder agilen Entwicklungs-

methoden [26], umgesetzt werden. Jedoch darf bei der Entwicklung nicht auf die Standards von AUTOSAR vergessen werden.

2.9 Gemeinsame Ressourcen

Bei gemeinsamen Ressourcen handelt es sich zum Beispiel um Speicher, Busse, Sensoren oder Aktuatoren, die von mehreren Tasks bzw. Funktionen verwendet werden. Besonders der Zugriff von verschiedenen Cores oder Steuergeräten auf diese Ressourcen und im Speziellen auf Aktuatoren stellt eine Herausforderung dar. In Echtzeitsystemen muss sichergestellt werden, dass die geforderten Elemente zu jedem Zeitpunkt dem richtigen Task zur Verfügung gestellt werden und keine unnötigen Blockier-Zeiten entstehen. Diese Blockaden können unerwünschte und unbekannte Auswirkungen auf andere Funktionen oder sogar Steuergeräte haben. Fehlender Freedom of Interference oder nicht vorhandene Datenkonsistenz können die Folge sein. Um diese Szenarien nicht aufkommen zu lassen, gibt es mehrere Möglichkeiten: Semaphore, um Ressourcen zu reservieren und freizugeben, Timing-Protection, um die Laufzeit eines Tasks zu begrenzen oder MUTEX, um die Datenkonsistenz zu gewährleisten. Durch “atomare”, also nicht teilbare, Funktionen kann der Zugriff auf Daten geregelt werden. [4][19][20]

Eine effektive Möglichkeit, gemeinsame Ressourcen zu verwalten, ist das Multiprocessor Priority Ceiling Protocol (MPCP). Hierbei wird zwischen Anwendungs- und Synchronisationsprozessoren unterschieden. Die Tasks werden dem Anwendungsprozessor statisch zugewiesen. Lokal kritische Sektoren werden auf dem Anwendungsprozessor gemäß den Anforderungen des Priority Ceiling Protocol (PCP) umgesetzt. Für einen global kritischen Bereich (global critical section GCS) wird der Synchronisationsprozessor verwendet. Verschachtelte Ressourcenzugriffe sind wegen der möglichen langen globalen Blockier-Zeiten aber nicht erlaubt. Die Multiprocessor Stack Resource Policy (MSRP), wie in [22] beschrieben, erweitert den Algorithmus von single-threaded-multiprocessors wie in [23] beschrieben. Dieses basiert auf einem prioritätsbegrenzenden Protokoll. Ähnlich wie beim MPCP, werden die Ressourcen in LCS und GCS unterteilt. Die GCS werden jedoch lokal am jeweiligen Prozessor ausgeführt. Das Verwenden von verschachtelten Ressourcenzugriffen ist auf Grund der Gefahr eines Deadlocks wiederum nicht gestattet. Dies stellt zwei Konzepte zur Umsetzung der Verwaltung von gemeinsamen Ressourcen dar [21].

Eine weitere Möglichkeit, geteilte Ressourcen zu verwalten, ist die der Mailboxen, welche von Compiler-Hersteller Hightec [46] verwendet wird. Der Datenaustausch von Tasks kann mithilfe von Mailboxen realisiert werden. Nachrichten (Anfrage an eine Ressource) müssen zugewiesen oder empfangen worden sein, bevor sie benutzt werden können. In dieser

Methode kann die Mailbox nicht übergangen werden. Übertragungen können von Task zu Task oder von Handler-to-Task erfolgen. Eine Aufgabe kann Meldungen und Ereignisse gleichzeitig erwarten. Nur der aktive Task hat Lese- und Schreibrechte, welche vom Besitzer der Nachricht gesperrt werden kann (Write Protection). Mailboxen haben folgende Eigenschaften:

- Mailboxen können eine beliebige Anzahl von Nachrichten erhalten
- Tasks senden Nachrichten zur Mailbox
- Tasks erhalten Nachrichten von der Mailbox
- Mailboxen sind als FIFO umgesetzt
- Die Umsetzung einer Priorisierung ist möglich
- Jeder Task hat eine private Mailbox.

Nachrichten besitzen die Attribute *Besitzer, Benutzer* und *Daten*. Daten können über zwei Möglichkeiten abgefragt werden: Direkte Abfrage mit Zurverfügungstellung des Speicherinhaltes oder Überprüfung, wenn bereits eine Kopie vorhanden ist. Empfangen werden Daten nur über die Mailbox. Dies kann eine private eigene sein, oder eine generelle öffentliche [45].

3. Design und Herausforderungen bei Umstellung auf ein Mehrkernbetriebssystem

In diesem Kapitel wird erörtert, welche Probleme es zu lösen gibt und welche Schritte, Elemente wie Tasks, Runnables und Ressourcen, und Tools dafür nötig sind. Als Entwicklungsumgebung soll eine freie kostenlose Plattform dienen. Die optimale Verwaltung dieser Elemente eines Betriebssystems (Task, Ressourcen, …) führen zu Herausforderung (Datenkonsistenz, optimales Scheduling, …) im Multicore-Bereich. Des Weiteren wird die Umsetzung eines statischen Scheduling betrachtet und die Optimierungsschritte, die das Analysetool SymTA/S durchführen kann. Die genaue Integration von SymTA/S wird in Kapitel 4 beschrieben.

3.1 Gründe einer Multicore-Umsetzung

Um eine möglichst optimale Umstellung von Single- auf Multicore-Systeme zu ermöglichen, gilt es, folgende Probleme zu behandeln:

- Single-ECU sind (temporär) überlastet und liefern nicht mehr die nötige Performanz
- Tasks können nicht immer rechtzeitig ausgeführt werden. Das Scheduling ist nicht ausreichend gut bzw. steht zu wenig CPU Zeit zur Verfügung
- Deadlines der Tasks werden nicht eingehalten und führen dadurch zu Fehlern im System
- Das Busnetzwerk ist (temporär) überlastet und kann die Nachrichtenpakete nicht rechtzeitig übertragen.
- Nachrichten kommen zu spät oder mit zu großer Verzögerung (Jitter) an
- Nachrichten gehen durch überladenen Buffer-Speicher verloren
- Fahrzeug-End-to-End-Funktionen können nicht rechtzeitig ausgeführt werden und deren Deadlines werden überschritten
- Die Stabilität von verteilten Steuergeräten wird beeinträchtigt.

Ein zeitliches Fehlverhalten bzw. Versäumen von Deadlines zieht nicht nur Folgen für die betroffenen Elemente nach sich, sondern kann in echtzeitkritischen Systemen für ein Versagen des gesamten Systems bzw. Fahrzeug sorgen. Zeitliches Fehlverhalten und nicht optimale Ausnützung machen sich auch auf andere Aspekte wie die Auswahl der Komponenten oder die Dimensionierung bemerkbar. Die zeitlichen Anforderungen der Funktionen müssen durch die Komponenten, auf denen sie durchgeführt werden, gewährleistet werden. Dafür ist für die Wahl der richtigen CPUs, Bus-Topologie und die

nötige Geschwindigkeit notwendig. Das beeinflusst die Zuverlässigkeit eines Systems. Andererseits bestimmen die zeitlichen Eigenschaften eines Systems auch die Performancereserve zum großen Teil. Diese kann für unerwartete auftretende Fälle genutzt werden oder für spätere Erweiterungen freigehalten werden. Am besten ist es daher, zeitliche Anforderungen von Anfang an zu berücksichtigen. Da die meisten Automobilhersteller auf das V-Modell zurückgreifen, sollte dieser Aspekt bereits in einem sehr frühen Zeitpunkt, aber auch während aller folgenden Schritte berücksichtigt werden.
Um die in Kapitel 2 erwähnten Standards und die damit entstehenden Herausforderungen zu bewältigen, wird in dieser Arbeit SymTA/S (***Sym***bolic ***T***iming ***A***nalysis) verwendet, um das Scheduling eines fiktiven Mehrkernechtzeitsystems zu analysieren und optimieren. Durch eine gutes Analyse und ein Optimierungstool kann der Zeitaufwand minimiert, durch optimale Integration die Stückzahl minimiert und die Qualitätssicherung garantiert werden. Durch die Gewährleistung eines stabilen Systems können auch noch Reserven für zukünftige Erweiterungen aufgedeckt und verwendet werden [41].

3.2 Analyse von Tools

3.2.1 GLIWA

GLIWA ist ein Timing-Werkzeug-Produzent, der es mit dem T1 möglich macht, vor allem im Automobilbereich Timing Analysen durchzuführen. Der Anwendungsbereich vom T1 reicht von Laufzeitmessungen, Überwachung der Laufzeitabsicherung, automatisierten Tests, Überprüfung der ISO 26262 [48] konformen Umsetzung, Sichtbarmachen von freien Ressourcen und die Untersuchung von Multi-Core-Datenflüssen. Umgesetzt wird die Analyse mittels einer Target-Software, die auf dem zu untersuchenden System umgesetzt und der Host-Software, die von einem externen Rechner gesteuert wird. Timing-relevante Informationen werden – wie in Abbildung 21 ersichtlich – von der Target-Software an die Host-Software übermittelt.

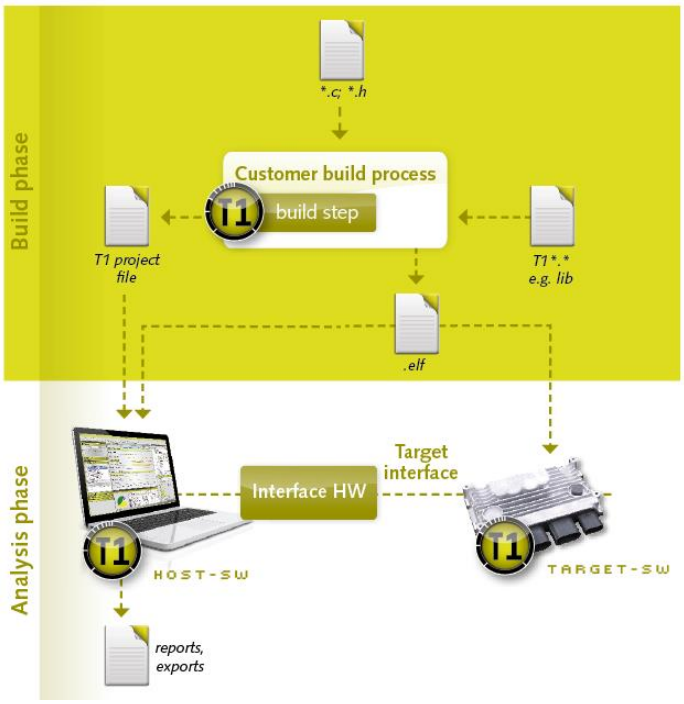


Abbildung 21 Übersicht Analyse-Software GLIWA T1 [49]

Die Effizienz liegt bei 0,2 bis 0,8% der CPU-Auslastung. CAN, JTAG, FlexRa oder Nexus können als Ziel-Interfaces eingesetzt werden. Am Zielsystem können direkt Deadline-Verletzungen analysiert und gegebenenfalls zusätzliche Informationen abgespeichert werden, um eine detaillierte Offline-Analyse durchzuführen. Diese Vorgehensweise ermöglicht auch das Aufspüren von komplexen Laufzeitfehlern. Timing Constraints, die optional erstellt werden können, ermöglichen ein Aufmerksam machen der definierten Grenzen. Durch verschiedene Methoden [49] kann man Timing-Absicherungen erreichen, die den Scheduling-Simulationen und statischen Scheduling-Analysen weit überlegen sind, da die Abweichungen dieser Modelle den Benutzer oft in einer falschen “Sicherheit” wiegen. Mittels T1.flex ist die Laufzeitmessung von Funktionen und Codeteilen möglich. Zum Hinzufügen von Delays kann manuell Last erzeugt werden, die das System auf unvorhersehbare Fehler testet und durch permanente Erhöhung des Delays auch die Timing-Grenzen aufzeigen kann. Mittels XML können über z.B. NET-Schnittstellen oder die Command-Line automatisierte Laufzeitmessungen angestoßen werden. Durch eine stetige Überwachung der Runnables aufgrund von Schwellwerten kann man schon früh eine Timing Vorhersage mittels T1 treffen. T1 bietet eine gute Möglichkeit, bereits existierende Software auf deren richtiges Laufzeitverhalten zu testen und gegebenenfalls auch schon in der Entwicklung, Optimierungen vorzunehmen (siehe Abbildung 22).

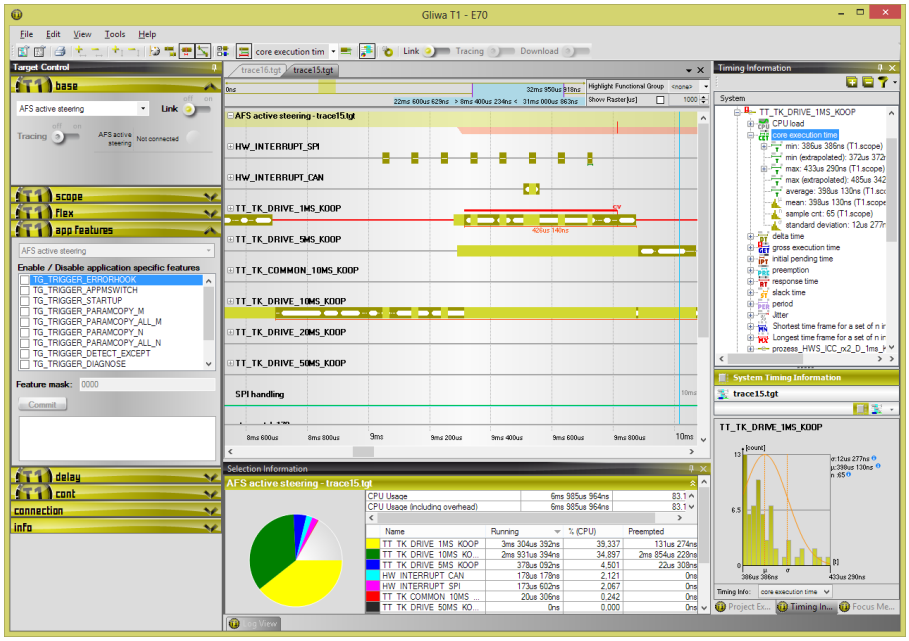


Abbildung 22 Visualisierung eines Echtzeit-Trace mittels GLIWA T1 [49]

3.2.2 PRECISION PRO

Precision Pro ist ein Tool zu Generierung von statischen Schedules. Sicherheitskritische, eingebettete Systeme sind die Hauptaufgabengebiete des Tools. Aufgrund unterschiedlicher Eingabeparameter wie Hardware, WCET, Periode oder andere Abhängigkeiten kann eine optimale Ressourcennutzung erreicht werden. Die Modellierung von konkreten Schedulings für komplexe Systeme und speziellen Ressourcen ist zusammen mit Ressourcenkonfliktvermeidung einer der Vorteile dieses Tools. Dadurch kommt z.B. eine Ressourcenüberlastung erst gar nicht zustande. In [51] wird das Problem von Interference Channels, welches mit PRECISION PRO im Vorhinein gelöst werden kann, genauer behandelt. Dadurch ist eine global synchronisierte Applikationsausführung auf einem Multi-Core-System garantiert. Die Simulation verschiedener Zeitkontingente für die Applikationen können schon im Entwicklungsprozess simuliert und optimiert werden. Speicherhierachien sind in PRECISION PRO noch nicht abgebildet und werden in den Ausführungszeiten der Tasks inkludiert. PRECISION PRO wurde vom Frauenhofer [50] Institut entwickelt [18].

3.2.3 Realtimenetwork

RTaW-ECU-ist ein Scheduling-Tool, welches die Task möglichst gleichmäßig über die verfügbare Zeit verteilt. Auf diese Weise kann mehr Performance aus den CPUs geholt werden. Ein weiterer Effekt ist es, die bessere Einschätzung der Task-Ausführungszeiten bzw. Aktivierungsmodelle zu erhalten. Eine visuelle Darstellung von mehreren hundert

Softwaremodulen ist möglich. Die aus den Ergebnissen gewonnen Informationen können verglichen und Strategien abgeleitet werden. RTaW-ECU hat folgende Features zu bieten:

- Bearbeitung von Single- und Multicore Systemen
- Keine Software Blackbox. Algorithmen sind in Papers veröffentlicht
- Einige Konfigurationsalgorithmen bieten eine Performanceoptimierung zugunsten minimaler CPU-Last
- Eine benutzerdefinierte Vorlage kann zur C-Code Generierung eingesetzt werden
- Benutzerspezifische Erweiterungen sind verfügbar.

3.2.4 Timing Architects

Timing Architects ist eine Toolgruppe, die es ermöglicht, mehrere Entwicklungsschritte in Many- und Multicore-Systemen umzusetzen. Verschiedene Tools wie:

- Designer: Spezifikation von Anforderungen, Design von Softwarearchitekturen, Mapping von Softwarefunktionen auf Hardware oder die Integration neuer Softwaremodule in bestehende Systeme
- Simulator: Bewertung der Systeme und Komponenten auf ihr Zeitverhalten, Auswertung des Hardwareressourcenverbrauchs, der Anwendungssoftware und des Betriebssystem, Performance-Analyse und Bewertung der unterschiedlichen Hardware-Plattformen und Software-Designs, Störungen und Ursache-Wirkungs-Analyse, Validierung und Vergleich von Designentscheidungen in einem frühen Entwicklungsstatus
- Optimizer: Halbautomatische Softwarearchitektur für Multicore-Systeme, Automatisches Mapping von Softwarefunktionen zu Hardwareressourcen, Machbarkeitsanalyse von Multicore Softwarearchitekturen und Hardwareplattformen
- Inspector: Verifikation der Systemumsetzungen auf der Zielhardware, Vergleich der Systemimplementierung und der simulierten Ergebnisse und Modelspezifikationen, Finden von Hardwareengstellen, Reverse-engineering von bereits vorhandenen Anwendungen.

Durch die Timing Architect Suite ist es möglich, AUTOSAR Projekte zu importieren. Dadurch ist eine Automatisierung der Entwicklungsprozesse realisierbar. Die Optimierung von Singlecore-Architekturen auf Multicore-Prozessoren ist eine weitere Aufgabe der Tool-Suite. Durch das Vergleichen von Softwareimplementierungsalternativen ist eine Optimierung der Leistung und der Echtzeiteigenschaften möglich.

3.2.5 ERNEST

Mit ERNEST (framework for the EaRly verification and validation of Networked Embedded SysTems) können Laufzeiten und Prioritäten von Software-Komponenten in vernetzten eingebetteten Systemen festgelegt und simuliert werden. Eine frühe Analyse von Softwarekomponenten im eingebetteten automotiven Bereich ermöglicht die Validierung von nicht-funktionalen Anforderungen. Die Unabhängigkeit von der Modellierungssprache der Systemarchitektur ist ein Hauptkriterium von ERNEST. Zurzeit werden EAST-ADL unterstützende Architekturen unterstützt. Spezifische Multicore-Charakteristika oder eine Optimierung sind aktuell nicht verfügbar. Durch die zukünftige Open Source Handhabung wird der Aufgabenbereich des Tools erweitert.

3.2.6 Symtavision

SymTA/S ist ein Model-basierendes Timing Analyse-Tool, welches auf die Optimierung und Verifizierung von Worst-Case Scheduling, und der Planung und Optimierung von harten Echtzeitanforderungen ausgelegt ist. Die Analyse und Planung der optimalen Verteilung der Prozessorzeiten und die Überprüfung der Zeitvorgaben sind weitere Fähigkeiten des Tools. Durch Szenario Analysen können spezifische und benutzerdefinierte Optimierungs- und Überprüfungsanalysen durchgeführt werden. Durch den Traceanalyser können reale Timing-Informationen, die von anderen Tools wie Vector CANanalyzer, Lauterbach TRACE32 oder Gliwa T1 kommen, mit den berechneten Werten verglichen werden. Durch verschiedene Optimierungsverfahren für Kommunikationsoverhead und Ressourcenverwaltung werden noch weitere nützliche Funktionalitäten ergänzt. Die Anbindung an andere Tools durch Python Scripting und ein Remote Interface macht die Tools für viele Toolchains interessant. Unterstützt werden alle gängigen Standards wie AUTOSAR, CAN, FlexRay, ERCOSek, RTA OSEK, Generic OSEK, …). Tools wie dSPACE SystemDesk, EB tresos Studio, ETAS i-SOLAR, IBM Rational Rhapsody, Vector PREEvision werden von SymTA/S unterstützt. Der Tranceanalyser unterstützt andere Tools wie Gliwa T1, Lauterbach TRACE32, Vector CANoe/ CANalyzer und andere Tools, die das CSV oder ASC Format unterstützen. Dadurch sind die Tools von Symtavision sehr universell und einfach mit anderen Tools einsetzbar.

3.3 Analysemöglichkeiten mit SymTA/S

Bei der Erstellung von offline Scheduling müssen die verschiedenen Periodenlängen, Abhängigkeiten und der Zugriff auf externe Ressourcen beachtet werden. Für einfache Single-Core-Prozessoren war das noch möglich, doch die komplexen Zusammenhänge von

Multicore-Systemen lassen diese Vorgehensmöglichkeit nicht mehr zu. Durch dieses statische Verfahren kann ein deterministisches Echtzeitverhalten erreicht werden und die Vorhersagbarkeit ist sehr hoch. Durch diese unflexible Methode entstehen auch Nachteile: Schwieriges Reagieren auf unvorhersehbare Ereignissen und geringe Adaptionsfähigkeit. Dazu gilt es die Punkte in den folgenden Kapiteln zu optimieren [14].

Im Analysetool SymTA/S, gibt es die Möglichkeit, Scheduling-Methoden zu wählen. Diese werden jeweils einem Betriebssystem zugeordnet. Welche Einstellungen getroffen werden können, werden in diesem Punkt erörtert [34].

GenericOSEK,

Tasks mit dieser Betriebssystem-Einstellung haben folgende Optionen:

- Präemptiv: Task sind unterbrechbar und haben einen Overhead
- Kooperativ: Tasks sind auf Runnable-Ebene unterbrechbar von anderen Tasks, die nicht die gleiche Unterbrechungsgruppe teilen. Auf der gleichen Ebene können Tasks nur zwischen 2 Runnables unterbrochen werden und haben auch einen Overhead.
- Non-präemptiv: Tasks, die nicht in der gleichen Präemption-Gruppe sind, können einander unterbrechen. Tasks der gleichen Gruppe können einander nur nach deren Beendigung unterbrechen. Diese Tasks haben einen Overhead.

ERCOSeK

ERCOSeK ist ein Betriebssystem, das vor allem in der Automobilindustrie eingesetzt wird. Es basiert auf dem OSEK OS Standard. Die Spezifikationen der Einstellung sind:

- HWTask: Der Task ist nicht unterbrechbar und hat keinen Overhead
- Kooperativ: Der Task ist unterbrechbar und hat einen Overhead. Die Priorität muss unter der Kernel Priorität sein.
- SWPräemptive: Die Tasks sind kooperativ und haben einen Overhead. Die Priorität muss kleiner als die des Kernels sein.
- Genauere Spezifikationen für ERCOSeK können unter [43] eingesehen werden.

