Robustes Transaktionsbasiertes EDF Scheduling für hybride Realzeitsysteme

Benito Liccardi

Lehrstuhl für Realzeit-Computersysteme

Robustes Transaktionsbasiertes EDF Scheduling für hybride Realzeitsysteme

Benito Liccardi

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor–Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.–Prof. Dr.–Ing. J. Eberspächer
Prüfer der Dissertation:	1. Univ.–Prof. Dr.–Ing. G. Färber
	2. Univ.–Prof. Dr. sc.techn. (ETH) A. Herkersdorf

Die Dissertation wurde am 19.08.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 18.04.2006 angenommen.

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die zum Entstehen dieser Arbeit beigetragen haben.

In fachlicher Hinsicht gilt mein Dank *Herrn Prof. Färber*, der mir die Möglichkeit zur Durchführung dieser Dissertation gegeben hat. Seine Betreuung und Unterstützung sind mir stets eine große Hilfe gewesen.

Ebenfalls möchte ich *Herrn Prof. Herkersdorf* für die Übernahme des Zweitgutachtens meiner Arbeit danken.

An die praktische Umsetzung der theoretischen Ansätze ist ohne studentische Unterstützung nicht zu denken. Daher möchte ich mich an dieser Stelle besonders bei meinen zwei Diplomanden, Maximilian Obermaier und Holger Lohn, für ihre hartnäckige Umsetzung meiner Konzepte bedanken.

Des Weiteren gilt mein Dank denjenigen Kollegen, die in vielen fachlichen und außerfachlichen Diskussionen für die entscheidenden Denkanstöße mitverantwortlich waren.

Der besondere Dank gilt jedoch meinen Eltern, Sofia und Italo Liccardi, die mir durch ihre uneingeschränkte Unterstützung und dem fortwährenden Vertrauen, meine Ausbildung und somit diese Arbeit erst ermöglicht haben.

Ebenso möchte ich mich bei meiner Frau Cornelia bedanken, die mit liebevoller Ausdauer und Verständnis mir während dieser Zeit zur Seite gestanden hat. Meinem Sohn Constantin Claudio danke ich für seine mir zu jeder Uhrzeit entgegengebrachte Freude, mit der er einen wesentlichen Beitrag zur Fertigstellung dieser Arbeit leisten konnte.

München, im August 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der verwendeten Symbole

TEDF

Γ_i	Transaktion (allgemein)
C_i	Worst Case Transaktionsausführungszeit
D_i	relative Transaktionsdeadline
E_{F_i}	Transaktionsereignisfunktion
I_{F_i}	Intervallfunktion
E_S	Ereignisstrom
T_i	Transaktionsgraph mit allen möglichen Transaktionspfaden
$\boldsymbol{A}$	Menge aller Threadaktivierungen
$\boldsymbol{E}$	Menge aller Threadevents
$A_{m,k}$	Threadaktivierung des Threads m durch das aktivierende Ereignis E_k
E_k	aktivierendes Ereignis
$p_{i,j}$	Ein Pfad aus dem Transaktionsgraphen Γ_i
$\boldsymbol{P_i}$	Menge aller möglichen Transaktionspfade
EAM	Ereignisabhängigkeitsmatrix
$\tau_{i,j}$	Transaktionsinstanz
d_i	absolute Deadline einer Transaktionsinstanz
$\boldsymbol{S_i}$	Menge aller verwendeten Ressourcen
S_i	Ressource
$z_{i,k}$	k–te kritische Abschnitt einer Transaktion i
$c_{i,k}$	Rechenzeit im kritischen Abschnitt $z_{i,k}$
π_i	Unterbrechungsebene
$\beta^d_{k,m}$	Menge der kritischen Abschnitte mit direkter Blockierung
$\beta^p_{k,m}$	Menge der kritischen Abschnitte mit *push through* Blockierung
B_i^β	Blockierungszeit (kritische Abschnitte)
$\zeta_{k,m,l}$	Menge der kritischen Abschnitte, die durch Belegung der Ressource S verursacht werden.
B_i^ζ	Blockierungszeit (Ressourcen)
B_i	Gesamtblockierungszeit
U	Gesamtrechenzeitbedarf
$D_i(I)$	Rechenzeitanforderungsfunktion einer Transaktion
I	Intervall