RTAOSEK

RTA OSEK wurde von ETAS [54] entwickelt. Es basiert auch auf dem OSEK OS Standard. Die Spezifikationen der Einstellung sind:

- ISR_Cat1 (Interrupts der Category 1): Der Task ist nicht unterbrechbar und hat keinen Overhead
- ISR_Cat2: Der Task ist nicht unterbrechbar und hat einen Overhead
- Präemptiv: Der Task ist unterbrechbar und hat einen Overhead
- Kooperativ: Der Task ist auf Runnable-Ebene von Tasks einer anderen Präemptiv-Gruppe unterbrechbar. Tasks der gleichen Gruppe können einander nur nach beendeten Runnables unterbrechen. Der Task hat auch einen Overhead.
- Non-präemptiv: Der Task kann von Tasks einer anderen Präemptiv-Gruppe unterbrochen werden. Von Tasks der gleichen Gruppe kann der Task nur nach dessen fertigem Zeitslot unterbrochen werden. Der Task hat auch einen Overhead.

Genauere Spezifikationen für RTAOSEK können unter [44] eingesehen werden.

3.3.1 WCET

Die Worst Case Execution Time ist die Zeit, die als längst mögliche Zeit definiert wird, um die Ausführung eines Tasks zu gewährleisten. Die WCET ist ein nötiges Kriterium, um ein hartes Echtzeitsystem, wie bereits unter Kapitel 2.1.2 dargestellt, zu gewährleisten.
Simulationen, Tests und Messungen haben den Nachteil, dass nicht alle Fälle eines Szenarios abgedeckt werden. Die Abdeckung der flachen Randwerte kann durch eine Berechnung, wie sie in SymTA/S verwendet wird, herausgefunden werden. Die ist ein großer Vorteil, wodurch der gesamte Bereich vom Best-Case bis zum Worst-Case abgedeckt ist. Diese Spanne bzw. der Unterschied zur gemessenen Spanne ist in Abbildung 23 ersichtlich.

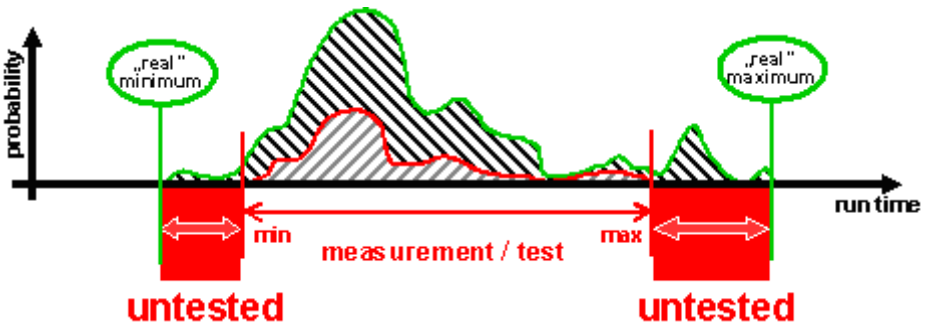


Abbildung 23 WCET Zeitspanne [41]

Scheduling-Berechnungen können diese Ergebnisse erreichen. Diese WCET-Berechnungen werden zwar in den meisten Fällen nicht auftreten, aber besonders für sicherheitskritische „harte" Echtzeitanforderungen, muss mit dieser Zahl gerechnet werden. Dies führt bei

Nutzung der WCET in allen Fällen zu einer zu konservativen und pessimistischen Bewertung und ist nicht in allen Fällen sinnvoll. Durch diese Analyse ist es schon zu frühen Entwurfszeitpunkten möglich, eine gute Ressourcenplanung zu erzielen (siehe Abbildung 24). Durch die schnelle Berechnungsmethode können durch SymTA/S mehrere Scheduling-Methoden erstellt und verglichen werden [41].

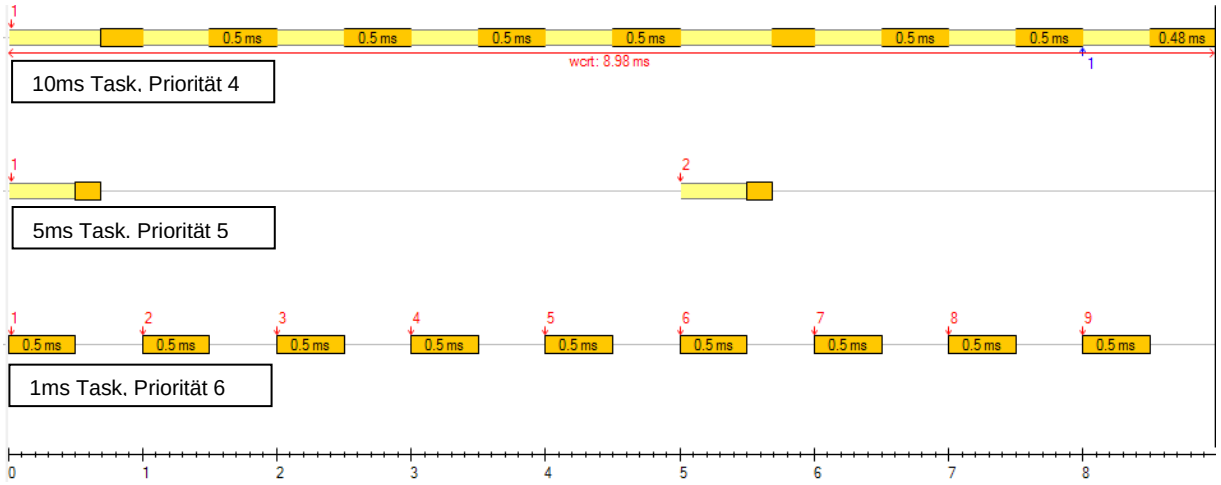


Abbildung 24 WCET-Analyse

3.3.2 Datenkonsistenz

In Multi-Core-Systemen haben die Kerne zwar ihren eigenen Cache, teilen sich jedoch auch gemeinsamen Speicher. Durch die eigenen Cache-Speicher wird ermöglicht, dass die neuesten Daten verwendet werden. Dies stellt jedoch im Multi-Core Bereich gleichzeitig eine der größten Herausforderungen dar. Die Änderung von Speicherinhalten auf einem Kern müssen auch auf die anderen (Cache-)Speicher übertragen werden. Dies kann über „Cache Coherency“ Module realisiert werden. Bei der Umsetzung von Multi-Core Systemen mit SMP (symmetrischen Multiprozessorsystemen) können alle Prozessoren durch gemeinsamen Adressraum auf den Speicher zugreifen. Bei AMP (asymmetrischen Multiprozessorsystemen) Systemen werden der Speicher und dessen Adressraum aufgeteilt. Bei dem moderneren NUMA- (Non-Uniform Memory Access) Systemen werden die Speicher in lokale und globale geteilt. Um die „richtigen“ Daten, also diejenigen, die benötigt werden, im Speicher zu haben, gibt es mehrere Möglichkeiten. Dabei muss auch Augenmerk auf die Trefferrate der richtigen Daten gelegt werden, da davon die Zugriffszeit abhängt. Mittels „Disabling“ werden die Cache Speicher deaktiviert und der Zugriff erfolgt nur über den gemeinsamen Speicher, was dann natürlich auch Zeit kostet, aber eine einfachere Verwaltung der Daten mit sich bringt. Mittels „Lock“ werden bestimmte Teile des

gemeinsamen Caches für bestimme Software reserviert. Dies hat den Nachteil, dass weniger Speicher für alle zur Verfügung steht, bringt jedoch eine Verbesserung der Performance. Mittels „Flushing“ ist es möglich, mit leerem Cache zu starten. Jeder kann auf den Speicher zugreifen und bekommt so etwas von den Bereichen ab. Der Wechsel dieser Daten sollte nicht oft erfolgen. Mittels „Partitioning“ werden Segmente den Cores zugeteilt. So profitieren alles Kerne vom Speicher und der Determinismus wird dadurch erhöht. Jedoch ist dies Variante, vor allem auf mehr Kernen, schwer umsetzbar. Die Strategien werden unter [56], [57] und [58] genauer beschrieben [56], [57], [58]. [59].

3.3.3 Inter-Core Kommunikation

Die Herausforderung bei der Verteilung der Software-Komponenten auf die verschiedenen Cores ist, die Rechenzeit effektiv zu nutzen und dass die Zeitanforderungen der SWC erfüllt werden. Der Overhead durch Intercore-Kommunikation soll dabei möglichst gering gehalten werden. Dazu ist oft eine sinnvolle Zerlegung der Softwarekomponenten nötig. Die Zerlegung muss aber gewisse Kriterien erfüllen, wie z.B. das Beibehalten der SWC am gleichen Core, wenn es eine gemeinsame Pointer-Variable gibt. Auch bei zu geringer Modellierungstiefe kann die Granularität der einzelnen Softwarekomponenten nicht ausreichen, um ein Aufteilung zu erreichen, die sinnvoll und effizient ist. Diese Zerlegung kann auch schon mittels Zerlegungsszenarien, z.B. mit Matlab Simulink, umgesetzt werden. Ein Verfahren, das angewendet werden kann, um die Intercore-Kommunikation zu minimieren und die Software möglichst gut aufzuteilen, basiert auf dem Graph-Cut-Problem.

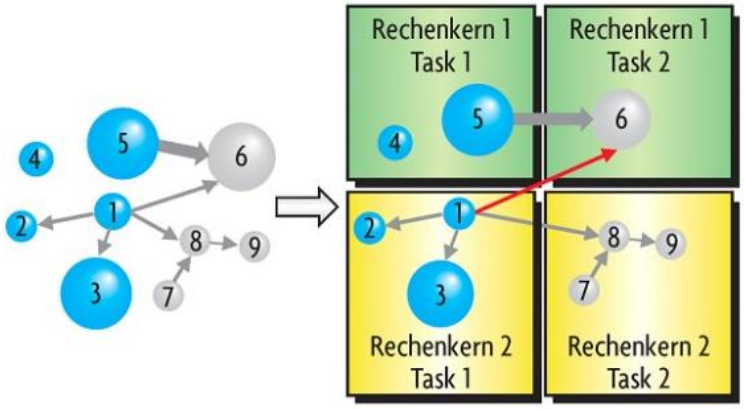


Abbildung 25 Graph-Cut Problem [32]

Hierbei werden die Knoten so aufgeteilt, dass die Knoten möglichst gerecht auf die Rechenkerne aufgeteilt sind, und die Kanten, bzw. deren Gewichte zwischen den Knoten auf verschiedenen Cores möglichst gering sind.

In Abbildung 25 sind die Runnables mit ihren WC-Laufzeiten als gewichteten Knoten, die aufgeteilt werden, zu sehen. Gewichtet nach der Zugriffshäufigkeit stellen die Kanten gemeinsame Datenobjekte, Zugriffe bzw. Abhängigkeiten dar. Um gemeinsame Pointer in den Komponenten beizubehalten, wurden diese Kantengewichte hoch gesetzt, um eine Aufteilung zu vermeiden [32].

3.3.4 Load Balancing

In Multicore-Systemen werden die bekannten OS-Applikationen statisch (zur Systemintegration) auf die unterschiedlichen Rechenkerne aufgeteilt. Tasks, Interrupts, Alarme, Schedule Tables, Counter und Ressourcen werden zu Gruppen zusammengefasst. Die Kommunikation zwischen den Softwarekomponenten wird transparent von der RTE umgesetzt. Durch die Intercore-Kommunikation ist es möglich, Tasks auf anderen Kernen zu aktivieren oder mit diesen zu kommunizieren. Die Kommunikation verlagert sich vom Bus auf die Intercore-Kommunikation, welche bessere Werte in der Latenzzeit, dem Durchsatz und auch der Sicherheit aufweist. Der begrenzende Faktor ist die gemeinsam genutzte Peripherie wie Speicher oder I/O. Complex Device Drivers (CDDs) können direkt auf die Basissoftware oder das OS zugreifen. Die exklusive Speichernutzung kann durch Spinlocks [55] realisiert werden. Jedoch können bei zu schwacher Absicherung auch Deadlocks [55] auftreten.

Die Herausforderung bei der Verteilung der Softwarekomponenten auf die verschiedenen Cores ist, die Rechenzeit effektiv zu nutzen und die Zeitanforderungen der SWC zu erfüllen (siehe Abbildung 26).

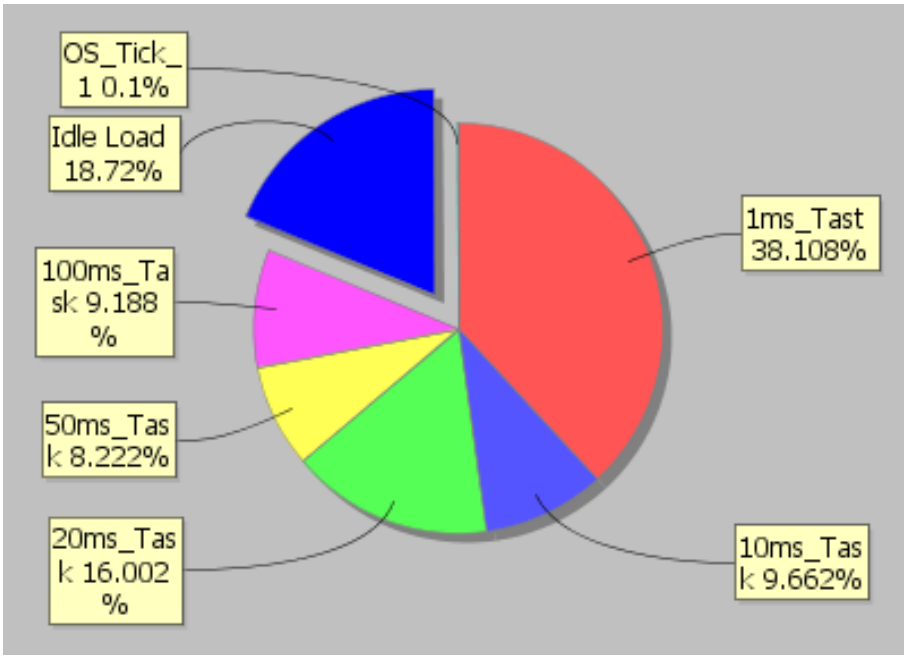


Abbildung 26 Auslastung eines Kernes

Der Overhead durch Intercore-Kommunikation soll dabei möglichst gering gehalten werden. Dazu ist oft eine sinnvolle Zerlegung der SWC nötig. Die Zerlegung muss aber gewisse Kriterien erfüllen, wie z.B. das Beibehalten der SWC am gleichen Core, wenn es eine gemeinsame Pointer-Variable gibt [32]. In SymTA/S gibt es durch die Möglichkeit, die Last der Kerne zu simulieren und Optimierung vorzunehmen. Diese Optimierung kann durch besseres Aufteilen der Task auf die Cores (siehe Abbildung 27) erfolgen oder durch die Optimierung der Intercore-Kommunikation, bzw. des Kommunikations-Overheads (siehe Abbildung 28).

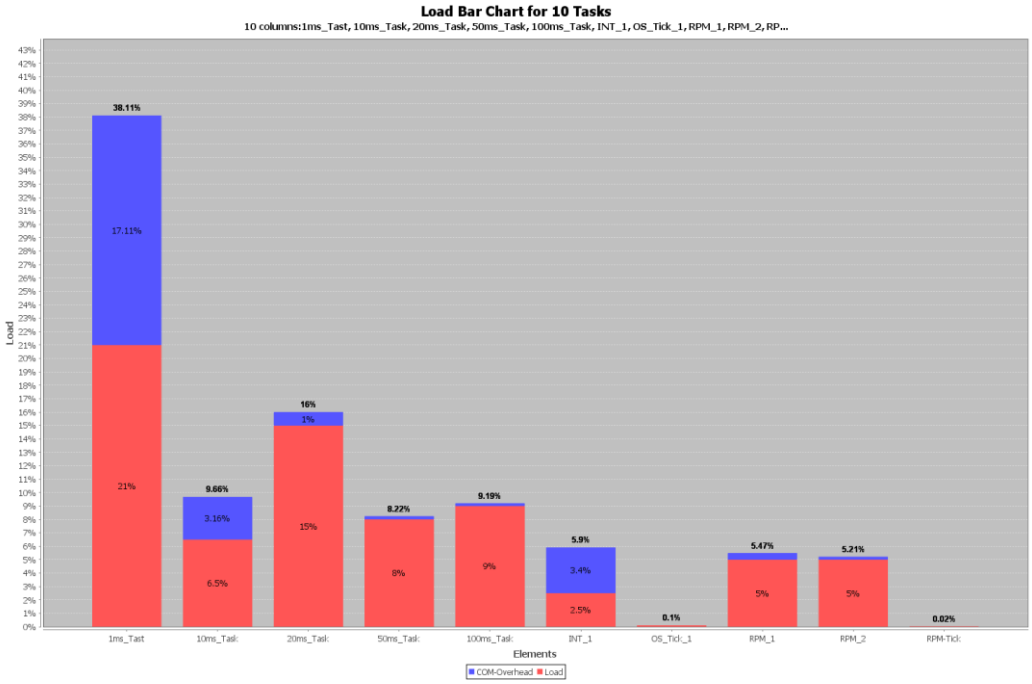


Abbildung 27 Overhead der Tasks

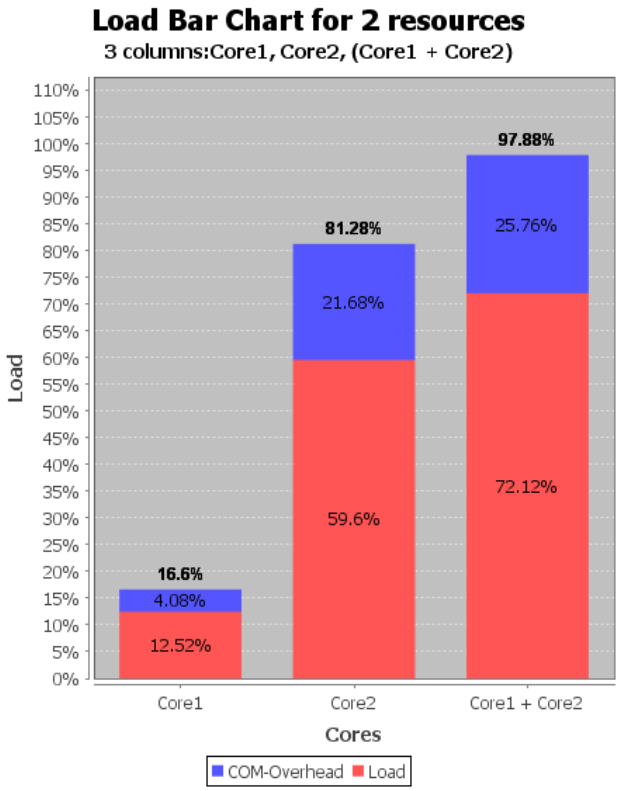


Abbildung 28 Kommunikations-Overhead

3.4 Integration von SymTA/S in eine Automotive-Toolchain

Um die Umsetzung von Single- zu Multicore-Systemen zu verwirklichen, sind einige Aspekte zu beachten. Besonderes Augenmerk wird auf das verwendete Timingtool und dessen Einstellungsparameter gelegt. SymTA/S bietet die Möglichkeit, sich in andere Toolchains zu integrieren. Dies ist ein sehr wichtiger Aspekt, der bei der Umstellung auf ein Multicore-System beachtet werden muss. Ansonsten müssen die gesamten Entwicklungsschritte neu überdacht werden bzw. entstehen unnötige Kosten für Anpassungstools, um bestehende Systeme für die Entwicklung und Analyse nutzen zu können. Die Integrierung des Analysetools stellt ein wichtiges Kriterium bei der Auswahl dar. Die nötigen Optionen und Einstellungen für die Konfiguration werden in den folgenden Punkten erläutert.

3.4.1 Erika RT Druid

In einem Echtzeitsystem spielt das Betriebssystem eine besonders große Rolle. Es muss die Echtzeitfähigkeit gewährleisten und die Ressourcen und Tasks des Systems verwalten. Laufzeiten und Deadlines dürfen nicht überschritten werden. Für die Umsetzung des E-Gas-Systems wurde Erika Enterprise [16] verwendet. Erika Enterprise ist aus einer Initiative von OSEK/VDX entstanden. Auch die verschiedenen Scheduling-Methoden werden in Erika Enterprise umgesetzt. Für die Umsetzung mittels des OSEK-Konzepts wurde das Betriebssystem auch OSEK OS, OSEK COM und OSEK OIL konform zertifiziert. Um die nötigen Schritte für die Analyse umzusetzen, sollten zuvor einige Fakten beachtet werden. Die Konfiguration von Erika Enterprise wird mittels des Plugin RT Druid umgesetzt. Wie bereits unter Kapitel 2.5.1 beschrieben, können die einzelnen Betriebssystemkomponenten mittels des OIL-Files konfiguriert werden. RT-Druid wandelt die Informationen im OIL-File in einen zu verwendbaren Quelltext um. Die Zuteilung der Komponenten, z.B. der Tasks oder Ressourcen zu einem Core, erfolgt mittels des OIL-Files. Durch die Umsetzung der Unterstützung im Multicore Bereich sind auch dessen Herausforderungen mittels dieses Betriebssystems zu lösen. Erika Enterprise mit RT Druid verspricht auch eine einfache Umstellung von Single-Core-Systemen zu Multicore-Systemen. Durch Anpassung des OIL-Files können mit geringem Aufwand bereits vorhandene Software und deren Funktionen auf mehrere Kerne aufgeteilt werden.

Zusammengefasst bietet Erika Enterprise folgende Features:

- „OSEK/VDX certified
- Real-time kernel, priority based, with stack sharing for RAM optimization
- Minimal multithreading RTOS interface
- RTOS API for: Tasks, Events, Alarms, Resources, Application modes, Semaphores, Error handling
- Support for conformance classes (FP, BCC1, BCC2, ECC1, ECC2, EDF, FRSH) to match different application requirements
- Support for preemptive and non-preemptive multitasking
- Support for fixed priority scheduling and Preemption Thresholds
- Support for Earliest Deadline First (EDF) scheduling
- Support for stack sharing techniques, and one-shot task model to reduce the overall stack usage
- Support for shared resources
- Support for periodic activations using Alarms
- Support for centralized Error Handling
- Support for hook functions before and after each context switch
- GPL with Linking Exception license” [42].

4. Implementierung

Die Umsetzung der Simulation eines echtzeitkritischen Systems wurde mit dem Analysetool SymTA/S umgesetzt. In dieser Arbeit wird der Fokus auf die Lösung der Probleme gelegt, die bei der Umstellung eines Single- auf ein Multicore-System entstehen. Besonderes Augenmerk wird auf die WCET-Analyse, die Reservezeiten und Optimierungsschritte gelegt und wie man auf Probleme bei Ressourcenzugriff reagieren kann. Die Randbetrachtung der WCET und System Distribution steht im Mittelpunkt. Aufgrund fehlender realistischer Szenarien und Daten wird auf die Trace Analyse nicht eingegangen. Die Verknüpfung und Integration in die vorhandene Toolchain und die Nutzung der darin enthaltenen Informationen ist Teil dieses Kapitels. Es werden verschiedene Möglichkeiten aufgezeigt, wie SymTA/S auf die Toolchain bzw. dessen bereits vorhandenen Informationen zugreift (siehe Abbildung 29).

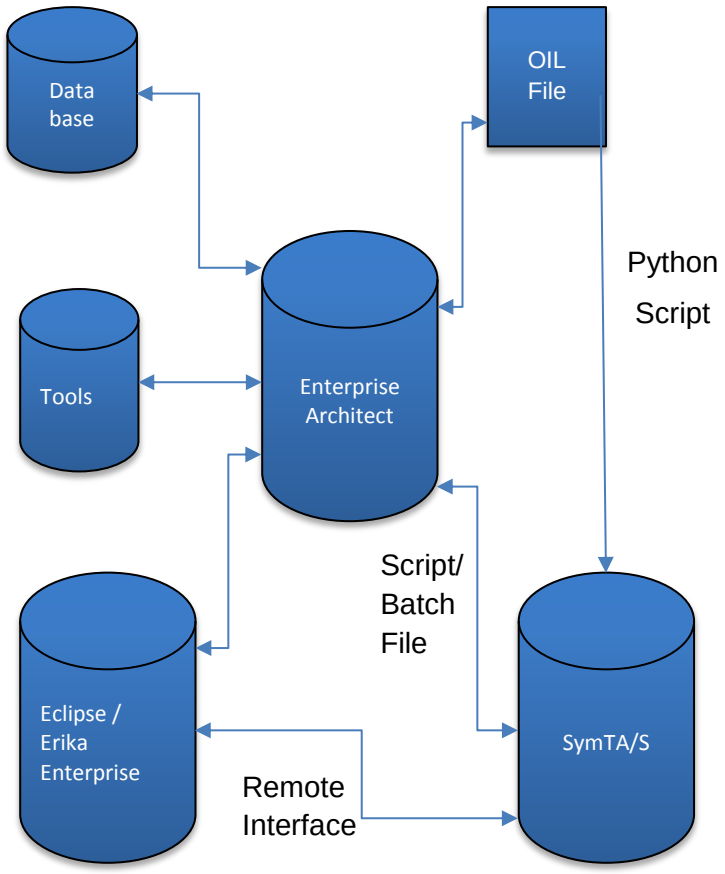


Abbildung 29 Toolchain Aufbau

4.1 Modelbausteine in SymTA/S

SymTA/S bietet eine sehr große Anzahl von Einstellungsmöglichkeiten, die teilweise sehr komplex ineinander verschachtelt sind. Deshalb wird in dieser Arbeit besonders auf die Konfigurationen im Zusammenhang mit Multicore-Systemen geachtet. Wichtig ist bei der Erstellung neuer Systeme, dass dessen Elemente richtig miteinander verknüpft und konfiguriert werden. In Abbildung 30 sind alle simulierbaren Elemente abgebildet, wobei im Genaueren nur auf Core, Task, Runnable und Trigger eingegangen wird.

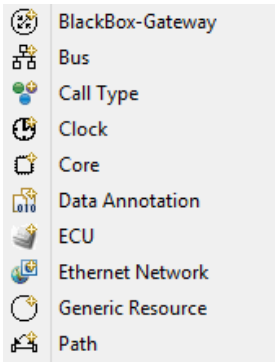


Abbildung 30 Elemente von SymTA/S

Für die Betrachtungen in dieser Arbeit wird ein System wie in Abbildung 31 ersichtlich, aufgebaut.

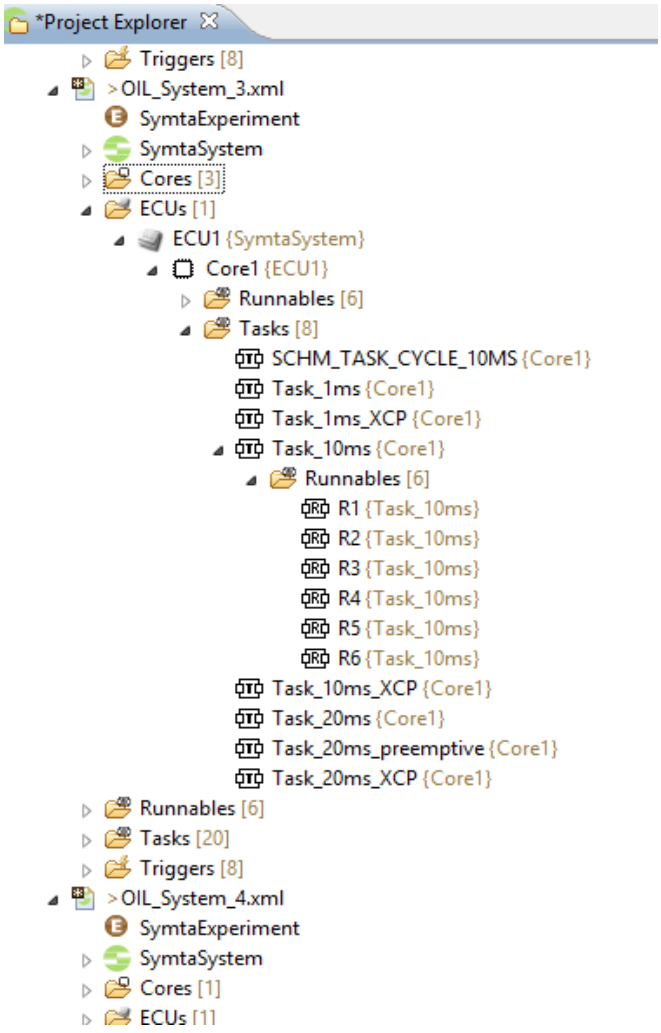


Abbildung 31 Aufbau eines Multicore Systems

Die Analysen sind auf ein Steuergerät bezogen, welches drei Kerne besitzt. Die Hierarchie sieht wie folgt aus: Mit dem simulierenden System werden die ECU verknüpft, auf diese folgen die Kerne, welche wiederum die Elemente Task und Speicher besitzen. Ein Task kann auch noch in Runnables aufgeteilt werden. Speicher halten noch Variablen, auf die von Tasks zugegriffen wird. Die Verbindungen der Tasks mit Variablen, die den gemeinsamen Speicherzugriff wiedergeben, ist in Abbildung 32 ersichtlich. Genauere und weitere Einstellungen und Elemente, die in einem System erstellt werden können, sind im Manual der Software zu finden.

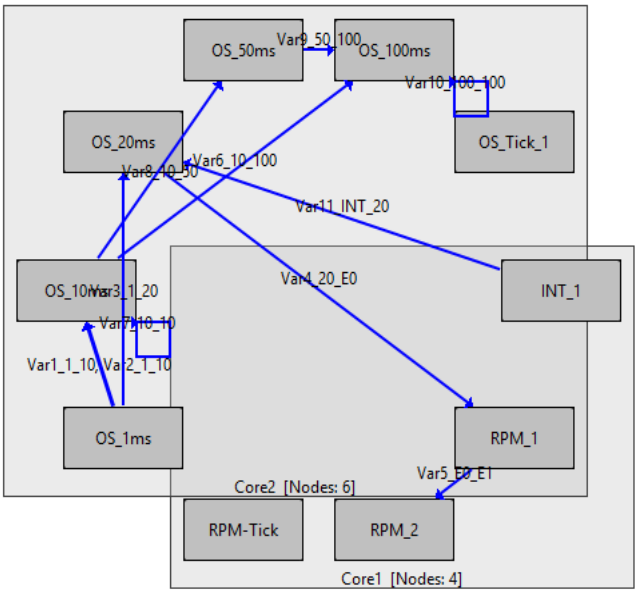


Abbildung 32 Task Graph im Multicore-System [27]

4.1.1 Core

Für jeden Core kann eines der beschriebenen Scheduling (AutosarOS, ERCOSeK, GenericOSEK und RTAOSEK) ausgewählt werden. Angaben zur OSEK-Analyse regeln, ob die Tasks voll präemptiv sind oder nicht. Für das Betriebssystem werden die Einstellungen bezüglich des Geschwindigkeitsfaktors auf die anderen Kerne oder die Kernel-Priorität geregelt. Um die Auslastung der Kerne zu regulieren, kann die Last dieser mit einer oberen und unteren Schranke geregelt werden.

4.1.2 Task

Folgende Einstellungen können für einen Task, der die wichtigste Rolle in den Scheduling-Analysen spielt, vorgenommen werden. Generische OSEK Task sind wie folgt definiert:

- Präemptive: Der Task kann unterbrochen werden und hat auch einen Overhead
- Cooperative: Der Task kann von anderen Tasks unterbrochen werden, wenn diese nicht die gleiche Präemption-Gruppe teilen. Tasks der gleichen Gruppe können nur auf Runnable-Ebene zwischen eines Runnable-Wechsels unterbrochen werden. Die Tasks haben ebenso einen Overhead

- Non-präemptive: Der Task kann (auf Tasklevel Ebene) von anderen Tasks, die nicht der gleichen preemption Gruppe angehören, unterbrochen werden. Tasks der gleichen Gruppe können einander nur zwischen eines Taskwechsels unterbrochen werden. Die Tasks haben ebenso einen Overhead

Eine weitere Einstellung für Betriebssysteme im Element Task ist ERCOSeK. ERCOSeK wird vor allem im Automobile Sektor verwendet. Es basiert auf dem OSEK OS-Standard. Die genauere Beschreibung lautet wie folgt:

- HWTask: Der Task kann nicht unterbrochen werden und hat auch keinen Overhead
- Cooperative: Der Task hat einen Overhead und kann unterbrochen werden. Die Priorität muss unter der Kernelpriorität liegen.
- SWPräemptive: Der Task ist kooperativ und hat einen Overhead. Die Priorität muss unter der Kernelpriorität liegen.

RTAOSEK wurde von ETAS [65] entwickelt und basiert auch auf dem OSEK-Betriebssystemstandard. Folgende Merkmale weißen die Tasks auf:

- ISR_Cat1 (Interrupts der Kategorie 1): Der Task ist nicht unterbrechbar und hat keinen Overhead
- ISR_Cat2 (Interrupts der Kategorie 2): Der Task ist nicht unterbrechbar und hat einen Overhead
- Präemptive: Der Task kann unterbrochen werden und hat einen Overhead
- Cooperative: Der Task ist (auf Runnable Ebene) von anderen Task unterbrechbar, wenn sie nicht in der gleichen präemptiv-Gruppe sind. Tasks der gleichen Gruppe können einander nur zwischen zwei Runnables unterbrechen. Die Tasks haben einen Overhead
- Non- präemptive: Der Task kann (auf Taskebene) von anderen Tasks, die nicht die gleiche präemptiv-Gruppe teilen, unterbrochen werden. Tasks der gleichen Gruppe können einander nur zwischen zwei Task-Ausführungen unterbrechen. Die Tasks haben einen Overhead

AUTOSAR OS ist ein internationaler Standard der AUTOSAR-Organisation. Es wird versucht, einen offenen Standard für den Automotiv-/Elektronik-Bereich zu etablieren.

- ISR_Cat1 ((Interrupts der Kategorie 1): Der Task ist nicht unterbrechbar und hat keinen Overhead
- ISR_Cat2 (Interrupts der Kategorie 2): Der Task ist nicht unterbrechbar und hat einen Overhead
- Präemptive: Der Task kann unterbrochen werden und hat einen Overhead

- Cooperative: Der Task ist (auf Runnable Ebene) von anderen Task unterbrechbar, wenn sie nicht in der gleichen präemptiv-Gruppe sind. Tasks der gleichen Gruppe können einander nur zwischen zwei Runnables unterbrechen. Die Tasks haben einen Overhead
- Non-preemptive: Der Task kann (auf Taskebene) von anderen Tasks, die nicht die gleiche präemptiv-Gruppe teilen, unterbrochen werden. Tasks der gleichen Gruppe können einander nur zwischen zwei Tasks-Ausführungen unterbrechen. Die Tasks haben einen Overhead

Um ein gutes Offline-Scheduling erzielen zu können, muss so viel Information wie möglich von den Tasks zur Verfügung stehen. Folgende weitere beeinflussende Parameter können eingestellt werden:

- Activation Overhead: Gibt dem Betriebssystem Overhead für die Aktivierung an
- Termination Overhead: Gibt dem Betriebssystem Overhead für die Determinierung an
- Priority: Muss eine Integer-Zahl sein. Hohe Werte bedeuten hohe Priorität.
- Non Preemption Group: Falls Cooperatives oder nicht unterbrechendes Scheduling gewählt ist, muss die „None Preemption Group" definiert werden, um ein korrektes Scheduling-Verhalten zu erzielen.
- Blocking Time: Gibt die maximale Zeit an, die der Task blockiert werden kann, z.B. beim Warten auf eine externe Ressource.

Zur Synchronisation können die Tasks in Synchronisationsgruppen eingeteilt werden und gelten dann auch für die Analyse als synchronisiert. Diese können dann auch mit einem Offset versehen werden, welcher eine Verzögerung beim Start, bzw. bei der erstmaligen Aktivierung des Tasks bewirkt. Verschiedene Analyseeinstellungen wie das Ausschalten des Offsets oder der WCET Analyse können vorgenommen werden. Weitere Punkte sind:

- Min./Max. Total Load: Hier wird bei einer Verletzung der Grenzen ein Fehler aufgezeigt. Kontrolliert wird die Einhaltung der Last durch die WC-Analyse, die Distribution Run Analyse oder die Trace Import Analyse.
- Min./Max. Core Execution Time: Hier gilt das gleiche wie bei einer Last über oder Unterschreitung.
- Min./Max. Response Time: Wird die Antwortzeit überschritten, wird wieder ein Fehler gemeldet. Analysiert wird dieser Wert von der Distribution Run Analyse oder der Trace Import Analyse.

- Min./Max. Activation Distance: Wird die Aktivierungsdistanz unter- oder überschritten, wird wieder ein Fehler gemeldet. Analysiert wird dieser Wert von der Distribution Run Analyse oder der Trace Import Analyse.
- Min./Max. Start Distance Werden die Startzeiten eines Events unter- oder überschritten, wird wieder ein Fehler gemeldet. Analysiert wird dieser Wert von der Distribution Run Analyse oder der Trace Import Analyse.

4.1.3 Runnable

Runnables können in Tasks zusammengefasst werden. Müssen bestimmte Funktionen alle Millisekunden ausgeführt werden, können diese zu einem 1ms-Task zusammengefasst werden. Weiters sollte beim Zusammenfügen von Runnables auf deren Zugriffsvariablen geachtet werden. Folgende Einstellungen können für Runnables getroffen werden:

- Repetition Factor: Gibt an, ob der Runnable bei jeder Taskausführung ausgeführt wird oder nur jeder zweiten oder dritten oder mehr.
- Period: Gibt den Intervall des Tasks an. Wird mit dem Repetition Faktor und dem Intervall des Tasks gegenverglichen.
- Order Execution: Gibt die Ausführungsreihenfolge der Runnables in einem Task an.
- Read Access: Ist eine Liste von Variablen, auf die lesend zugegriffen wird. In dieser Liste können Variablen auch mehrmals stehen, wenn auf diese öfters zugegriffen wird.
- Write Access: Ist eine Liste von Variablen, auf die schreibend zugegriffen wird. In dieser Liste können Variablen auch mehrmals stehen, wenn auf diese öfters zugegriffen wird.
- Core Execution Time: Die reine Ausführungszeit, ohne Unterbrechungen. Wird als min und max. Wert angegeben.

Constraint Checks können in den Runnables für die System Distribution und Imported Trace Analyser durchgeführt werden. Hier gibt es wieder ähnlich zum Task, folgende Elemente:

- Min./Max. Core Execution Time
- Min./Max. Effective Execution Time
- Min./Max. Response Time

4.1.4 Memory

Um in SymTA/S für ein Multicore-System eine Memory-Einheit zu simulieren, ist es nötig, gewisse Parameter für den Kommunikations-Overhead einzugeben. Diese sind nötig, um ein realistisches Zugreifen auf die Speichereinheit zu simulieren und einen etwaigen Engpass

bzw. “Flaschenhals” beim gemeinsamen Speicherzugriff aufzuzeigen. Folgende Werte können definiert werden.

- Call Type: Auf welche Weise wurde der Wert ermittelt (RTE/Gemessen)
- Access Type: Für welche Zugriffsart gilt die Verzögerung (Read/Write)
- Instruction Core: Definiert den zugreifenden Core
- Data Memory: Definiert, um welchen Speicher es sich handelt
- Size [bit]: Definiert die Größe des betroffenen Wertes
- Duration: Definiert den Bereich, in dem die Verzögerung auftritt.

4.1.5 Trigger

Trigger werden zur Aktivierung von verschiedenen Elementen, z.B. eines Tasks, verwendet. Ein Synonym, das in diesem Zusammenhang verwendet werden kann, ist „chaining”. Wichtig ist hierbei, dass ein Trigger interne Aktivierungseinstellungen überschreibt. Diese Einstellungen in z.B. den Tasks geben einfach das nächste zu aktivierende Element an. Weitere Einstellungen, die für einen Trigger gesetzt werden können, sind:

- Caller: Dieses Element ruft diesen Task bei dessen Beendigung auf (siehe Abbildung 33).
- Trigger Event Model: Alternative Aktivierung, falls der Trigger nicht richtig eingestellt ist oder ein Problem auftaucht.
- Trigger Offset: Offset, falls der Trigger nicht optimal umgesetzt wurde.
- Repetition Factor: Definiert den down-sampling Faktor. Gibt an, ob Aktivierung bei jeder Ausführung durchgeführt wird oder nur jeder zweiten oder dritten oder mehr.
- Base Cycle: Definiert die Verzögerung, der erneuten Ausführung, wenn der Repetition Factor genutzt wird.

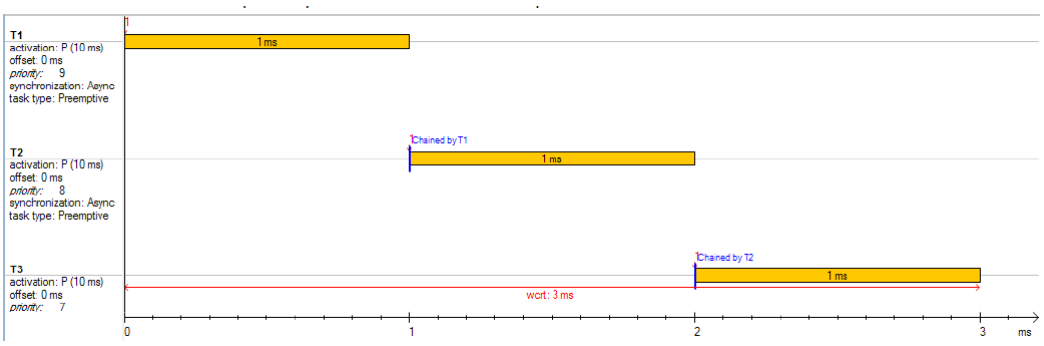


Abbildung 33 Interne Aktivierung [27]

Für die Worst Case Analyse muss noch darauf geachtet werden, dass das Trigger Event Model, die Synchronisation und der Offset erforderlich sind, falls die Aktivierung über mehr als zwei verschiedene Ressourcen geht. Weiters kann man das Event händisch oder automatisch bearbeiten.

4.2 Standard Schnittstellen von SymTA/S

Um die Erstellung von bereits vorhandenen Systemen im Analysetool zu vereinfachen, werden folgende Formate unterstützt, die durch einen Importer geladen werden können:

- FIBEX 2.0.1 und FIBEX 3.1.0
- DBC oder Zwischenformate von CAN ermöglichen den Import von Vector CANoe DBC Daten
- CAN, OSEK und FlexRay files
- LDF (LIN Deskription File)
- SymTA/S 1.x
- AUTOSAR
- Communication Overhead.

4.3 OIL-Import via Python-Skript

In SymTA/S können auch externe Files mittels eines Python-Skriptes eingelesen werden. Hierbei muss eigentlich nur die Struktur und der Syntax der Datei bekannt sein, damit Informationen eingelesen werden können und ein System erstellt wird. Die OIL-Files zur Konfiguration der CPUs können noch nicht automatisiert eingelesen werden. Mittels dieser Methode kann jedoch jede beliebige Quelle, also auch OIL-Files, eingelesen und aus den gewonnenen Informationen ein System erstellt werden. Die Informationen, die über eine OIL File gewonnen werden können, sind in Kapitel 2.5.1 beschrieben. Zeitverhalten von Task

(CET, Offset) oder Zugriffszeiten auf Variablen sind im OIL-File nicht vorhanden und müssen über andere Quellen (z.B. Excel-File) eingelesen werden. Wie in Abbildung 34 ersichtlich, ist es ratsam, den Datenfluss in eine Richtung zu beschränken. Durch das Einlesen der OIL-File Informationen werden keine Daten manipuliert. Es werden lediglich Daten aus der Datei gelesen. Diese OIL Datei muss jedoch schon vorhanden sein, bzw. von einem anderen Tool wie Enterprise Architect oder RT Druid zur Verfügung gestellt werden. Werden Änderungen im System vorgenommen, z.B. Verbesserungen nach der Analyse von SymTA/S, sollten diese in dem Tool vorgenommen werden, in welchem das OIL-File erstellt wurde oder dieses Verwaltet. z.B. eine Datenbank. Da vor allem das OIL-File von mehreren Tools oder Entwicklungsumgebungen verwendet werden kann, wird nicht empfohlen, dieses direkt zu ändern.

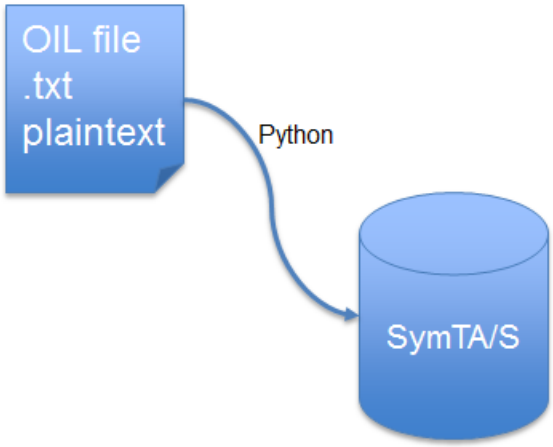


Abbildung 34 Einlesen eines OIL-Files mittels Python

Als Quelldatei kann jedes beliebige Dateiformat (.txt, .import .xlsx) gewählt werden, solange die relevanten Informationen enthalten sind. Diese eingelesenen Informationen sollen zuerst in eine Struktur gebracht werden, z.B. Liste von Tasks mit den jeweiligen Eigenschaften (siehe Kapitel 4.1), und können dann weiterverarbeitet werden, um ein Projekt zu generieren. Mittels folgender Syntax können in SymTA/S Ausgaben, wie in Abbildung 35 ersichtlich, getätigt werden:

- SYMTAOUT.status(String statusMessage)
- SYMTAOUT.warning(String warningMessage)
- SYMTAOUT.error(String errorMessage).