RTEDF

$\boldsymbol{T}$	Menge aller Transaktionen (Transaktionsset)
$\Gamma_{hrt,i}$	harte Transaktion
$\Gamma_{qos,i}$	*QoS*–Transaktion
$\Gamma_{soft,i}$	*Soft*–Transaktion
λ	Pattern für *QoS*–Transaktionen
v_i	maximale Anzahl an konsekutiven *Delta*–Instanzen
Δ_i	Deadlineerweiterung um Intervall Δ_i
f_i	konsekutive harte Instanzen nach *Delta*–Instanzen
ψ	Pattern für *Soft*–Transaktionen
$\boldsymbol{H}$	Menge aller harten Transaktionen
$\boldsymbol{Q_{qos}}$	Menge aller *QoS*–Transaktionen
$\boldsymbol{Q_{soft}}$	Menge aller *Soft*–Transaktionen
L_i	Laxity
$E_{F_{qos}}$	Transformierte Ereignisfunktion einer *QoS*–Transaktion
$E_{F_{soft}}$	Transformierte Ereignisfunktion einer *Soft*–Transaktion
E_{F_γ}	Ereignisfunktion für die Umschaltphase in den Überlastmodus
E_{F_δ}	Ereignisfunktion für die Phase der *Delta*–Instanzen
E_{F_σ}	Ereignisfunktion für die Phase der *Firm*–Instanzen

Systemanalyse

$\boldsymbol{N}$	Menge aller Transaktionsinstanzen
$\boldsymbol{N_{delta}}$	Menge aller *Delta*–Transaktionsinstanzen
$\boldsymbol{N_{firm}}$	Menge aller *Firm*–Transaktionsinstanzen
$\boldsymbol{N_{qos}}$	Menge aller *QoS*–Transaktionsinstanzen
$\boldsymbol{N_{skip}}$	Menge aller verworfenen Transaktionsinstanzen
$\boldsymbol{N_{meet}}$	Menge aller *Delta*–Instanzen, deren Reaktionszeit innerhalb der primären Deadline geblieben ist.
Q_{kfirm}	gleichgewichteter Mittelwert für konsekutive *Firm*– und *QoS*–Instanzen
$Q^{\sim}_{\text{kfirm}}$	auf den Parameter f_i normierte Größe Q_{kfirm}
Q_{kdelta}	gleichgewichteter Mittelwert für konsekutive *Delta*–Instanzen
Q_{kdelta}	auf den Parameter v_i normierte Größe Q_{kdelta}
Q_{krel}	relativer Anteil an nicht erweiterten Transaktionen einer Simulation
$Q^{\sim}_{\text{krel}}$	gleichgewichtete Mittelung aller Q_{krel} Werte eines Transaktionssets
Q_{meet}	relativer Anteil an Transaktionen, die unterhalb ihrer primären Deadline geblieben sind
$Q^{\sim}_{\text{meet}}$	gleichgewichtete Mittelung aller Q_{meet} Werte eines Transaktionssets
$\rho^{\sim}_{\text{demand}}$	mittlere angeforderte Rechenzeit pro Simulatinsintervall für ein Transaktionsset
$\rho^{\sim}_{\text{rtedf}}$	mittlere realisierte Rechenzeit pro Simulationsintervall

Zusammenfassung

Die steigende Leistungsfähigkeit von eingebetteten Systemen führt zu einer stärkeren Integration von unterschiedlichen Softwaremodellen. Die Unterschiede manifestieren sich in der Einhaltung der Zeitanforderungen (hart, *QoS*, weich), den unterschiedlichen Aktivierungsmodellen (periodisch, aperiodisch, sporadisch) und der Variation von Ausführungszeiten sowohl durch datenabhängige Softwareaktivierungen als auch durch auf Durchsatz optimierte Hardwarearchitekturen. Für diese integrierten Softwaremodelle werden des Weiteren effiziente und realistische Nachweisverfahren benötigt, die eine Einhaltung der geforderten zeitlichen Eigenschaften von harten, weichen und auch *QoS* Modellen a priori garantieren.

Bestehende Laufzeitsysteme sind jeweils auf einzelne Teilbereiche dieser Anforderungen spezialisiert und daher für eine Integration dieser Softwaremodelle entweder gar nicht oder nur bedingt geeignet.