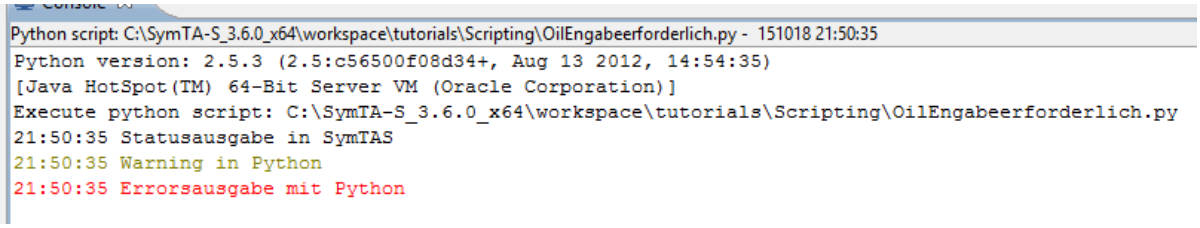


Abbildung 35 Verschiedene Konsolenausgaben in SymTA/S

Ein Beispiel, wie die Syntax für die Erstellung von Elementen aussieht, kann nachfolgend betrachtet werden. Hier wir ein neues Projekt, System, ECU, Core, dessen Einstellungen und die Verlinkung der Elemente, beispielhaft angegeben. Ein genaueres Beispiel zur automatischen Erstellung eines Systems aus einem OIL-File ist im Anhang zu finden.

```
# create project
SVWorkbench.createProject("Oil_File_Project")
# create a system in the given project
SVSystem = SVWorkbench.createSystem("Oil_File_Project", "OIL_System")
#create CPU without naming it
#create ECU (in this Case only 1 ECU is created)
ecu = SVSystem.createElement("ECU", None)
# link ECU to SymtaSystem
ecu.linkToParent(SVSystem)
# create core
core = SVSystem.createElement("Core", None)
# map core to ECU
core.linkToParent(ecu)
# set core properties
core.setProperty("scheduler", "GenericOSEK")
SYMTAOUT.log("core scheduler: " + core.getProperty("scheduler"))
core.setProperty("speedFactor", "%d" % 2)
SYMTAOUT.log("core speedFactor: " + core.getProperty("speedFactor"))
core.setProperty("kernelPriority", "%d" % 16)
SYMTAOUT.log("core kernelPriority: " + core.getProperty("kernelPriority"))
core.setProperty("executionBuffer", "%d" % 1)
SYMTAOUT.log("core executionBuffer: " + core.getProperty("executionBuffer"))
core.setProperty("extendedAnalysisResult", "All")
SYMTAOUT.log("core extendedAnalysisResult: " + core.getProperty("extendedAnalysisResult"))
…
```

Im Beispiel, das im Anhang zu finden ist, wurden zur Erstellung der Datenstruktur “Regular Expressions” [55] verwendet.

4.4 Import mittels Batch

Um SymTA/S automatisch mit dem Einlesen eines Files, zum Beispiel OIL-File (siehe Kapitel 4.3), zu starten, kann man dies über die Eingabe in einer Konsole erreichen. Vereinfacht wird dieser Vorgang durch Umsetzung mittels eines Batch Files. So muss nicht der ganze Befehl in die Konsole eingegeben werden, sondern nur die auszuführende Datei, die dann das Skript und die nötigen Befehle ausführt. Batch-Files, auch Skripte genannt, können Routinen abarbeiten oder sich wiederholende Befehle automatisieren. Diese Skripte können dann auch von allen Tools gestartet werden, wie es z.B. in Enterprise Architect möglich ist Beinhalten kann eine Batch-Datei verschiedene Befehle, die aber nicht formatiert werden. Die Endung der Datei, bzw. die Dateinamenerweiterung erfolgt mittels „.bat” oder alternativ „.cmd”. Wird die Batch-Datei ausgeführt, führt die Konsole, bzw. cmd.exe die Befehle der Reihe nach aus. Auch andere Batch Dateien können wiederum aufgerufen werden [56].
Folgende Befehle sind für das Starten eines Skriptes in SymTA/S notwendig:

cd \
cd Symta

start SymTA-Sc.exe -noSplash -data workspace -Skript -shell workspace\tutorials\Skripting \python_Skripting_example_environment.py workspace\....

- Mittels cd \ wird auf das “Root” Verzeichnis, meist C:\ (Festplatte C) gewechselt, danach wird in das Verzeichnis gewechselt, in der die “SymTA-Sc.exe” ist (cd SymtaS).
- „start SymTA-Sc.exe” startet dann das eigentliche Programm, das von verschiedenen nachfolgenden Parametern gesteuert und konfiguriert wird. Um das Analysetool im Hintergrund zu starten, kann „-noSplash” ausgeführt werden.
- „-data workspace” gibt an, wo die Daten geholt werden. Hier kann auch ein absoluter Wert für einen Pfad angegeben werden. Der nächste Befehl startet ein Skript, dass nachstehend angegeben werden muss: -Skript workspace\Diplomarbeit\Skripting\Oil_File_Einlesen.py

Das Skript kann auch noch im „Shell Modus” gestartet werden, wobei hier auch die Ausgabe in der Konsole erfolgt (-shell). Als letzter Wert wird noch das betreffende Repository angegeben. Durch ein Batch File, bzw. auch die Eingabe über die Command Line kann das

Analysetool SymTA/S auch von anderen Programmen gestartet werden. Eine weitere Möglichkeit ist die Ansteuerung mittels eines Remote Interface.

4.5 Remote Interface

Mittels Remote Interface ist es möglich, über bestehende Anwendungen bereits vorhandene Systeme in SymTA-S zu integrieren oder zu verändern. Die Schnittstelle arbeitet mit SOAP (Simple Object Access Protocol) ist zur Zeit aber auf JAVA auf Clientenseite beschränkt und kann nur eine Anfrage, die dann alle anderen blockiert, bzw. ignoriert, auf einmal verarbeiten. Ein großer Vorteil durch ein Remote Interface ist die einfache Portierung und Optimierung von bestehenden Systemen. Es werden daher keine Umkonventierungen oder zusätzliche Programme verwendet und man kann mit seinem bestehenden System und deren Toolchain arbeiten.

4.5.1 Einstellung in SymTA/S

Erstellung eines Clients

In einer Konsole (Windows Konsole) können mittels wsipmort die Klassen für den Klient importiert werden:

wsimport -Xnocompile -keep http://127.0.0.1:1088/symtaremoteinterface?wsdl

Wird nun die Main aufgerufen, wird ein Service erstellt, das die richtigen Einstellungen (URL, Server, ...) beinhaltet. Um mit dem Server zu kommunizieren bzw. Befehle oder Strings zu senden, muss immer eine Session beantragt werden. Dies gilt dann natürlich auch für die Methoden. Eine Anleitung für das interaktive Remote Interface, welches ein bestehendes System in SymTA/S importiert, findet sich in [62]. Zum Ausführen des Interfaces sind folgende Punkte notwendig um ein bestehendes System zu importieren:

- Installation von Java SE (Programmierschnittstelle)
- Eclipse
- SymTA/S.

SymTA/S oder der TraceAnalyzer müssen gestartet werden und das Remote Interface muss richtig konfiguriert werden („Preferences → Symtavision → Remote Interface"). Dort sollten schon alle Einstellungen richtig gesetzt sein. Gegebenenfalls können Benutzer- und Administratorpasswort geändert werden. Danach kann der Server schon gestartet werden (SymTA/S → Start Server). Zur Erstellung eines Clients sind folgende Schritte notwendig: Im

ersten Fall wird ein Testsystem erstellt, das den prinzipiellen Aufbau und die Vorgehensweise zeigt. In der IDE (Eclipse) wird ein neues Java- Projekt erzeugt, welches dann eine Verbindung zum Server von der Software aufbaut und alles nötigen Elemente importiert. Für diesen Zweck muss eine Konsole geöffnet werden, mittels der, im Workspace der IDE, folgender Befehl ausgeführt wird:
„*wsimport -Xnocompile -keep http://127.0.0.1:1088/symtaremoteinterface?wsdl*“ Hiermit sind alles Files mit den nötigen Klassen ins neue Projekt importiert.
Mögliche Fehlerquelle könnte sein, dass zu viele Java Versionen installiert sind bzw. wsimport nicht im richtigen Verzeichnis zu finden ist. Dann müssen die nicht nötigen Versionen entfernt werden, da es ansonsten zu Problemen mit der Ausführung von wsimport kommen kann. Weiters sollte überprüft werden ob wsimport im Bin Ordner der jeweiligen jdk Installation vorhanden ist. Wenn der Import funktioniert hat, sollte ein Paket „com.symtavision.symtas.remoteinterface.service“ vorhanden sein. Um nun das vorhandene Beispiel wiederum in SymTA/S einzulesen, muss noch eine Client Klasse erstellt werden. Es ist wieder darauf zu achten, dass die Klasse gleich wie das Paket heißt. Der nötige Code für die Client Klasse ist unter Punkt 8 im Manual schwer ersichtlich verlinkt: Client.java. Sollte die Main Funktion nicht ausgeführt werden, muss die JRE System Libary neu eingestellt werden. Ist die Verbindung erfolgreich hergestellt, kann die Erstellung und Konfiguration eines SymTA/S Projektes wieder wie gewohnt erfolgen. Es muss nur darauf geachtet werden, dass die Befehle mit einer Session verbunden werden, z.B. server.createProject(session, projectName). Auf diese Weise kann dann von Eclipse aus, SymTA/S beliebig konfiguriert werden [62]

4.5.2. Zugriff auf Enterprise Architect

Mittels eines Java Plugin [57] kann in Eclipse auf Enterprise Architect Dateien zugegriffen werden. Dies hat den Vorteil, dass alle Informationen, die in EA vorhanden sind, auch in Eclipse und durch das Remote Interface auch in SymTA/S verfügbar sind. EA unterstützt Multicore-Systeme, besitzt zusätzliche Informationen der Tasks, wie deren Laufzeit und die Visualisierung gemeinsamer Ressourcen kann auf Probleme aufmerksam machen (siehe Abbildung 36).

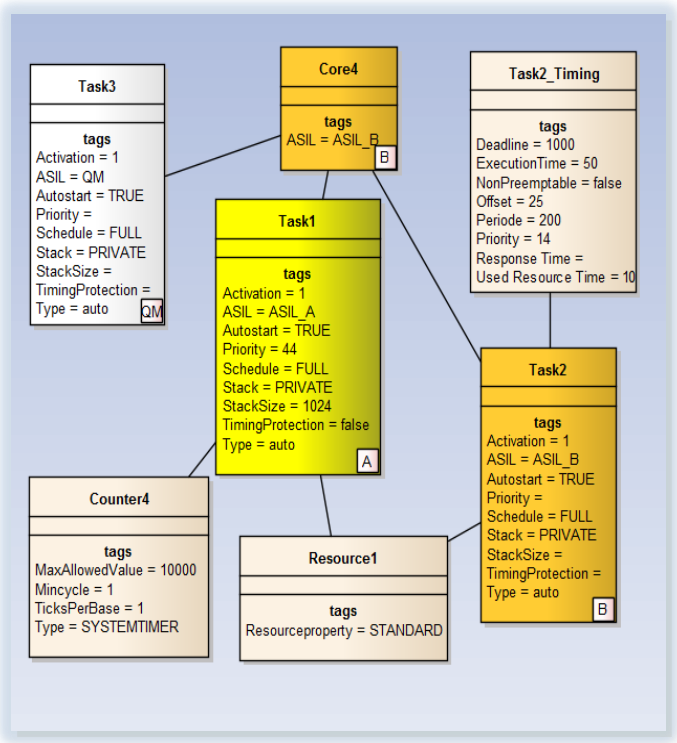


Abbildung 36 EA Inhalte [28]

EA greift auf Datenbanken zurück, um Informationen wie:

- Die ASIL Stufe
- Response Time
- Offset
- Execution Time
- Deadline
- Periode
- Used Resource Time

zu erhalten. Diese Abfragen für Zusatzinformationen von Elementen kann über das Plugin auch von Eclipse erfolgen, wie in Abbildung 37 ersichtlich. So stehen alle Informationen der EA Datenbank zur Verfügung.

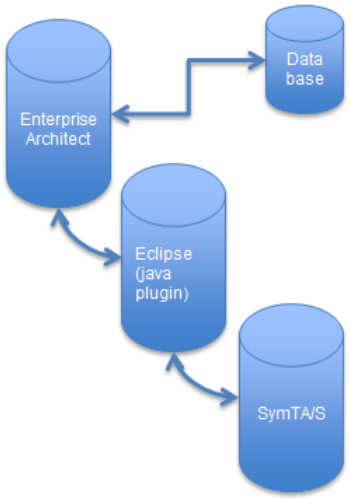


Abbildung 37 Eclipse als Schnittstelle

Über das Eclipse Plugin ist auch eine bidirektionale Kommunikation möglich, da Informationen von SymTA/S auch in die Datenbank geschrieben und somit aktualisiert werden können. Somit ist über das Remote Interface die Schnittstelle zu anderen Tools und Toolchains. Eine Eingliederung, Analyse und Optimierung bestehende System und Softwarekomponenten kann auf diese Weise einfach umgesetzt werden.

4.6 Timing Analyse

In den folgenden Punkten werden Timing Analysen und anderweitige relevante Analysen behandelt. Die Überprüfung einzelner Task auf die Einhaltung derer Laufzeit ist Voraussetzung für die Betrachtung des Gesamtsystems. Werden die erforderlichen Deadlines eingehalten, kann das Gesamtsystem betrachtet werden.

4.6.1 WCET

Für eine einfache WCET Analyse wird zunächst nur ein Singlecore-System betrachtet. Die Ausgangslage ist ein System mit 5 Tasks (siehe Abbildung 38) welche definierte Parameter wie Core Execution Time (min.-max.), Prioritäten und die maximale Antwortzeit erfordern. Ziel wird es sein, zu überprüfen, ob diese Überschritten wurde. In diesem Fall, wird die Laufzeit des 10ms Tasks überschritten (siehe Abbildung 39).

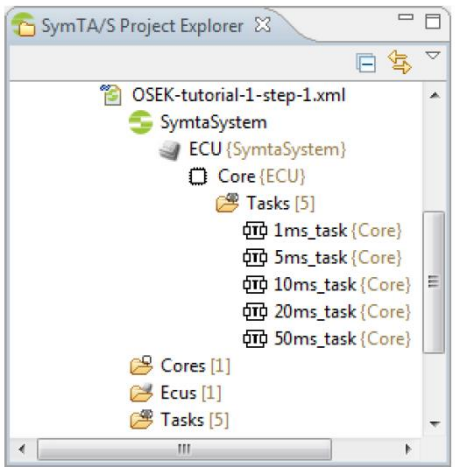


Abbildung 38 WCRT Use Case

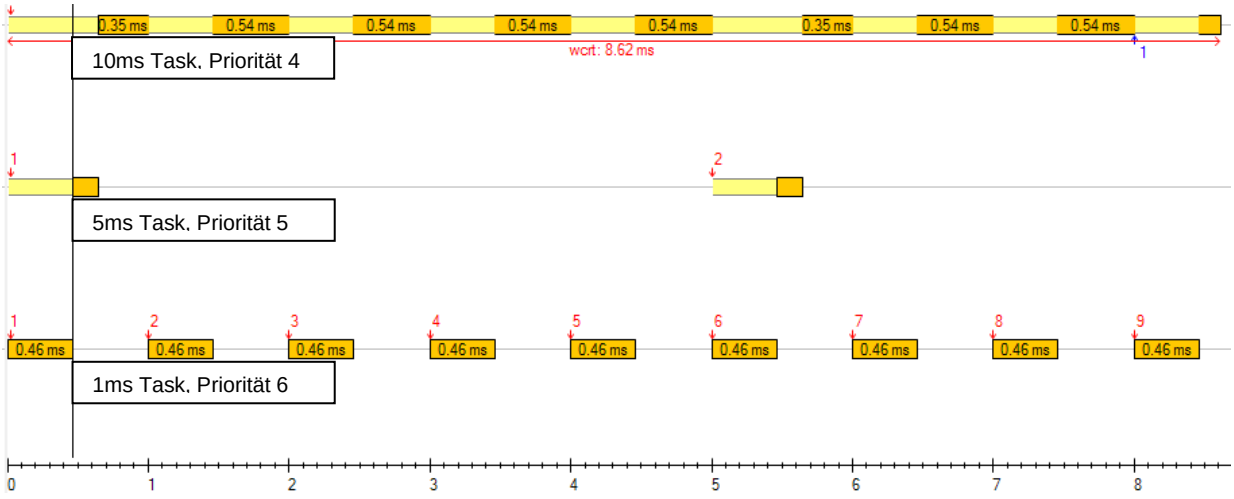


Abbildung 39 Überschreitung der Laufzeit eines Task

Abhilfe kann z.B. durch das Hinzufügen von Offsets schaffen. So wird der 10ms Task weniger oft unterbrochen und kann seine Deadline einhalten, siehe Abbildung 40.

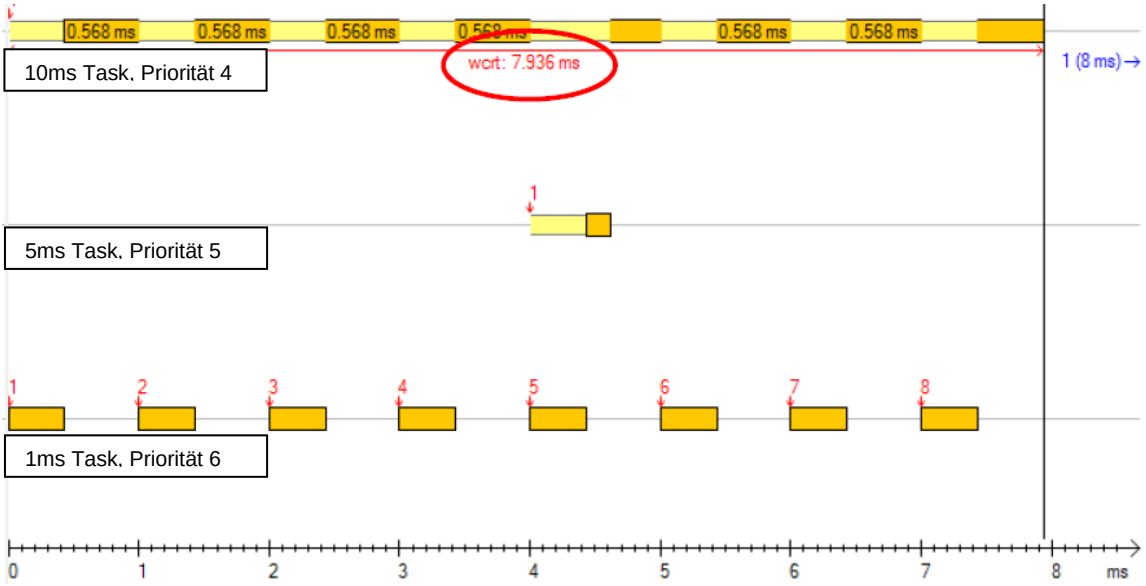


Abbildung 40 Einhaltung durch Offsets [27]

Natürlich können noch weitere Änderungen wie Optimierung der Laufzeit, Veränderung der Eigenschaften, wie Priorität oder Scheduling-Verfahren, vorgenommen werden, um die Deadlines zu erfüllen. Sind für den Task nur „weiche“ Echtzeitanforderungen gestellt, können auch Überlegungen angestellt werden, was ein nichteinhalten der Deadline für Konsequenzen nach sich zieht.

4.6.2 Timing Analyse eines Multicore Systems

Ist das System stabil, das heißt dass die jeweiligen Tasks ihre Deadlines einhalten, kann das Gesamtsystem betrachtet werden. Als Ausgangslage dient ein System mit zwei Kernen und zehn Tasks. Wobei die Aufteilung auf 4 Tasks auf dem Core1 und 6 Task auf dem Core2 geregelt ist. In Abbildung 41 und Abbildung 42 sind die Tasks der jeweiligen Cores und deren Verknüpfungen zueinander über gemeinsame Variablen ersichtlich. Die Eingabe des Kommunikations-Overheads für dieses Beispiel sind 144 Einträge für 10 Tasks, 11 Variablen und zwei Speichereinheiten.

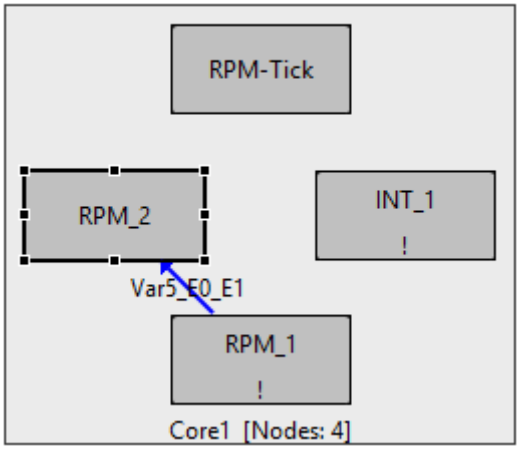


Abbildung 41 Task Graph von Core1

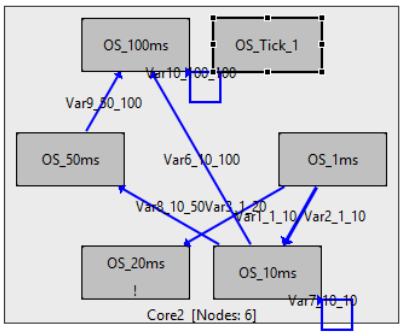


Abbildung 42 Task Graf von Core2

Da der Kommunikations-Overhead, besonders über verschiedene Kerne auf deren Speicher, sehr hoch werden kann, sollten diese Zugriffe durch besseres Verteilen der Task und Variablen, minimiert werden (siehe Abbildung 43 und Abbildung 44).

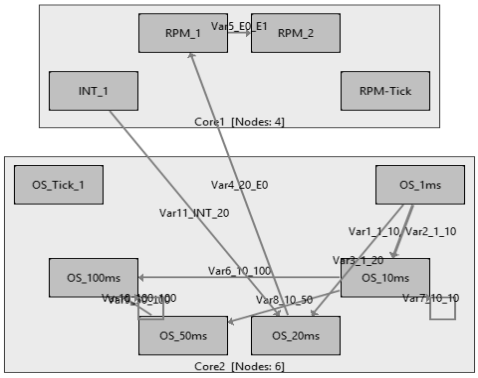


Abbildung 43 Verknüpfung von Tasks auf verschiedenen Cores.