In dieser Arbeit wird ein robustes und transaktionsbasiertes Laufzeitsystem (RTEDF) vorgestellt, dass diesen Anforderungen Rechnung trägt.

- Die transaktionsbasierte Laufzeitsemantik (TEDF) basiert auf dem optimalen dynamischen EDF Scheduling und sorgt beim Nachweisverfahren für einen realistischen Realzeitnachweis. Zudem wird zur Laufzeit eine spezifikationsgerechte Berücksichtigung der dynamischen Prioritäten gewährleistet.
- Die robuste Erweiterung (RTEDF) überprüft zur Laufzeit den aktuellen Rechenzeitbedarf des Systems. Im Unterlastfall werden die zur Laufzeit entstehenden Slack Zeiten (durch Variation von Ausführungszeiten, nicht entstehenden Blockierungszeiten und Variation von Ereignisabständen) zur Erzielung einer hohen *Qualität* von *QoS*–/*Soft*–Transaktionen genutzt. Der Scheduler verhält sich wie ein EDF Scheduler. Im Überlastfall hingegen wird für diese Taskklassen eine vorab definierte Reduktion der Qualität mit vorab spezifzierten und im Nachweis berücksichtigten *QoS*–Pattern angewendet, so dass ein robustes Verhalten gewährleistet ist.
- Die Spezifikation der *QoS*–Pattern erfolgt als Metadaten im Softwaremodell. Die Einhaltung und Berücksichtigung dieser Metadaten führt zu einem parallelen Meta–Entwurfsfluss.
- Die in dieser Arbeit betrachtete Auslagerung der Überlastüberwachung (RTEDF) auf einen *loosely coupled* Coprozessor führt zu einer hardwaretechnischen Trennung, die eine Entkopplung des Meta–Entwurfsflusses vom funktionalen Entwurf ermöglicht.

Das TEDF Laufzeitsystem und der RTEDF Algorithmus wurden prototypisch auf einer SOPC (System–on–a–Programmable–Chip) Architektur mit Realzeitbetriebssystem und Soft–Core Prozessor implementiert. Es werden die Simulationsergebnisse des Verfahrens mit Hilfe eines System Simulators ebenso wie die Laufzeiten– und Datenbedarfsmessungen der realen Implementierung präsentiert.

1 Einleitung

Eingebettete Realzeitsysteme sind dadurch charakterisiert, dass sie in extremen Wechselwirkungen mit den fachlichen Domänen stehen, in denen sie zum Einsatz kommen — Sie sind „Dienstleister“ und gleichzeitig innovationstreibende Kraft für die Realisierung zukunftsträchtiger Entwicklungen. Ihre Anwendung zieht sich wie ein roter Faden quer durch verschiedene Technologien und spiegelt somit die besondere Rolle wider, die eingebettete Systeme als „Aspekttechnologie“ [1)] besitzen.

Die Relevanz dieser Technologien und somit indirekt auch die Relevanz der treibenden eingebetteten Systeme für die nächsten Jahre wird auch in einer Studie des BMBF [3] verdeutlicht, in der internationale Technologieprognosen verschiedener Nationen für einen Zeitraum von 5 bis 15 Jahren miteinander verglichen werden. Die Hauptbereiche Produktions– und Prozesstechnik, Informations– und Kommunikationstechnik, Elektronik und Medizintechnik werden von allen führenden Industrienationen als Zukunftstechnologien bezeichnet. Jeder dieser Bereiche wäre bereits heute ohne den Einsatz eingebetteter Systemtechnologie nicht denkbar.

Diese Interdisziplinarität führt dazu, dass meistens keine allgemeine optimale eingebettete Systemlösung, die allen Anforderungen genügt, existiert. Vielmehr sind speziell zugeschnittene Lösungen gefragt, die die jeweiligen Aufgabenstellungen bestmöglich abdecken können.

Der Entwickler dieser Systeme sieht sich daher besonderen Herausforderungen gegenübergestellt: Er muss sowohl mit den Algorithmen für die Lösung der Aufgaben in dem zu bedienenden Fachbereich vertraut sein, als auch die passende Hardware–, Laufzeit– und Softwaresystemarchitektur bereitstellen können.