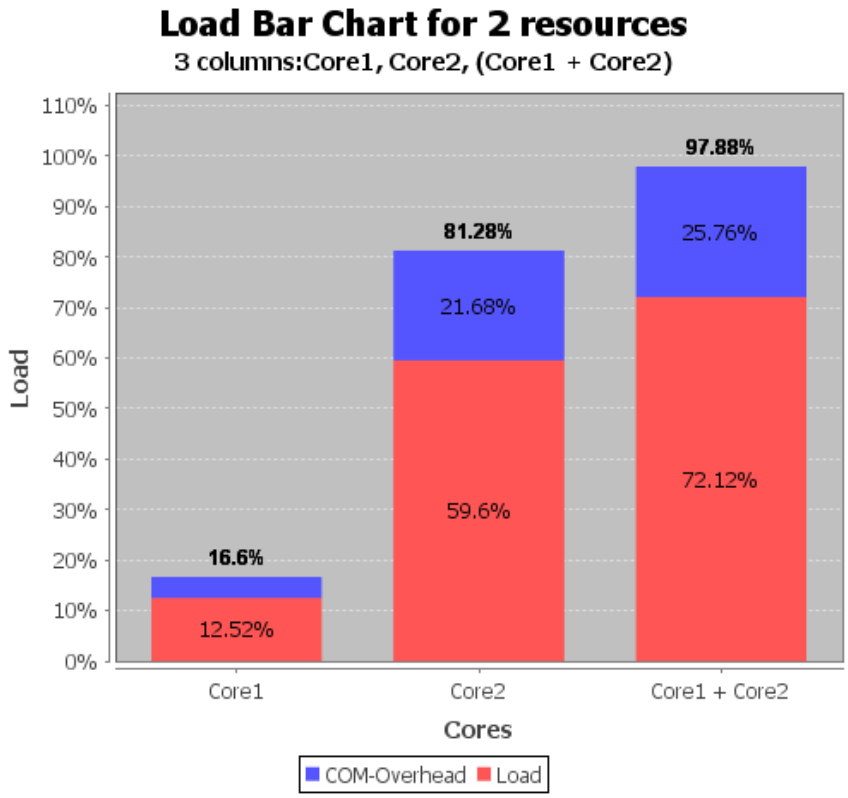


Abbildung 44 Ungleiche Auslastung von 2 Cores

4.7 Kommunikation Overhead Analyse

SymTA/S kann den Kommunikations-Overhead automatisch von einer XML Datei importieren. Jedoch ist auch wieder eine Importierung durch ein manuell erstelltes Python-Skript möglich. Die im bestehenden System vorhandenen Einheiten wie CPUs und Speicher können dann den zu importierenden Werten zugeordnet werden. SymTA/S bietet die Möglichkeit, die verschiedenen Overheads nach CPUs, Cores, Tasks usw. anzuzeigen. Dadurch kann man den Kommunikations-Overhead besser auf die verschiedenen Cores aufteilen. Der Kommunikations-Overhead kann in SymTA/S manuell eingegeben oder importiert werden. Die zu importierenden Werte können von einem Trace Tool mittels XML File stammen. Folgende Parameter können für den Kommunikations-Overhead gesetzt werden:

- Call Type: Für welche Art von Zugriff wurde der Overhead gemessen (Pure /RTE)
- Access Type: Für welchen Zugriff (Lesen/Schreiben) gilt der Overhead
- Instruction Core: Auf welchen Core bezieht sich der Overhead
- Data Memory: Definiert, welcher Memory verwendet wird
- Size [bit]: Definiert die Größe der Daten, für die der Wert zutrifft
- Duration Definiert die Zugriffszeit, die benötigt wird.

Die einfachste Möglichkeit, den Kommunikations-Overhead für die jeweiligen Elemente einzugeben, ist der Import über XML Files (File - Import - Model Import). Die dazugehörigen Elemente sollten schon im Vorfeld übereinstimmen und das Zuweisen zu vereinfachen. Eine mögliche Fehlerquelle ist die falsche Zuordnung des XML zu einer falschen ECU oder das falsche Zuweisen der Speicherelemente.

Für die Analyse muss ein Name definiert werden und auf welcher ECU sie angewandt werden soll. Beim Ausführen von Analysen ist es notwendig, Informationen in das Modell zurückzuschreiben. Diese Informationen können entweder für Benutzer oder Skripte hilfreich sein. Data Annotion können zu jedem Zeitpunkt an jedes beliebige Element angewendet werden. In der Worst Case Analyse wird die Last für jeden Core oder für alle gemeinsam angezeigt. In Abbildung 45 sind die Last im roten Balken und der Overhead im blauen Balken abzulesen. In der Analyse sollte auch unbedingt der Overhead beachtet werden, um die verschiedenen Tasks optimal aufteilen.

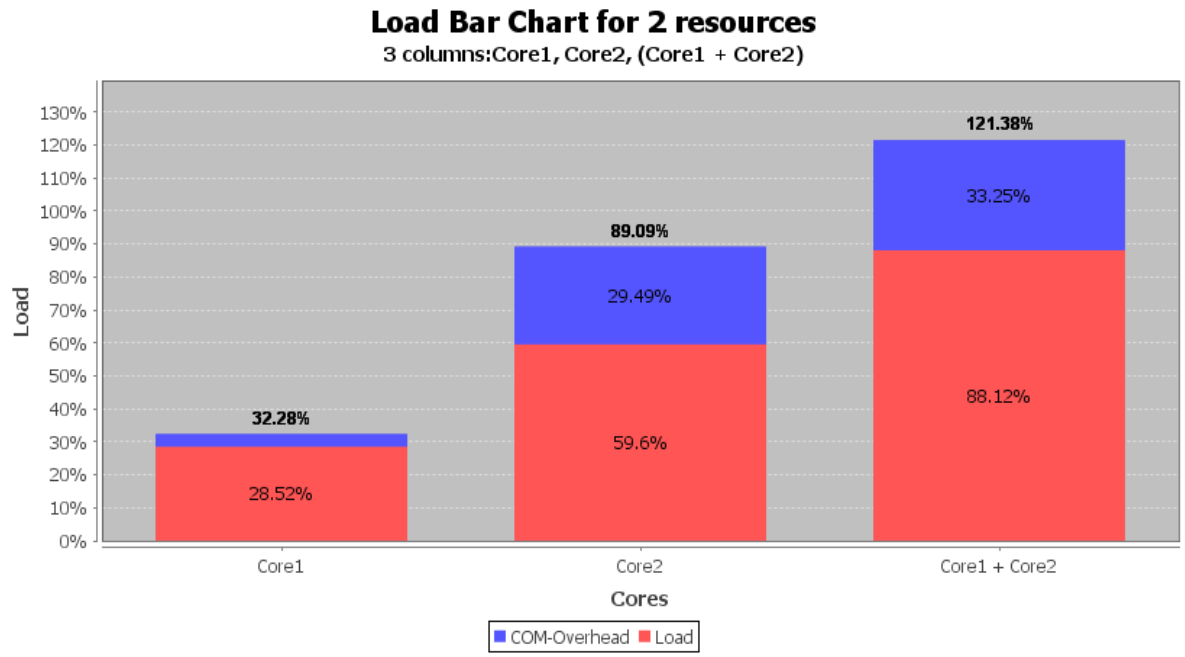


Abbildung 45 Kernauslastung mit Overhead

Durch die verschiedenen Optionen ist es auch möglich, über verschiedene Anzeigestile (Torten- oder Balkendiagramm). In Abbildung 46 ist ersichtlich, wieviel Zeit jeder Task benötigt. Man könnte den Overhead noch nach Task optimieren, um eine bessere Performance zu erreichen.

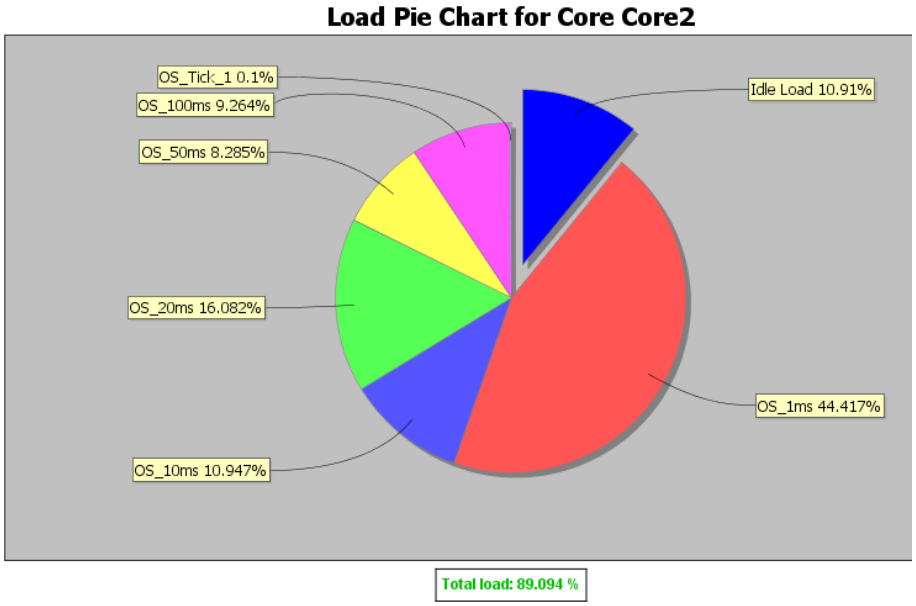


Abbildung 46 Auslastung von Core 2 nach Tasks

In Abbildung 47 ist die Auslastung mit Overhead abzulesen. Diese Darstellung kann Optimierungspotential der Tasks aufzeigen. Kann der Overhead minimiert werden, durch besseres Aufteilen und Verwalten der gemeinsamen Ressourcen, wird auch der Kern entlastet.

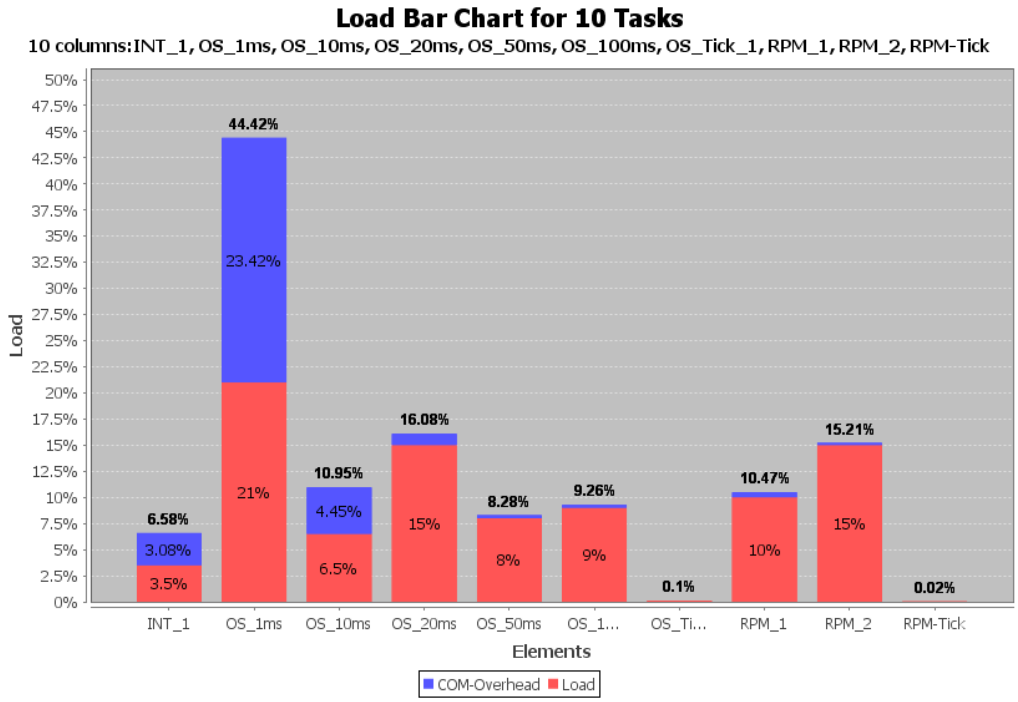


Abbildung 47 Tasks mit Kommunikation Overhead

Der Task OS_1ms benötigt 21% der Rechenzeit, hat aber einen mehr als doppelt so großen Overhead mit 23,42%. Ein weiterer wichtiger Punkt bei der Analyse eines Systems, bzw.

Cores ist die maximale Response Time. Die Response Time ist die Gesamtantwortzeit, also die Zeitspanne von der Anfrage bis die Rückantwort übermittelt wird. In Abbildung 48 ist diese für einen 10ms Task ersichtlich. Diese beträgt im vorliegenden Fall 7,9ms.

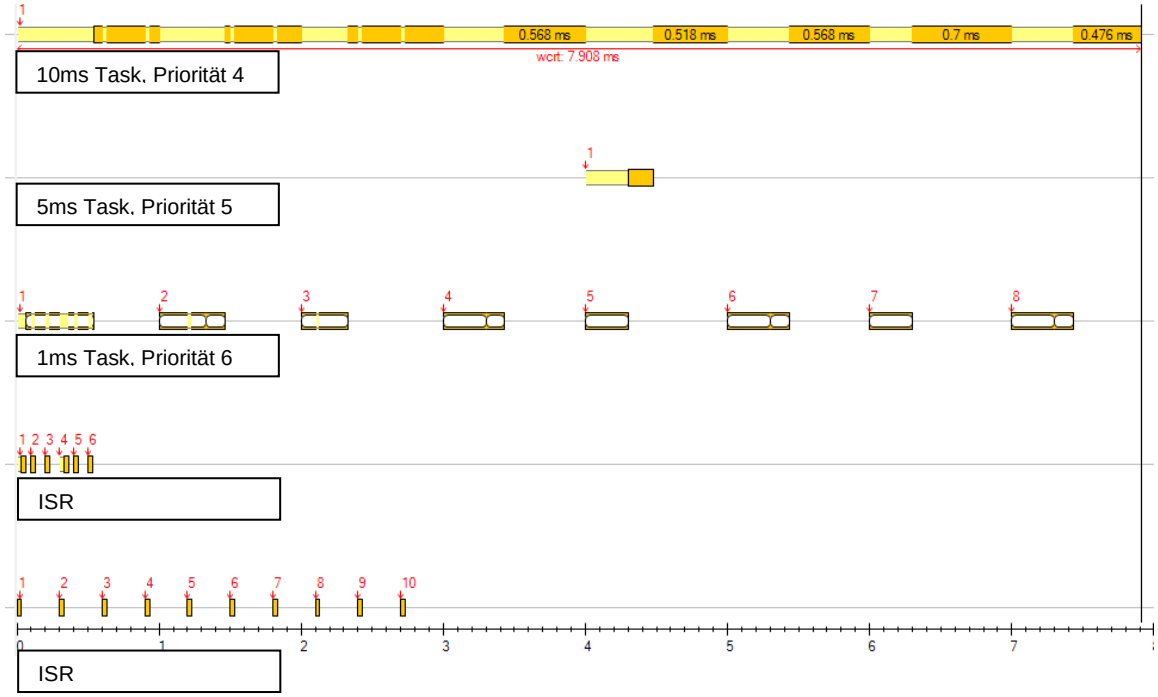


Abbildung 48 WCRT für 10ms Task

Durch lange Blockier-Zeiten und Warten auf Ressourcen oder zu häufiges Wechseln und Unterbrechen eines Tasks können diese Zeit und auch der Kommunikation-Overhead erheblich steigen. In Abbildung 49 ist das Response Time Profile für Core1 ersichtlich.

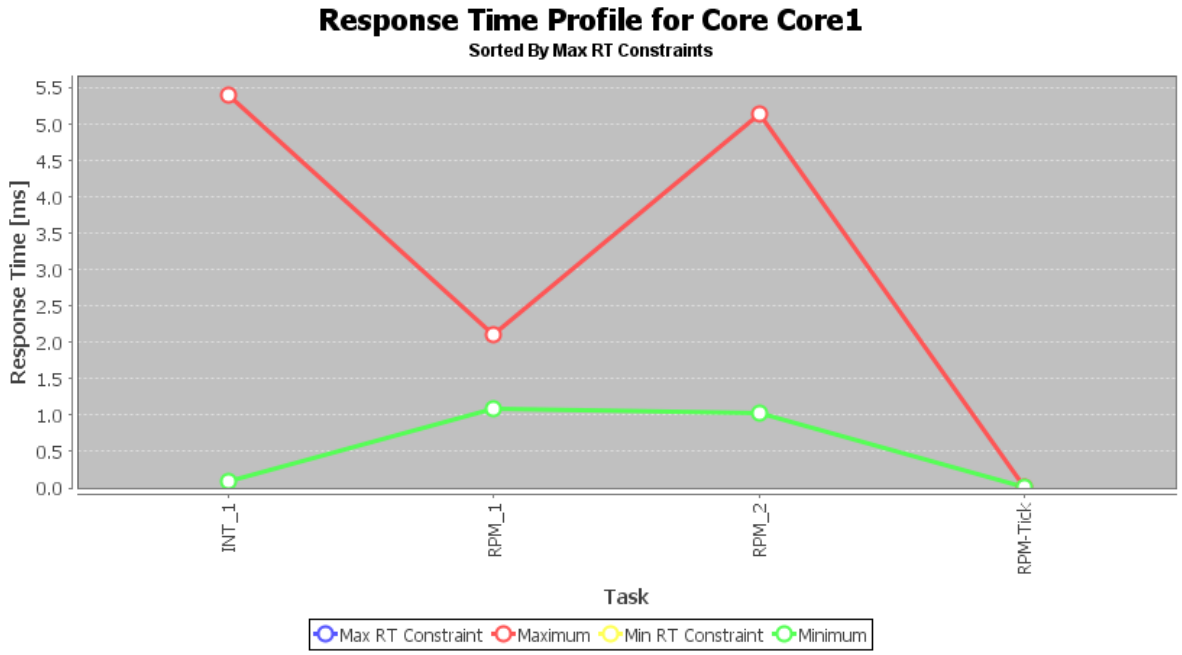


Abbildung 49 Response Time

Sind der minimale und maximale Wert zu weit entfernt, sollten Überlegungen angestellt werden, ob diese Werte tolerierbar sind oder optimiert gehören. Wichtig ist, dass die vordefinierte maximale Antwortzeit nicht überschritten wird. Hierzu sind noch weiter Parameter anzugeben, wie z.B. die Lesedauer für den Speicher bzw. auch verschieden große Blöcke von Speicherinhalten.

Durch die verschiedenen Einstellungsmöglichkeiten für die Analyse können für die jeweiligen Elemente die Diagramme analysiert werden und gegebenenfalls Optimierungen vorgenommen werden. In Abbildung 50 ist nochmals ein Diagramm, welches Runnables abbildet. Hier wird der Overhead nochmals auf die jeweiligen Runnables runtergebrochen. Dieser kann verschiedenste Dimensionen annehmen. Durch das Analysieren der Runnables können auch hier Überlegungen angestellt werden, diese auf andere Task aufzuteilen.

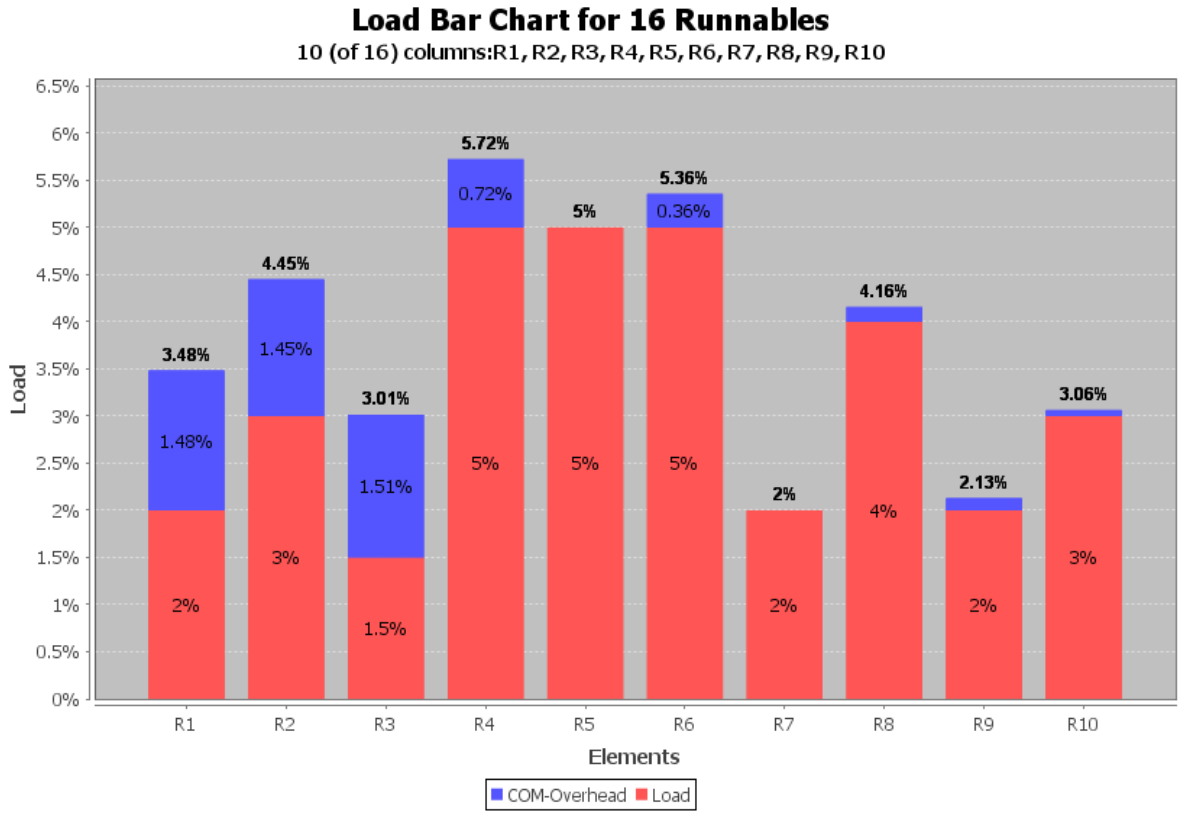


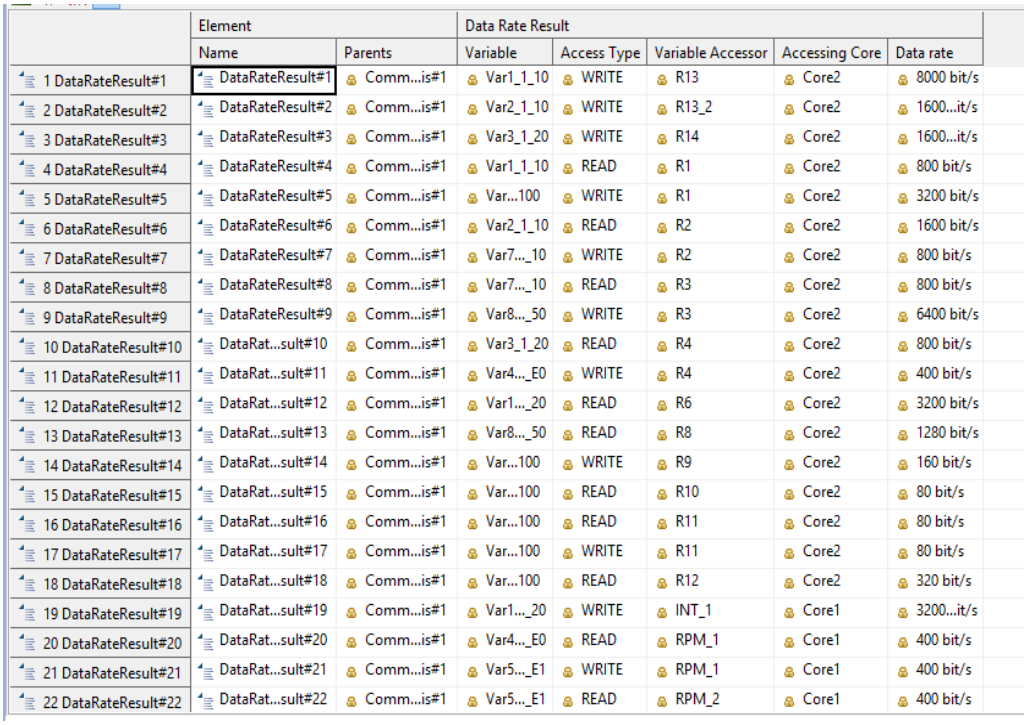
Abbildung 50 Overhead von Runnables

4.8 Optimierung des Kommunikations-Overhead

Wie bereits in den Analysen im vorherigen Punkt erwähnt, ist der Kommunikations-Overhead ein ausschlaggebender Punkt für die Leistungsfähigkeit und auch Sicherheit eines Multicore-Systems. Um das System schneller und leistungsfähiger zu machen, sollte dieser aufs Nötigste reduziert werden.

4.8.1 Datenrate Metrik

Die Datenrate Metrik berechnet die durchschnittliche Datenrate zwischen einer Variablen und deren “”Aufrufer”, also Task oder Runnable. Um eine Unabhängigkeit von der Plattform (WCET) bzw. der Hardware (Anzahl der Kerne, Speichertyp, …) zu erreichen, wird die Rate mittels des Aktivierungsmodels des Elements, den Zugriffszähler während eines Ausführung der Ressource und der variablen Größe errechnet. Ein periodischer Task (10ms) greift z.B. zweimal auf eine Variable mit der Größe 8bit während seiner Ausführung zu. Somit beträgt die Datenrate 1 / 10ms * 8bit * 2 = 100 1 / s * 8bit * 2 = 1600 bit / s. Für ein Multicore-Beispiel sieht die Metrik wie in Abbildung 51 ersichtlich aus.



	Element		Data Rate Result				
	Name	Parents	Variable	Access Type	Variable Accessor	Accessing Core	Data rate
1 DataRateResult#1	DataRateResult#1	Comm...is#1	Var1_1_10	WRITE	R13	Core2	8000 bit/s
2 DataRateResult#2	DataRateResult#2	Comm...is#1	Var2_1_10	WRITE	R13_2	Core2	1600...it/s
3 DataRateResult#3	DataRateResult#3	Comm...is#1	Var3_1_20	WRITE	R14	Core2	1600...it/s
4 DataRateResult#4	DataRateResult#4	Comm...is#1	Var1_1_10	READ	R1	Core2	800 bit/s
5 DataRateResult#5	DataRateResult#5	Comm...is#1	Var...100	WRITE	R1	Core2	3200 bit/s
6 DataRateResult#6	DataRateResult#6	Comm...is#1	Var2_1_10	READ	R2	Core2	1600 bit/s
7 DataRateResult#7	DataRateResult#7	Comm...is#1	Var7..._10	WRITE	R2	Core2	800 bit/s
8 DataRateResult#8	DataRateResult#8	Comm...is#1	Var7..._10	READ	R3	Core2	800 bit/s
9 DataRateResult#9	DataRateResult#9	Comm...is#1	Var8..._50	WRITE	R3	Core2	6400 bit/s
10 DataRateResult#10	DataRat...sult#10	Comm...is#1	Var3_1_20	READ	R4	Core2	800 bit/s
11 DataRateResult#11	DataRat...sult#11	Comm...is#1	Var4..._E0	WRITE	R4	Core2	400 bit/s
12 DataRateResult#12	DataRat...sult#12	Comm...is#1	Var1..._20	READ	R6	Core2	3200 bit/s
13 DataRateResult#13	DataRat...sult#13	Comm...is#1	Var8..._50	READ	R8	Core2	1280 bit/s
14 DataRateResult#14	DataRat...sult#14	Comm...is#1	Var...100	WRITE	R9	Core2	160 bit/s
15 DataRateResult#15	DataRat...sult#15	Comm...is#1	Var...100	READ	R10	Core2	80 bit/s
16 DataRateResult#16	DataRat...sult#16	Comm...is#1	Var...100	READ	R11	Core2	80 bit/s
17 DataRateResult#17	DataRat...sult#17	Comm...is#1	Var...100	WRITE	R11	Core2	80 bit/s
18 DataRateResult#18	DataRat...sult#18	Comm...is#1	Var...100	READ	R12	Core2	320 bit/s
19 DataRateResult#19	DataRat...sult#19	Comm...is#1	Var1..._20	WRITE	INT_1	Core1	3200...it/s
20 DataRateResult#20	DataRat...sult#20	Comm...is#1	Var4..._E0	READ	RPM_1	Core1	400 bit/s
21 DataRateResult#21	DataRat...sult#21	Comm...is#1	Var5..._E1	WRITE	RPM_1	Core1	400 bit/s
22 DataRateResult#22	DataRat...sult#22	Comm...is#1	Var5..._E1	READ	RPM_2	Core1	400 bit/s

Abbildung 51 Datenrate Metrik

Der Variablentyp, der Zugriffstyp (Lesen/Schreiben) und der aufrufende Task bzw. Core und die Datenrate haben für eine weitere Analyse einen wichtigen Stellenwert.

4.8.2 Kommunikations-Overhead-Metrik

Die Kommunikations-Overhead-Metrik berechnet die Last, welche der Zugriff auf eine Variable auf einem bestimmten Core verursacht. Dies kann dann als Kommunikations-Overhead durch reine Kommunikation bezeichnet werden. Dazu muss für jeden möglichen Task/Runnable für die Variable ein Wert errechnet werden. Das Ergebnis ist der Wert, den das aufrufende Element benötigt, um auf die Variable zuzugreifen. All diese Zeiten werden für jeden Core und Variable zusammengefasst und in der Variablenlast gespeichert. Zwingend notwendig ist es natürlich, dass die Konfiguration des Kommunikations-Overhead im System bekannt und konfiguriert ist. Über eine Experiment Analyse in SymTA/S kann die Last der Variablen für jeden Core ausgerechnet werden. Die Ergebnisse der jeweiligen Zugriffszeiten auf Variablen sind in Abbildung 52 ersichtlich.

	Element		Variable Load Result				
	Name	Parents	Buffer	Memory	Mapped on Core	Accessing Core	Load
1 Var1_1_10 Core2 load	Var1_1_10 Core2 load	Commun...ysis#1	Var1_1_10	DMI_Core2	Core2	Core2	0.0578175
2 Var2_1_10 Core2 load	Var2_1_10 Core2 load	Commun...ysis#1	Var2_1_10	DMI_Core2	Core2	Core2	0.063365
3 Var3_1_20 Core2 load	Var3_1_20 Core2 load	Commun...ysis#1	Var3_1_20	DMI_Core2	Core2	Core2	0.061045
4 Var4_20_E0 Core1 load	Var4_20_E0 Core1 load	Commun...ysis#1	Var4..._E0	DMI_Core1	Core1	Core1	0.00206275
5 Var4_20_E0 Core2 load	Var4_20_E0 Core2 load	Commun...ysis#1	Var4..._E0	DMI_Core1	Core1	Core2	0.00378185
6 Var5_E0_E1 Core1 load	Var5_E0_E1 Core1 load	Commun...ysis#1	Var5..._E1	DMI_Core1	Core1	Core1	0.004744
7 Var6_10_100 Core2 load	Var6_10_100 Core2 load	Commun...ysis#1	Var...100	DMI_Core2	Core2	Core2	0.006937
8 Var7_10_10 Core2 load	Var7_10_10 Core2 load	Commun...ysis#1	Var7..._10	DMI_Core2	Core2	Core2	0.009555
9 Var8_10_50 Core2 load	Var8_10_50 Core2 load	Commun...ysis#1	Var8..._50	DMI_Core2	Core2	Core2	0.0079495
10 Var9_50_100 Core2 load	Var9_50_100 Core2 load	Commun...ysis#1	Var...100	DMI_Core2	Core2	Core2	0.00149175
11 Var10_100_100 Core2 load	Var10_100_100 Core2 load	Commun...ysis#1	Var...100	DMI_Core2	Core2	Core2	0.0009555
12 Var11_INT_20 Core1 load	Var11_INT_20 Core1 load	Commun...ysis#1	Var1..._20	DMI_Core1	Core1	Core1	0.034
13 Var11_INT_20 Core2 load	Var11_INT_20 Core2 load	Commun...ysis#1	Var1..._20	DMI_Core1	Core1	Core2	0.00391555

Abbildung 52 Last in Prozent für jeden Core und jede Variable

Das folgende Variable Load Diagramm in Abbildung 53 zeigt das Optimierungspotential an.

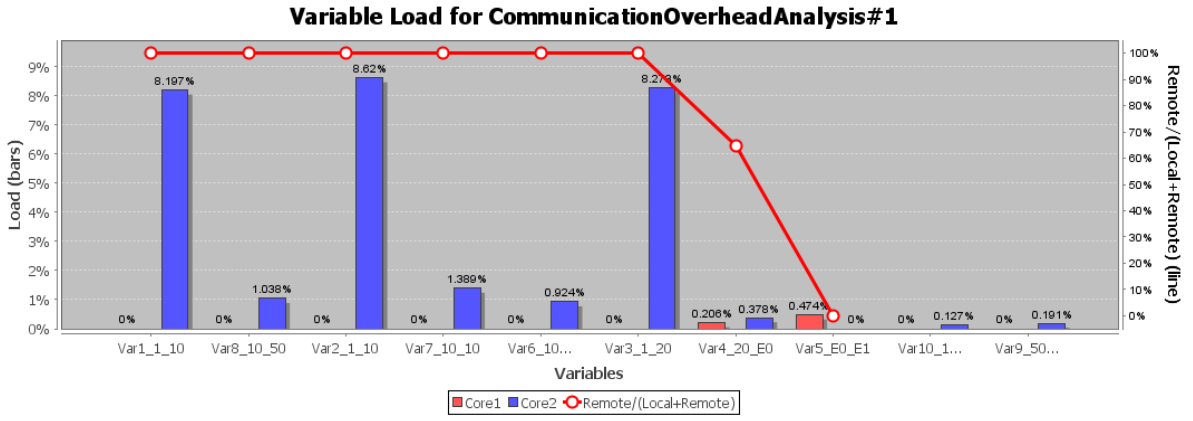


Abbildung 53 Kommunikations-Overhead

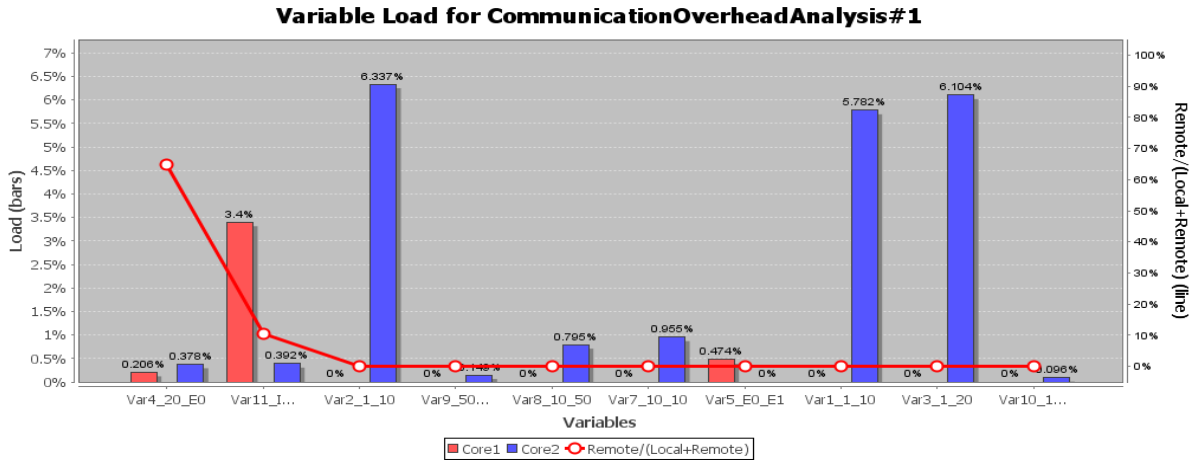


Abbildung 54 verbesserter Kommunikations-Overhead

Auf der X-Achse werden alle Variablen für jeden Core angezeigt. Die Last bezieht sich auf die Gesamtlast auf jedem Core In Abbildung 54 ist eine bessere Verteilung zu sehen. Var11

verursacht 3,4% der Last von Core 1 für deren Zugriff, aber nur 0,392% wenn Core 2 auf diese Variable zugreift. Die rote Linie kennzeichnet für jede Variable die Prozent des Overheads, um auf die Variable vom Remote Core aus zuzugreifen, bezogen auf den gesamten Overhead des Zugriffs auf die Variable. Der Wert wird auf der rechten Seite abgelesen und kann nur ein zielführendes Ergebnis bringen, wenn die Variable auf einen lokalen Speicher (der eines Cores) liegt. Erst unter dieser Bedingung kann ein Verhältnis der Gesamtlast zu „Remote" Last hergestellt werden. Ist der Wert der Kennlinie hoch, so sollte die Variable auf den Speicher eines anderen Core ausgelagert werden. Durch die Analyse des Diagramms können Zuordnungen von Variablen optimiert werden. Wird auf eine Variable von einem gewissen Kern aus aufgerufen (hohe Last für diesen Kern), so sollte sie „nahe" an diesem Kern (lokaler Speicher) platziert werden. Dies könnte beispielsweise ein lokaler Speicher des Kerns oder eine schnellerer globalen Speicher oder beides sein. Ein schneller Speicher würde die Zugriffszeit bzw. die Last für den Variablenzugriff reduzieren.

4.9 Durchführung von Analysen - SymTA-Experiment

Um ein System zu optimieren bzw. dessen freie Ressourcen und Engpässe festzustellen, ist es nötig, verschiedene Aspekte zu analysieren. Die Laufzeitanalyse, Jitter, Abhängigkeiten oder Datenkonsistenz sind dabei wichtige Punkte, die Auswirkungen auf das Verhalten des Gesamtsystems haben können. SymTA/S bietet für diese Optimierungen das Feature SymTA-Experiment an. Diese Optimierungen können in drei Kategorien geteilt werden:

- Konfiguration
- Umsetzung und
- Übernehmen ins System.

Besonders interessant sind natürlich die Ergebnisse, die ins bestehende System übernommen werden können. Hierbei ist zu beachten, dass bestehende Werte von Tasks, Ressourcen usw. überschrieben werden. Für die folgenden Eigenschaften können Verbesserungen bzw. Optimierungen in SymTA-Experiment erzielt werden:

- Dependency Offset (by selecting an experiment result)
- Sensitivity Frame Count (CAN Bus)
- Sensitivity Frame Size (CAN Bus)
- Sensitivity Jitter
- Sensitivity Period
- Sensitivity Resource Speed

- Sensitivity WCET Task
- Sensitivity WCET Runnable
- Sensitivity WCET Additional Runnable.

Mit der Bibliothek der Sensitivitätsanalyse kann festgelegt werden, wie hoch die Sensibilität des Schedulings gegenüber Veränderungen in den verschiedenen Bereichen eines individuellen Systems und dessen Ressourceneigenschaften ist. Im besonderen Core Execution Times (CET), veränderten Frequenzen der CPU und auch geänderte Häufigkeiten von Events, können betrachtet und optimiert werden. Die Analysen beinhalten auch wichtige Information über die Systemlast. Diese Informationen sind wichtig für weitere Erkenntnisse über Engpässe bzw. Reserven und der Leistung des Systems. Die Funktionalität eines einzelnen Tasks kann unter Berücksichtigung von Systemeinschränkungen optimiert werden. Sind die geeigneten Parameter festgelegt, die optimiert werden dürfen, kann sogar aus der Ausgangslage eines instabilen Systems ein stabiles erzielt werden. Das eignet sich besonders für Systeme, die wachsen und durch neue Funktionalitäten plötzlich nicht mehr korrekt laufen. Die Analyse erfolgt mittels eines transienten Systemmodells bestehend aus vielen unterschiedlichen Experimenten. Die Veränderungen bzw. Verbesserungen einer Analyse können dann gleich für die Ausführung einer weiteren Analyse herangezogen werden, indem diese wiederholt werden. Durch die immer wiederkehrenden unterschiedlichen Ausgangslagen ist es sinnvoll, die Analysen öfters auszuführen um auf die unterschiedlichen Ausgangswerte einfließen zu lassen. Ziel der Analyse ist es, soweit an die Grenzen des System zu gehen, das dieses noch stabil läuft, aber trotzdem optimal ausgelastet werden kann und auch nicht Ressourcen verschwendet werden, bzw. frei bleiben. Durch den sogenannten „precision value" kann die Grenze bestimmt werden, ab wann die Analyse endet. Das bedeutet, dass der Abstand des stabilen und des instabilen Zustandes diesen Wert erreicht.

Im folgenden Beispiel, siehe Abbildung 55, wird die Grenze gefunden, an der die CET optimal ist. Wie bereits zuvor erwähnt, werden die Parameter von Analyse zu Analyse verändert. Die CET wird Schritt für Schritt erhöht bis das System instabil wird. Ab diesem Wert wird die CET wieder reduziert, bis das System wieder stabil ist. Die "Precision" ist in diesem Fall die orange Linie, zwischen dem stabilen und instabilen Zustand und der erreichte Wert lässt die Analyse enden.

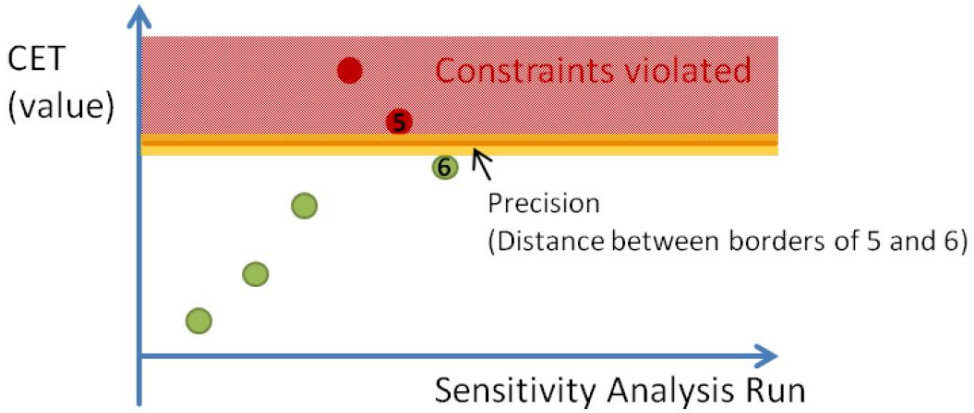


Abbildung 55 Optimierung der CET [63]

Die Analyse der Frames (Count und Size) sind nur für die Analyse eines CAN Buses relevant, aber nicht für die in dieser Arbeit untersuchten Multicore Constrains. Relevant sind aber die restlichen, verbleibenden Analysen (Jitter, Period, Resource Speed, WCET Task (analog zu WCET Runnable), WCET Runnable, WCET Additional Runnable) wobei Jitter und Period in dieser Arbeit nicht behandelt werden und WCET Task und Runnable den gleichen Vorgang haben und nur einmal beschrieben und durchgeführt wird. Für die Engstellen, bzw. deren Auslastung (CPU, Speicher und dessen Zugriffszeiten von verschiedenen Stellen, Busgeschwindigkeit...) kann die Ressourcen Speed Analyse hilfreich sein. Zu diesem Zwecke müssen auch die Tasks genauer untersucht werden, bezogen auf deren WCET. Die möglichen Runnables, die den Tasks zugeordnet werden können, können im Absoluten betrachtet werden, also die Gesamtdauer aller Runnables pro Task, oder in Relation (in Prozent) zu deren vorheriger Zuordnung. Die Ausführungszeit eines Tasks kann auch noch optimiert werden, indem analysiert wird, wieviel Prozent er länger benötigen darf.

4.9.1 Sensitivity WCET Runnable

Für ein System mit sieben Tasks (1ms_task, 5ms_task, 10ms_task, 20ms_task, 50ms_task und zwei ISR Task) wird versucht, ein zusätzliches Runnable dem 1ms Task zuzuordnen. Der Kern ist schon zu 84,07 % ausgelastet. Um die Analyse für ein Runnable zu starten, muss unter dem Punkt Experimente in dem jeweiligen Projekt ein neues “Experiment” Sensitivity WCET Runnables hinzugefügt werden und für das zu analysierende Runnable unter dem gleichnamigen Punkt angegeben werden. Die Werte Min Load, Max Load und Precision sollten zwischen 0,01 und 0,97 (97% Auslastung) bzw. bei ca. 0,01ms liegen. Die Einstellungsmöglichkeiten und Ergebnisse sind in Abbildung 56 zu sehen.

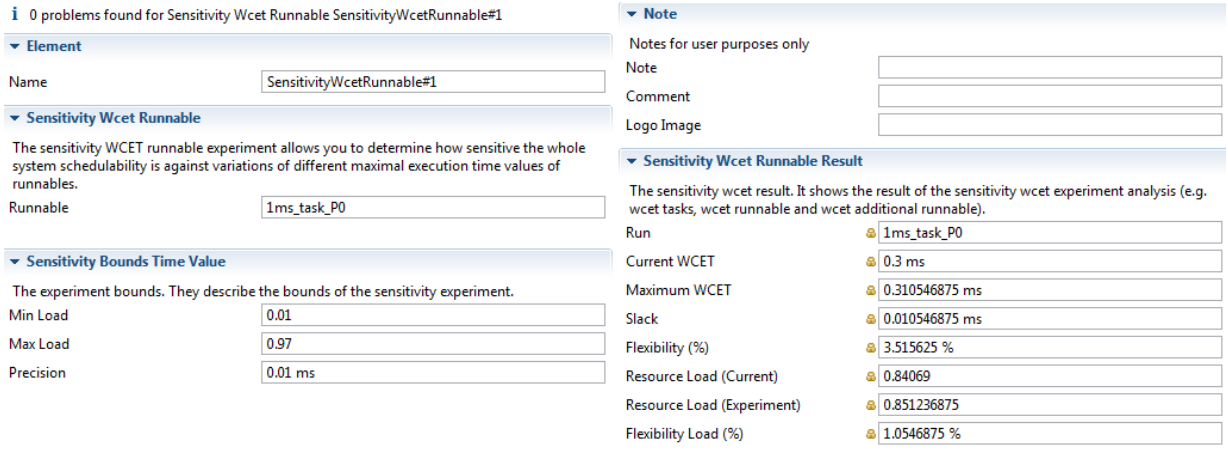


Abbildung 56 Einstellungsmöglichkeiten WCET Optimierung

In diesem Fall konnte ein Slack von ca. 0,01ms erreicht werden, was einer Flexibilität von 3,5% bedeutet. Wie dieser Wert erreicht wird, veranschaulicht folgendes Logfile, welches nach [63] erstellt wurde:

“Experiment analyses started...
SensitivityWcetRunnableAnalysis for SensitivityWcetRunnable#1
Getting experiment config...
Analyzing config (Runnable: 1ms_task_P0, ExecutionTime: 0.3 ms)
Analyzing config (Runnable: 1ms_task_P0, ExecutionTime: 0.6 ms)
Task 10ms_task is overloaded and will be ignored.
Total load of Core is greater than the configured maximal load 0.97.”

Die Execution Time wird auf 0,3ms gesetzt und dann auf 0,6ms erhöht. Hierbei tritt eine Verletzung auf, dass der 10ms Task überlastet ist und die CPU Last über 97% gestiegen ist,

Task 10ms_task is overloaded and will be ignored.
Could not validate constraint for 10ms_task. Setting the current value as the system bound.
Analyzing config (Runnable: 1ms_task_P0, ExecutionTime: 0.45 ms)
Total load of Core is greater than the configured maximal load 0.97.
Maximum response time constraint at 10ms_task not met (WC analysis). Response time should not exceed 8 ms but is 10.772 ms
Maximal constraint (8 ms) of task 10ms_task is violated (RT:10.772 ms).

Die Execution Time wird durch die Überschreitung wieder auf 0,45ms reduziert, was aber wieder eine Runtime Überschreitung vom 10ms Task hervorruft. Die Antwortzeit sollte 8ms

nicht überschreiten, hat in diesem Fall aber 10,772 ms. In den folgenden Schritten wird die Execution time vom Runnable solange verringert, bis keine Verletzung vorliegt.

Analyzing config (Runnable: 1ms_task_P0, ExecutionTime: 0.375 ms)
Maximum response time constraint at 10ms_task not met (WC analysis). Response time should not exceed 8 ms but is 8.883 ms
Maximal constraint (8 ms) of task 10ms_task is violated (RT:8.883 ms).
Analyzing config (Runnable: 1ms_task_P0, ExecutionTime: 0.3375 ms)
...
Analyzing config (Runnable: 1ms_task_P0, ExecutionTime: 0.309375 ms)
...
Analyzing config (Runnable: 1ms_task_P0, ExecutionTime: 0.310546875 ms)
Experiment bounds (0.0078125 ms) have reached the precision (0.01 ms).
Writing result (Runnable: 1ms_task_P0, ExecutionTime: 0.310546875 ms)
Experiment analyses finished.

Die Analyse hat die Schrittgrenze erreicht und auch keine Verletzung der Zeiten festgestellt. Das Ergebnis für den 1ms Runnable beträgt wie bereits erwähnt 0,310546875ms.

4.9.2 Sensitivity WCET Additional Runnable Analysis

Das nächste Experiment soll für ein mögliches zusätzliches Runnable die Execution Time berechnen. Dazu können wiederum verschiedene Parameter definiert werden (siehe Abbildung 57).

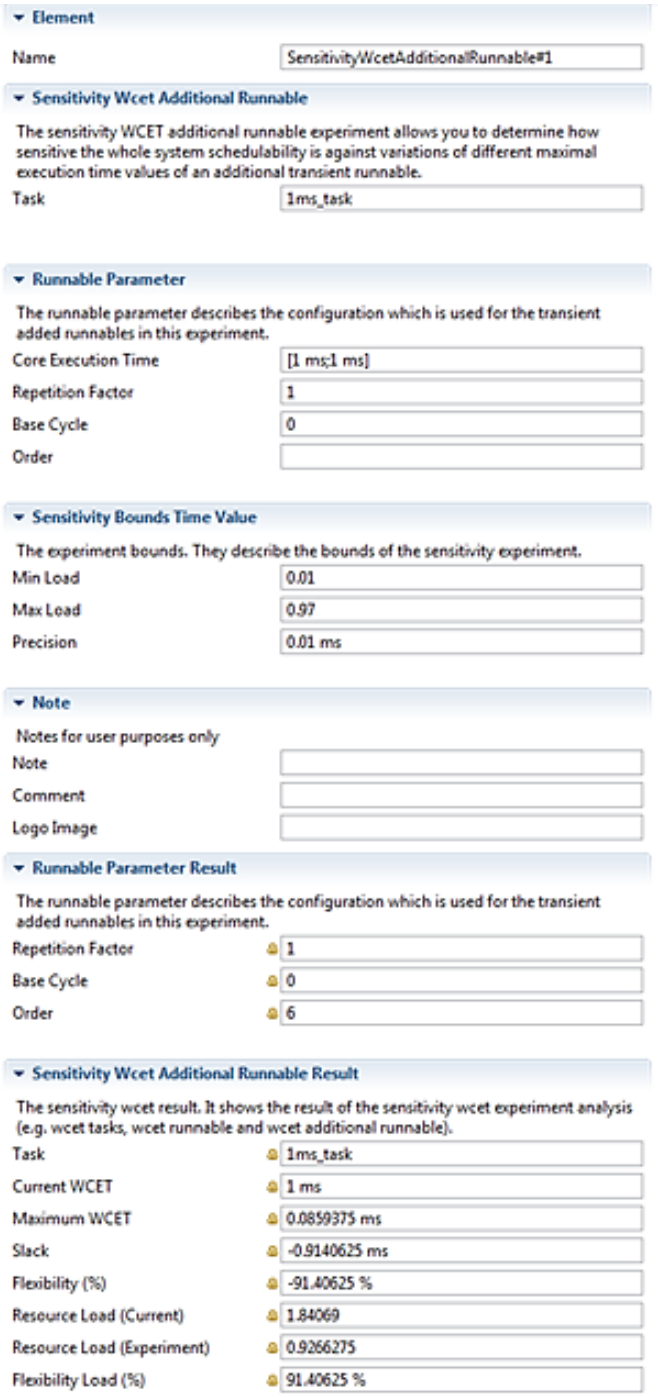


Abbildung 57 Additional Runnable Analysis

Das Experiment wurde wieder nach [63] ausgeführt.

Experiment analyses started...

SensitivityWcetAdditionalRunnableAnalysis for SensitivityWcetAdditionalRunnable#1

Getting experiment config...

Analyzing config (Runnable: R1, ExecutionTime: 1 ms)

Task 1ms_task is overloaded and will be ignored.
Task 10ms_task is overloaded and will be ignored.
Total load of Core is greater than the configured maximal load 0.97.
Task 1ms_task is overloaded and will be ignored.
Could not validate constraint for 1ms_task. Setting the current value as the system bound.
...
Task 10ms_task is overloaded and will be ignored.
Total load of Core is greater than the configured maximal load 0.97.
Task 10ms_task is overloaded and will be ignored.
Could not validate constraint for 10ms_task. Setting the current value as the system bound.
Analyzing config (Runnable: R1, ExecutionTime: 0.125 ms)
….
Experiment bounds (0.0078125 ms) have reached the precision (0.01 ms).
Writing result (Runnable: R1, ExecutionTime: 0.0859375 ms)
Experiment analyses finished.

Die Schritte erfolgen wie im vorherigen Beispiel und nach [63]. Die Execution Time für einen zusätzlichen Runnable würde in diesem Fall ca. 0,086ms betragen. Die Sensitivity WCET Percentaged Analyse sucht für den angegebenen Task die mögliche Ausweitung der „Execution time" in Prozent und gibt an um wieviel er länger brauchen darf. Das Verfahren ist wieder wie bei den vorherigen Beispielen.

Experiment analyses started...
SensitivityWcetPercentagedAnalysis for SensitivityWcetPercentaged#1
Getting experiment config...
Analyzing config (Percent (per mill): 0.01, Tasks: 10ms_task)
...
Analyzing config (Percent (per mill): 0.32, Tasks: 10ms_task)
Total load of Core is greater than the configured maximal load 0.97.
...
Maximum response time constraint at 10ms_task not met (WC analysis). Response time should not exceed 9 ms but is 9.61410796875 ms
...
Analyzing config (Percent (per mill): 0.1842578125, Tasks: 10ms_task)
Experiment bounds (0.008) have reached the precision (0.01).
Writing result (Percent (per mill): 0.1842578125, Tasks: 10ms_task)

Experiment analyses finished.
“

Der 10 ms Task darf in diesem Beispiel seine Execution Time um 18,4% überschreiten. Der Slack ist in diesem Beispiel 17,4 ms auf die vordefinierten 0,01%

Experiment analyses started...
SensitivityWCETAnalysis for SensitivityWcet#1
Getting experiment config...
Analyzing config (Task: 5ms_task, ExecutionTime: 0.182 ms)
...
Analyzing config (Task: 5ms_task, ExecutionTime: 1.456 ms)
Task 10ms_task is overloaded and will be ignored.
Total load of Core is greater than the configured maximal load 0.97.
...
Could not validate constraint for 10ms_task. Setting the current value as the system bound.
Analyzing config (Task: 5ms_task, ExecutionTime: 0.91 ms)
...
Maximum response time constraint at 10ms_task not met (WC analysis). Response time should not exceed 9 ms but is 10.5105 ms
Maximal constraint (9 ms) of task 10ms_task is violated (RT:10.5105 ms).
Analyzing config (Task: 5ms_task, ExecutionTime: 0.75075 ms)
...
Analyzing config (Task: 5ms_task, ExecutionTime: 0.772078125 ms)
Maximum response time constraint at 10ms_task not met (WC analysis). Response time should not exceed 9 ms but is 10.506234375 ms
Experiment bounds (0.0078125 ms) have reached the precision (0.01 ms).
Writing result (Task: 5ms_task, ExecutionTime: 0.77065625 ms)
Experiment analyses finished.

Für den 5ms Task können folgende Ergebnisse zusammengefasst werden:

- Current WCET 0,182ms
- Maximum WCET 0,77ms
- Slack 0,59ms
- Flexibility 323%
- Resource Load (Current) 0,84

- Resource Load (Experiment) 0,96
- Flexibility Load 11,77%

Die Flexibilität kann auch noch durch Diagramme, wie in Abbildung 58 ersichtlich, visualisiert werden:

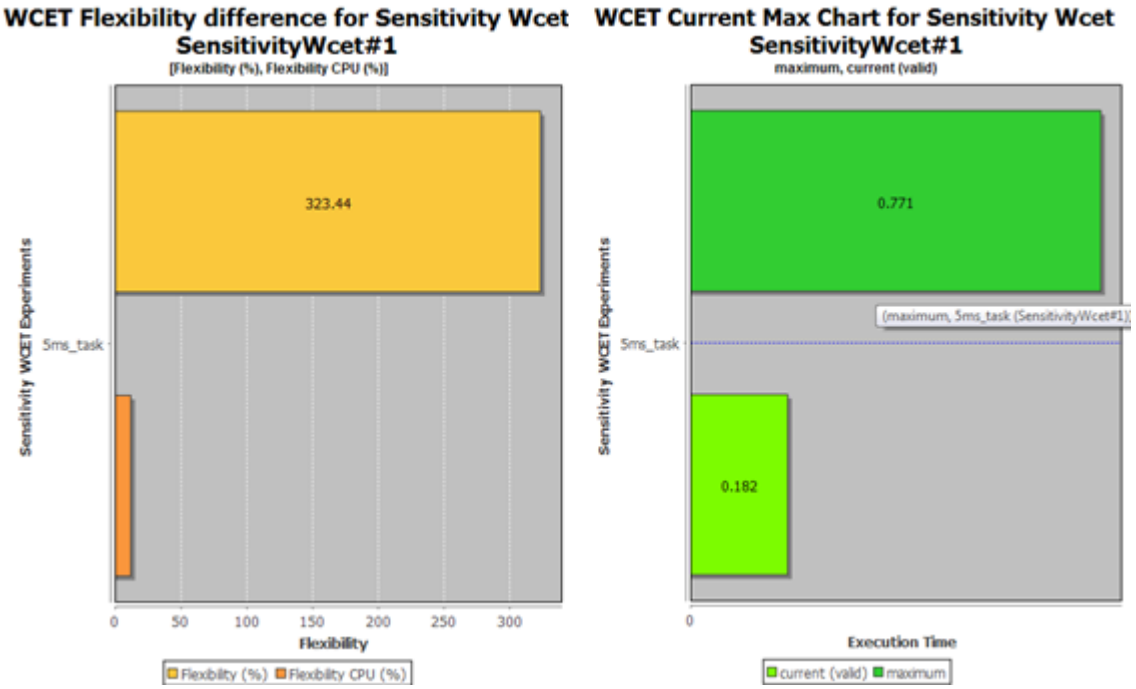


Abbildung 58 Flexibilität der Tasks in Prozent und Absolutwert

Diese Analysen dienen, um herauszufinden, inwiefern Tasks mehrbelastet werden können. Die wichtige Erkenntnis ist, dass 0,59 ms dem Task für andere Aufgaben hinzugefügt werden kann. Dieses Experiment kann jetzt durch “Apply to System“ dem bestehenden System hinzugefügt werden und mit allen anderen, vorherigen Parametern und Systemobjekten neu analysiert werden. Durch das Worst Case Gantt kann dies auch noch veranschaulicht werden
(siehe Abbildung 59). Der Task 10 war einer der kritischen in der Analyse und wird deshalb auch im GANTT Diagramm betrachtet.

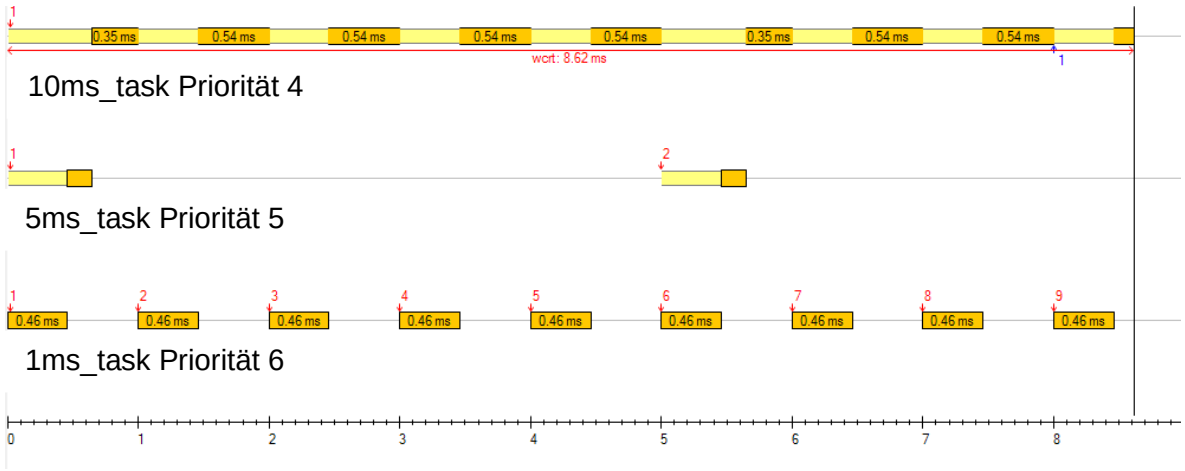


Abbildung 59 GANTT Diagramm

Auch die Gesamtauslastung des Cores bleibt unter den definierten 97% wie in Abbildung 60 ersichtlich:

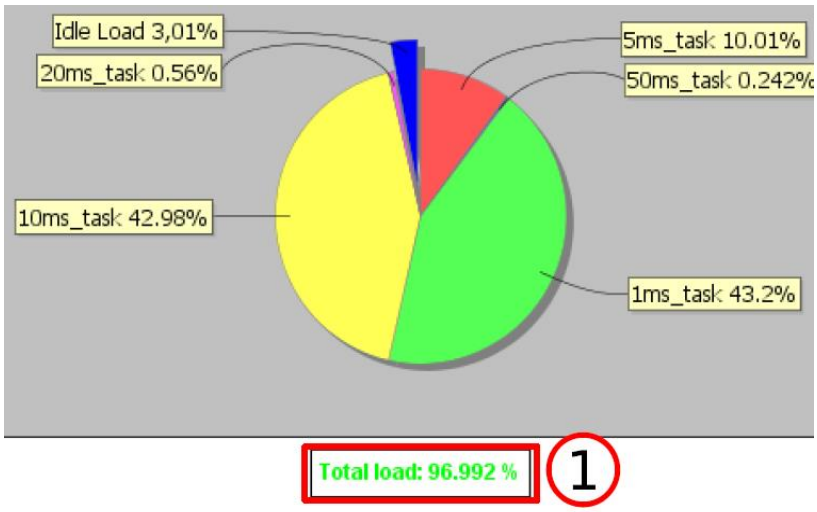


Abbildung 60 Coreauslastung unter 97% (1) [63]

4.9.3 Datenkonsistenz

Ein weiteres Problem, das mittels SymTA/S gelöst werden kann, ist jenes der Datenkonsistenz. Die Datenkonsistenz-Analysebibliothek analysiert im bestehenden System alle Lese- und Schreibezugriffe auf Variablen. Durch die verschiedenen Tasks und Prioritäten kann es hier zu Datenkonsistenzproblemen kommen.

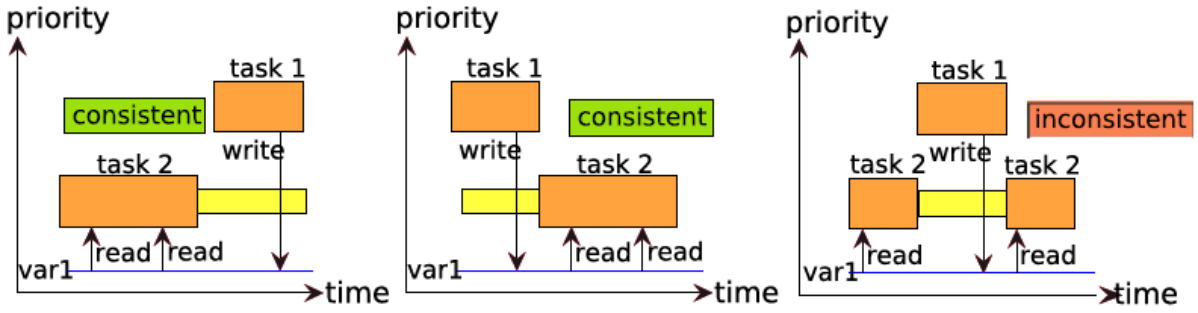


Abbildung 61 Dateninkonsistenz [63]

Wird ein lesender Task von einem höheren schreibenden Task unterbrochen, kann es, wie in Abbildung 61 ersichtlich, zu Dateninkonsistenz kommen. Eine Übersicht über kritische Konstellationen ist in Abbildung 62 ersichtlich.

cases	#reads	#writes	result
1	0	0	OK. Variable is not used.
2	0	1	WARNING: "Variable is never read."
3	0	>1 (by single task)	WARNING: "Variable is never read."
4	0	>1 (by multiple tasks)	ILLEGAL: "Multiple writers."
5	1	0	WARNING: "Variable is never written."
6	1	1	OK. Can not be inconsistent.
7	1	>1 (by single task)	OK. Can not be inconsistent.
8	1	>1 (by multiple tasks)	ILLEGAL: "Multiple writers."
9	>1	0	WARNING: "Variable is never written."
10	>1	1	**The analysis table** is checked for every reader and writer combination (see "analysis table" below).
11	>1	>1 (by single task)	**The analysis table** is checked for every reader and writer combination (see "analysis table" below).
12	>1	>1 (by multiple tasks)	ILLEGAL: "Multiple writers."
13	>1	1 (via different core)	HAZARD: "Writer on core1 can change variable between accesses of reader on core2."

Abbildung 62 Daten Konsistenz-Spezifikationen [63]

Für die Einträge 10 und 11 sind noch weitere Spezifikationen nötig, die dann zu folgenden Erkenntnissen gelangen. Dazu muss das OSEK Analyse Setting die cooperativen Tasks auf fullpreemptive setzen (für den „Parent Core")

		high priority (write access)					
		Task: p	Task: c	Task: np	Run.: p	Run.: c	Run.: np
low priority (read access)	Task: p	hazard	hazard	hazard	hazard	hazard	hazard
	Task: c, s	hazard	special	ok	hazard	ok	ok
	Task: c, ns	hazard	hazard	hazard	hazard	hazard	hazard
	Task: np, s	hazard	hazard	ok	hazard	hazard	ok
	Task: np, ns	hazard	hazard	hazard	hazard	hazard	hazard
	Runnable: p	hazard	hazard	hazard	hazard	hazard	hazard
	Runnable: c, s	hazard	ok	ok	hazard	ok	ok
	Runnable: c, ns	hazard	hazard	hazard	hazard	hazard	hazard
	Runnable: np, s	hazard	hazard	ok	hazard	hazard	ok
	Runnable: np, ns	hazard	hazard	hazard	hazard	hazard	hazard

Abbildung 63 Analyse Tabelle [63]

p: preemptive, c: cooperative, np: non preemptive, s: tasks oder Runnables gehören zur gleichen non-preemption gruppe, ns: tasks oder runnables gehören nicht zur gleichen non-preemption gruppe

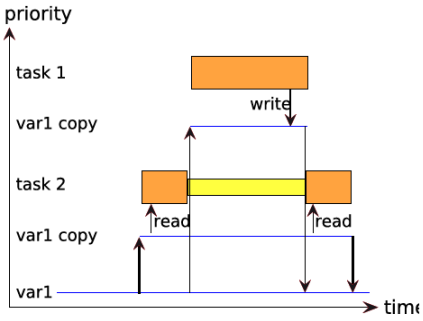


Abbildung 64 Buffern von Variablen [63]

Um die Datenkonsistenz zu bewahren, wird in realen Systemen ein Schutzmechanismus, wie Semaphore [55] oder doppeltes Buffern, realisiert (siehe Abbildung 64). Um dies in SymTA/S wieder zuspiegeln, kann der „Flag" Typ Protected Access gewählt werden. So kann die Analyse diese Variablen beachten und als sicher deklarieren. Für ein Beispiel mit 5 Variablen kann die erste Analyse wie in Abbildung 65 aussehen:

Element		Data Consistency Result						
Name	P...	Variable	Readers	Reading Cores	Writers	Writ...ores	Result	Reason
D...1		Var1	T3 (RTE)	Core1	<empty>	<empty>	W...g	Variable Var1 is never written.
D...2		Var2	T4 (RTE)	Core1	T4 (RTE)	Core1	OK	All accesses to Var2 are consistent.
D...3		Var3	T5 (RTE)	Core1	T5 (RTE)	Core1	OK	All accesses to Var3 are consistent.
D...4		Var4	T4 (RTE)	Core1	T5 (RTE)	Core1	OK	The priority of reader T4 (RTE) is equal or higher than the priority of writer T5 (RTE).
D...5		Var5	T2 (RTE)	Core1	T3 (RTE)	Core1	Hazard	Access to Var5 of reader T2 (RTE) (PREEMPTIVE) can b...ed by higher priority writer T3 (RTE) (PREEMPTIVE).

Abbildung 65 Variablenanalyse kommt noch besseres

In diesem Beispiel gibt es mehrere Bemerkungen. Var1 wird nie beschrieben, Var4 hat einen Reader, der die gleiche oder höhere Priorität wie der Writer hat (daher sicher) und Var5 hat eine potentielle “gefährliche” Situation, da der Writer eine höhere Priorität als der Reader hat. Gelöst wird dieses Problem in diesem Fall ganz einfach durch Zuweisung eines „Writers” zu Var1 und Zusammenfassung der variablen 2 und 3 in eine gemeinsame „NonPräemptiv” Gruppe, wie in Abbildung 66 ersichtlich.

	Element		Task			Critic... Level	Execution Time	Internal	Osek Task Param
	Name	Parents	Read Accesses	Write Accesses	Use Runnables	Criticality level	Core Exe...ion Time	Activation	Task Type
1 T1	T1	Core1	<empty>	Var1(RTE)	IfExist	<empty>	[1 ms;1 ms]	P(5 ms)	Preemptive
2 T2	T2	Core1	3*Var5(RTE)	<empty>	IfExist	<empty>	[1 ms;1 ms]	P(5 ms)	NonPreemptive
3 T3	T3	Core1	3*Var1(RTE)	Var5(RTE)	IfExist	<empty>	[1 ms;1 ms]	P(5 ms)	NonPreemptive
4 T4	T4	Core1	Var2(R...(RTE)	Var2(RTE)	IfExist	<empty>	[1 ms;1 ms]	P(5 ms)	Preemptive
5 T5	T5	Core1	Var3(RTE)	Var3(R...(RTE)	IfExist	<empty>	[1 ms;1 ms]	P(5 ms)	Preemptive
6	ne...k	<empty>	<empty>	<empty>	IfExist	<empty>	<empty>	No Event	<n/a>

Abbildung 66 Lösungen für Dateninkonsistenz

4.9.4 Multicore Daten Konsistenz

Das getestete System besteht aus 3 Cores mit jeweils 2 Tasks, welche auf 6 Variablen schreibend und lesend zugreifen

Element		Data Consistency Result						
Name	P...	Variable	Readers	Reading Cores	Writers	Writ...ores	Result	Reason
D...1		Var1	T1 (RTE)	Core1	T1 (RTE)	Core1	OK	All accesses to Var1 are consistent.
D...2		Var2	T5 (RTE)	Core2	T1 (RTE)	Core1	Hazard	Writer T1 (RTE) on core Core1 can change variable Var2 between accesses of reader T5 (RTE) on core Core2.
D...3		Var3	T3 (RTE)	Core2	T4 (RTE)	Core3	OK	All accesses to Var3 are consistent.
D...4		Var4	T4 ...#1)	Core3	T5 (RTE)	Core2	Hazard	Writer T5 (RTE) on core Core2 can change variable Va...n accesses of reader T4 (CallType#1) on core Core3.
D...5		Var5	T3 (RTE)	Core2	T4 (RTE)	Core3	OK	All accesses to Var5 are consistent.
D...6		Var6	T6 (RTE)	Core3	T6 (RTE)	Core3	OK	All accesses to Var6 are consistent.

Abbildung 67 MC Datenkonsistenz Problem

Folgende Probleme wurden festgestellt:

- Task 1 und 5 können einander beeinflussen, wenn T1 schreibt und den lesenden T5 dabei unterbricht

- Task 5 und 4 können einander beeinflussen, wenn T5 schreibt und den lesenden T4 dabei unterbricht

Um das erste Problem (siehe Abbildung 67) zu lösen, wird T5 einfach zum Core 1 verschoben. Durch die richtige Prioritätensetzung ist dieses Problem behoben. Für das Problem mit Task 5 und 4 kann angenommen werden, dass auf Systemebene eine Lösung mit einem Semaphor gefunden wird. Dies muss dann auch in SymTA/S konfiguriert werden. Dazu muss der „Call Type" der beiden Cores auf diese Variable bezogen auf „Protected Access = true" gesetzt. Somit weiß das Analysetool, dass diese beiden Tasks sich die Variable über einen Semaphore teilen.

5. Zusammenfassung und Ausblick

Um den steigenden Anforderungen im Automobilbereich nachzukommen, ist eine Umstellung auf Multicore Systeme unausweichlich. Die Herausforderungen, die mit einer solchen Umstellung bzw. Umsetzung einhergehen wurden in dieser Arbeit behandelt. Diese waren unter anderem:

- Import von bereits bestehenden Systemen nach SymTA/S
- Einbindung in eine vorhandene Toolchain
- Timing Analysen
- Kommunikationsoverhead
- Timing-Reservenanalyse
- Datenkonsistenz.

5.1 Import von bereits bestehenden Systemen nach SymTA/S

Das Importieren von bereits vorhandenen und entwickelten Systemen ist ein wichtiger Aspekt bei der Umstellung auf Multicore-Systeme. Durch das wiederverwenden bereits existierender Systeme können Zeit und Kosten gespart werden und auch auf die Expertise der Entwickler dieser bereits eingesetzten Software gebaut werden. SymTA/S unterstützt dabei gängige Formate wie FIBEX 2.0.1, DBC, CAN, OSEK, FlexRay, LDF oder AUTOSAR. Weiters bietet die Schnittstelle über Python die Möglichkeit, auch andere und einfachere Formate in SymTA/S einzuspielen. In dieser Arbeit wurde als Informationsquelle eine OIL-Datei verwendet, um die relevanten Informationen in einem System in SymTA/S abbilden zu können. Mittels den Informationen, die in einem OIL-File vorhanden sind, kann schon ein Großteil der nötigen Informationen gewonnen werden. Weitere wichtige Angaben, wie Timing Verhalten (Laufzeit, Offset) eines Tasks, müssen über andere Quellen hinzugefügt werden. Diese können aus einem anderen Dateiformat gewonnen werden oder auch wieder mittels eines Python Skriptes eingelesen werden. Auf jeden Fall ist der Import von Informationen über die Python-Schnittstelle ausreichen um das System aufzubauen. Für genauere Spezifikationen, die für die Analyse notwendig sind, werden weitere Informationen benötigt.

5.2 Einbindung in eine vorhandene Toolchain

Die Entwicklung von Multicore-Systemen ist auch vom Einsatz von vielen Tool (Enterpreise Architect, Matlab-Simulink, AbsInt, …) abhängig. Diese Tools und die damit aufgebauten

Toolchains gilt es natürlich wieder- und weiterzuverwenden, da die Tools meist Lizenzgebühren kosten. Neue Tools und die Zusammenstellung eines neuen toolgesteuerten Entwicklungsprozesses, würde unnötig Kosten verursachen. SymTA/S bietet die Möglichkeit über Shell-Befehle und Python-Scripting sich in andere Systeme oder Toolchains zu integrieren. Diese Methode ist zwar eine recht einfache Methode, benötigt aber ein händisches Erstellen dieser Skripte. Das in dieser Arbeit betrachtete Remote-Interface bietet die Möglichkeit, über Eclipse und einem Java-Plugin, die Verbindung zu bestehenden Toolchains herzustellen. In dieser Arbeit wurde gezeigt, wie dies mit Enterprise Architect funktioniert. Mittels EA können zusätzliche Systeminformationen wie Laufzeiten oder Deadlines gewonnen und somit alle nötigen Einstellungen in SymTA/S getroffen werden. Hierbei dient Eclipse als zentrales Tool um die Informationen aus EA und deren Datenbank zu gewinnen und an SymTA/S weiterzuleiten. Sind diese Informationen von SymTA/S verarbeitet und analysiert, können Informationen auch wieder in Eclipse eingespielt werden und somit an die Datenbank weitergegeben werden.
In Abbildung 68 wird ein Überblick über die vor und Nachteile der Importmöglichkeiten gezeigt. Wie bereits in Kapitel 4.3 beschrieben, macht es unter gewissen Voraussetzungen nicht Sinn, Veränderungen wieder zu exportieren. Veränderungen im Bereich der CPU, des Betriebssystems oder die Erstellung von Ressourcen sind so grundlegend, dass diese besser auf anderen Ebenen manipuliert werden. Auch der Export mittels Python ist nicht empfehlenswert (NR, not recommended)

Element	Toolbridge Remote Interface		Python OIL	
	import	export	import	export
CPU	+	NR	+	NR
OS	+	NR	+	NR
Task/Runnables	+	+	+	NR
- attribute	+	+	+	NR
- timing	+	+	-	NR
Resources	+	NR	+	NR
-attribute	+	+	-	NR
flexible	-		+	
Roundtrip	+		-	

Abbildung 68 Übersicht der Importmöglichkeiten (Not Recomendable, NR)

5.3 Timing Analysen

In dieser Arbeit wurde Timinganalysen für Multicore-Systeme durchgeführt. Basierend auf grundlegenden WCET Analysen von diversen Tasks, wurde die Auslastung des

Gesamtsystems betrachtet. Hierbei können noch Optimierungen und die Aufteilung von Tasks auf anderen Kernen vorgenommen werden. Fokus wurde aber auf die Analyse und den im nächsten Punkt betrachteten Kommunikationsoverheads gelegt.

5.3.1 Kommunikationsoverhead Analysen

SymTA/S kann den Kommunikations-Overhead automatisch von einer XML Datei importieren. Jedoch ist auch wieder eine Importierung durch ein manuell erstelltes Python-Skript möglich. Die im bestehenden System vorhandenen Einheiten wie CPUs und Speicher können dann den zu importierenden Werten zugeordnet werden. SymTA/S bietet die Möglichkeit, die verschiedenen Overheads nach CPUs, Cores, Tasks usw. anzuzeigen. Dadurch kann man den Kommunikations-Overhead besser auf die verschiedenen Cores aufteilen und somit zur Verringerung der Core-Auslastung dienen. Ist der Kommunikations-Overhead im Verhältnis zur Gesamtlast zu hoch, kann versucht werden den lastverursachenden Task oder Runnable oder die Variable auf einen anderen Kern zu legen. Mittels der Datenraten-Metrik kann die Kommunikationslast für jede Variable und jeden Kern bestimmt werden. Hierbei sollte eine Variable mit hoher Datenrate bzw. hoher Kommunikation für einen Kern auch auf dessen lokalen Speicher liegen, um ein schnelleres Zugreifen zu erreichen. Durch eine graphische Darstellung ist es leicht ersichtlich, welche Variablen besser aufgeteilt werden können.

5.3.2 Timing-Reservenanalyse

SymTA/S bietet die Möglichkeit, die freien Ressourcen eines Kernes zu berechnen. Hierbei kann auf der einen Seite berechnet werden, wieviel ein betrachteter Task länger brauchen darf oder wieviel ein Task, der neu hinzugefügt wird, an Rechenzeit verwenden darf. Diese Analysen sind vor allem für neue Funktionen relevant und hilfreich, diese in bestehende Systeme einzugliedern. Die gleiche Analyse ist auch für Runnables verfügbar.

5.3.3 Datenkonsistenz Analyse

Wird ein Task beim Lesen von einem höher priorisierten Task, der auf die gleiche Variable schreibt unterbrochen, kann dies zu Problemen mit der Aktualität der Daten kommen. Diese Probleme können von SymTA/S zwar nicht gelöst, aber zumindest Aufgezeigt werden. Damit keine Situation entsteht, bei der es zu Komplikationen kommen könnte, können Zugriffsregelungen wie ein Semaphore diese Variablen schützen und werden von der Analyse berücksichtigt. Das Problem wird schwieriger, wenn der Zugriff über mehrere Tasks

auf verschiedenen Cores erfolgt. SymTA/S kann auch diese Konflikte erkennen und Lösungen vorschlagen.

5.4. Ausblick

Diese Arbeit hat einen Überblick über die Probleme, die bei der Umstellung eines Single- auf Multicore-System auftreten können, gegeben. Jedes der beschriebenen Probleme ist für sich eine große Herausforderung und kann auf mehrere Varianten gelöst werden. Welche Probleme für das jeweilige System auftreten können, kann mithilfe von SymTA/S festgestellt werden. Für weitere Arbeiten können realistische und reelle Systeme analysiert werden. Da dieses Thema in der Automobilbranche noch relativ neu ist, ist es schwer möglich reelle Daten von Herstellern zu bekommen. Mit realistischen Werten können dann Optimierungen vorgenommen und es kann analysiert werden, wie sich Veränderungen auf das gesamte System auswirken Zukünftige Arbeiten sollten sich mit realistischen Use-Case-Szenarien beschäftigen, um aussagekräftige Resultate zu bekommen. Zum Import dieser umfangreichen Datensätze kann diese Arbeit herangezogen werden. Die Analyse und Bearbeitung eines umfangreicheren Models wird eine große Herausforderung darstellen. Weiters gilt es dann diese simulierten und analysierten Werten mit dem Traceanalyser von Symtavision zu vergleichen, um weitere Optimierungen vorzunehmen

Literaturverzeichnis

[1] M. Hillenbrand. Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen. Herausgegeben von M. Hillenbrand. KIT Scientific Publishing, 2012.

[2] D. Claraz u. a. Introducing Multi-Core at Automotive Engine Systems. In: ERTS 14. ERTSS, 2014

[3] W.A. Günther. Neue Wege in der Automobillogistik, Die Version der Supra-Adaptivität. Springer, 2007. ISBN: 978-3-540-72404-9

[4] Jörg Schäuffele and Thomas Zurawka. Automotive Software Engineering. ATZ/MTZ-Fachbuch. Vieweg + Teubner, Wiesbaden, 4. edition, 2010. ISBN 3834803642

[5] Ault, C., 2013. Medizinische Geräte auf Multi-Core-Prozessoren migrieren. http://www.all-electronics.de/texte/anzeigen/51571/Medizinische-Geraete-auf-Multi-Core-Prozessoren-migrieren [Zugriff 10.10.2015].

[6] AUTOSAR Development Cooperation 2009. http://www.autosar.org/ [Zugriff 25.5.2015]

[7] Wolfgang A. Halang. Herausforderungen durch Echtzeitbetrieb. VDI/VDE-Gesellschaft für Mess- und Automatisierungs-technik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG) Boppard, 3. und 4. November 2011

[8] ETAS GmbH. http://www.etas.com/en/terms_of_use.php [Zugriff 10.11.2014]

[9] B. Duwairi and G. Manimaran. 2003 Combined Scheduling of Hard and Soft Real-Time Tasks in Multiprocessor Systems. Springer-Verlag Berlin Heidelberg New Yor ISBN 3-540-20626-4 k

[10] Bruce Powel Douglass. Design Patterns for Embedded Systems in C. Newnes, Oxford, UK, 1. edition, 2011. ISBN 9781856177078.

[11] Infineon. Application Kit TC2X5 User's Manual. http://www.infineon.com/dgdl/Infineon-Tricore+Family+BR_2015-BC-v01_00-EN.pdf?fileId=db3a30431f848401011fc664882a7648 Zugriff [6.7.2015]

[12] B. Hardung, T.Kölzow und A.Krüger. Reuse of software in distributed embedded automotive systems. In: Proceedings of the fourth ACM international conference on Embedded software, EMSOFT '04, Seiten 203–210, Pisa, Italien, September 2004. Association for Computing Machinery (ACM).

[13] MCKinsey & Company und Institut für Produktionsmanagement, Technologie und werkzeugmaschinen (PTW) TU Darmstadt: HAWK 2015 - Herausforderung automobile Wertschöpfungskette. Wissensbasierte Veränderung der automobilen Wertschöpfungskette, Band 30 der Reihe Materialen zur Automobilindustrie. Verband der Automobilindustrie (VDA), 2003

[14] Werner Zimmermann, Ralf Schmidgall. Bussysteme in der Fahrzeugtechnik. 2. Auflage 2007, ATZ/MTZ-Fachbuch, ISBN 978-3-8348-0235-4

[15] OSEK. OSEK/VDX Operating System. Version 2.2.3 February 17th, 2005

[16] Evidence.ERIKA Enterprise. Erika Enterprise Reference Manual v. 1.4.5 http://erika.tuxfamily.org/drupal/documentation.html

[17] B. P. Douglass. Design Patterns for Embedded Systems in C. Seite 84. Newnes, 2011. ISBN 9781856177078.

[18] R. Hilbrich,et al. Modellbasierte Generierung statischer Schedules für sicherheitskritische, eingebettete Systeme mit Multicore-Prozessoren und harten Echtzeitanforderungen. In: Herausforderungen durch Echtzeitbetrieb. W. A. Halang. Informatik aktuell. Springer, 2012, Seiten 29–38. isbn: 978-3-642-24657-9.

[19] P. Radojkovic, et al. On the Evaluation of the Impact of Shared Resources in Multithreaded. COTS Processors in Time-critical Environments. In: ACM Trans. Archit. Code Optim. 8.4 (Jan. 2012), http://doi.acm.org/10.1145/2086696.2086713.

[20] M. Paulitsch, J. Nowotsch. Leveraging Multi-core Computing Architectures in Avionics. In: Dependable Computing Conference (EDCC), 2012 Ninth European.

[21] Florian Kluge, et al. Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processo. Department of Compute Science University of Augsburg. 2009

[22] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In IEEE Real-Time Systems Symposium, IEEE Computer Society, 2001.

[23] P. Baker. A stack-based resource allocation policy for realtime processes. In IEEE Real-Time Systems Symposium 1990.

[24] International Organization for Standardization / International Electrotechnical Commission ISO/IEC: 15504-1: Concepts and introductory guide. 1998.

[25] IABG Industrieanlagen-Betriebsgesellschaft mbH. Das V-Modell – Entwicklungsstandard für IT-Systeme des Bundes. 1997/2005. www.v-modell.iabg.de [Zugriff 6.7.2015]

[26] M. Cohn: Succeeding with Agile, Addison-Wesley, 2010.

[27] Symtavision OSEK Manual Version 3.6.0

[28] G. Macher, C. Kreiner, et al. Automotive Real-time Operating Systems: A Model-Based Configuration Approach, 2014

[29] Jane W. S. Liu: Real-Time Systems . Prentice Hall, 2000.

[30] T. Kilian. Scripting Enterprise Architect, 2015 *samples.leanpub.com/ScriptingEA-sample.pdf*

[31] Emil Michta. Scheduling Systems, http://eu.wiley.com//legacy/wileychi/hbmsd/articles.html [Zugriff: 8.9.2015]

[32] K. Schneidemann, et. al. Load Balancing in AUTOSAR-Multicore-Systemen, 2010

[33] M. Jersak, K. Richter, H. Sarnowski, P. Gliwa Laufzeitanalysen zur frühzeitigen Absicherung von Software

[34] A. Monot, et al. Multicore scheduling in automotiv ECUs, 2010, http://nicolas.navet.eu/publi/ertss_2010.pdf [Zugriff 31.10.2015]

[35] Sparx Systems. http://www.enterprisearchitect.at/ [Zugriff 20.10.2015]

[36] OSEK / VDX. System Generation. OIL: OSEK Implementation Language Version 2.5 2004. http://www.osek-vdx.org

[37] Evidence. ERIKA Enterprise. Tutorial: RT-Druid and OIL basics. http://erika.tuxfamily.org/wiki/index.php?title=Tutorial:_RT-Druid_and_OIL_basics [Zugriff 10.10.2014]

[38] J. Schneider, M. Bohn, R. Rößger. Migration of Automotive Real-Time Software to Multicore Systems: First Steps towards an Automated Solution. 22nd EUROMICRO Conference on Real-Time Systems, Brussels,Belgium, 2010

[39] B. Moyer. Real World Multicore Embedded Systems. Newnes 2013

[40] S. Ding, et.al. Based Scheduling Method for FlexRay Systems. Nagoya University. Furo-cho, Chikusa-ku, Nagoya, Japan

[41] Symtavision. SymTA/S. Analysis Introduction and Theory, Version 3.6.0

[42] Evidence. ERIKA Enterprise. http://erika.tuxfamily.org/drupal/features-and-beneficts.html Zugriff [2.8.2015]

[43] IT-Designers Gruppe. Das OSEK Betriebssystem ERCOSEK. http://www.it-designers-gruppe.de/uploads/media/OSEK_Ercosek.pdf Zugriff 6.9.2015

[44] R.Davis, N. Tracey. Impact Case Study: The world's smallest automotive real-time operating system. http://www-users.cs.york.ac.uk/~robdavis/papers/ImpactCaseStudyRTOS.pdf

[45] Mario Cupelli PXROS-HR for TriCore, The safety platform for embedded systems, 2011 https://www.hightec-rt.com/en/downloads/pxros-hr/50-pxros-hr-training-1/file.html

[46] HighTec EDV-Systeme GmbH https://www.hightec-rt.com

[47] A. Dold, M. TrappHeruasforderungen und Erfahrungen eines OEM bei der Gestaltung sicherheitsgerechter Prozesse. http://cs.emis.de/LNI/Proceedings/Proceedings110/gi-proc-110-088.pdf

[48] ISO 26262-1:2011. http://www.iso.org/iso/catalogue_detail?csnumber=43464

[49] GLIWA, http://www.gliwa.com/ [Zugriff 13.10.2015]

[50] Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V. http://www.fraunhofer.de/

[51] R. Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th, pages 5.E.3–1 –5.E.3–11, oct. 2010.

[52] John Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance, 1999.

[53] Frauenhofer Institut Mit ERNEST frühzeitig Fehler im Softwareentwurf finden. http://www.esk.fraunhofer.de/de/projekte/ERNEST.html

[54] EAST-ADL http://www.east-adl.info/

[55] Andrew S. Tannenbaum. Moderne Betriebssysteme Pearson Education Deutschland GmbH, 2002

[56] DDC-I, Cache Partitioning Enables Multi-Core Usage. 2013b

[57] DDC-I. Unleashing the Power of Multi-Core Processors in Safety-Critical Software Webinar. 2010 http://www.ddci.com/downloadsRequest.php?thing=multicore-webinar.mp4&type=media.

[58] Moyer, B., Embedded Systems Real World Multicore Embedded Systems A Practical Approach Expert Guide, .

[59] Mundhenk, P., Operating Systems for Multicore Systems.

[60] RTaW. REALTIME-AT-WORK, France. http://www.realtimeatwork.com/

[61] Timing-Architects™, Embedded Systems GmbH, http://www.timing-architects.com

[62] Symtavision. SymTA/S. SymTA-S_Manual_Interfaces, Version 3.6.0

[63] Symtavision. SymTA/S. SymTA-S_Manual_Experiment, Version 3.6.0

[64] M. Wernicke, J.Rein, Entwicklung von Steuergerätesoftware nach AUTOSAR in Elektronik Informationen 11-2006 http://vector.com/portal/medien/cmc/press/Vector/AUTOSAR_ElektronikInformationen_200611_PressArticle_DE.pdf

[65] ETAS GmbH, http://www.etas.com/de/products/rta_osek.php [Zugriff 20.10.2015]

[66] I.Bruse, P.Gula, S.Eberle.„Artop“ – Neue und offene Plattform für Autosar-Tools, ATZelektronik 01I2009 Jahrgang 4

[67] H. Brock, J. Kalmbach. AUTOSAR goes Multicore – mit Sicherheit, vector.com Juli 2014

[68] G. Macher, A. Höller, et. al. „Automotive Embedded Software: Migration Challenges to Multi-Core Computing Platforms“. 2015

Abkürzungsverzeichnis

ABS	Antiblockiersystem
ACC	Abstands-Regel-Tempomat (engl. Adaptive-Cruise-Control)
AD	Analog Digital
AMP	Asymmetrischen Multiprozessorsystemen
API	Application programming interface (‚Anwendungs-programmier-schnittstelle)
ARXML	Inputdateiendung für die RTE
ASC	American Standard Code
Asil	Automotive Safety Integrity Level
Autosar	AUTomotive Open System ARchitecture
BSW	Basissoftware
CAN	Controller Area Network
CAPA	Criticality As Priority Assignment
CET	Core Execution Time
CDDs	Complex Device Drivers
COTS	Commercial off-the-shelf
CPU	Central Processing Unit,
DMA	Direct Memory Access
EA	Enterprise Architect
EAST-ADL	Architecture Description Language
ECU	Engine Control Unit
EDF	Earliest Deadline First
EE	Eletrik Elektronik
ERNEST	Framework for the EaRly verification and validation of Networked Embedded SysTems
ESP	Elektronisches Stabilitäts Programm
FIFO	First In, First Out
GCS	Gglobal critical section
HAL	Hardware Abstraction Layer
HW	Hardware
IDE	Integrated development environment
IO	Input Output
IR	Interrupt
LIN	Local Interconnect Network
LED	Licht-emittierende Diode

MCAL	Microcontroller Abstraction Layer MCAL
MPCP	Multiprocessor Priority Ceiling Protocol,
MSRP	Multiprocessor Stack Resource Policy
NUMA	Non-Uniform Memory Access
NR	Not Recomendable
ORTI	OSEK Runtime Interface
OS	Operating System
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
PCP	Priority Ceiling Protocol,
RAM	Random Access Memories.
RMS	Rate Monotonic Scheduling
Rom	Rread-only memory
RTE	Runtime environment
SMP	Symmetrischen Multiprozessorsystemen
SOAP	Simple Object Access Protocol
SW	Software
SWC	Softwarecomponente
URL	Uniform Resource Locator
VAR	Variable
VDX	Vehicle Distributed Executive
VFB	Virtual Functional Bus
WC	Worst Case
WCET	Worst case execution time
WCRT	Worst case response time
XML	Extensible Markup Language

Abbildungsverzeichnis