In einer Vordringlichen Aktion mit dem Thema „Entwicklung, Produktion und Service von Software für eingebettete Systeme in der Produktion“[18], wird analysiert, wie der Softwareentwurfsprozess der einzelnen Unternehmen durchgeführt und begleitet wird. Die Ergebnisse der Befragung bei kleinen und mittleren Unternehmen zeigen, dass vielfach die Softwareentwicklung ohne grundlegende Methodik praktiziert wird. Vergleichbare Resultate werden auch in der Problemanalyse „SOFTBED: Problemanalyse für ein Großverbundprojekt, Systemtechnik Automobil – Software für eingebettete Systeme“[1] festgestellt. Dabei werden insbesondere das Fehlen der Durchgängigkeit der Werkzeugkette, die fehlende Modellierungsmöglichkeit zeitabhängiger Informationen und die fehlende Entwicklungsmethodik als Defizite hervorgehoben.

[1)] Der Begriff ist aus der Aspektorientierten Programmierung entliehen, in der das „Verweben“ von Funktionalität quer durch Softwaremodule und –objekte betrachtet wird.

1.1 Anforderungen an eingebettete Systeme

Der Anteil der Software in Fahrzeugen und in eingebetteten Systemen im Allgemeinen steigt stetig an. Dabei nimmt die Software einen immer größeren Stellenwert gegenüber der Hardware ein. Dieser Trend ist jedoch vor allem dem Fortschritt auf der Hardwareseite zu verdanken. Nicht zuletzt aufgrund der angestiegenen Rechenleistung und applikationsspezifischen Erweiterungen (Hardware Coprozessoren zur Objekterkennung und Instruktionssatzerweiterungen für die digitale Signalverarbeitung) ist die Realisierung zahlreicher Funktionalitäten in Software überhaupt erst möglich geworden.

Der Anstieg des Softwareanteils an den Gesamtlösungen führt jedoch in Verbindung mit den bestehenden Defiziten in der Entwurfsmethodik zu neuen Problemen, die eine effiziente Softwareentwicklung behindern. Dazu tragen insbesondere folgende Faktoren bei:

- Höhere Integrationsdichte von Software aufgrund stetig steigender Rechenleistung,
- verschiedene Softwaremodelle auf einem Kern (**reaktive** Modelle für ereignisgesteuerte Anteile und **transformierende** Modelle für daten– und signalverarbeitende Anteile),
- verschiedene Abarbeitungsmodelle mit verschiedenen Anforderungen (harte, *Quality of Service*, weiche oder keine Realzeitanforderungen),
- periodische, aperiodische und sporadische Systemaktivierungen und
- Software mit größeren Laufzeitschwankungen aufgrund komplexer Algorithmen (z. B. Point of Interest (Bildverarbeitung))

Daraus ergeben sich für das Laufzeitsystem und die Werkzeugunterstützung spezielle Anforderungen im Hinblick auf die Entwicklung und Spezifikation von Software für eingebettete Systeme.

1.1.1 Laufzeitsystem

Die Anforderungen, die an ein Laufzeitsystem mit den oben genannten Rahmenbedingungen gestellt werden, setzen sich sowohl aus einer optimalen Nutzung der Ressource CPU als auch aus einer applikationsspezifischen Realisierung von Laufzeiteigenschaften zusammen. Das bedeutet, dass effektive Strategien benötigt werden, um die Systemdynamik durch variierende Rechenzeiten einzelner Tasks mit variierenden Aktivierungszeitpunkten bei ereignisgesteuerten Systemen effizient verwalten zu können. Zu diesen Anwendungen zählen beispielsweise aktive Realzeitdatenbanken. Aktivitäten werden hier nur dann angestoßen, wenn bestimmte Bedingungen erfüllt sind. Damit hängt das Verhalten stark vom Systemzustand ab und wird in der Regel keinem periodischen Muster unterliegen.

Hinzu kommt, dass nicht nur Tasks mit harten sondern auch solche mit *Quality of Service* Realzeitanforderungen auf einem Prozessorkern abgearbeitet werden müssen. Diese Abarbeitung sollte auf der einen Seite so effizient wie möglich sein, auf der anderne Seite jedoch zu keiner Zeit zu einer Deadlineverletzung der harten Realzeittasks führen. Der Anteil der Software, bei

der es bei einer Deadlineüberschreitung zu katastrophalen Folgen kommen kann, ist in der Praxis relativ klein[2]. Der Großteil der Software führt bei Deadlineüberschreitungen lediglich zu Qualitätseinbußen. Dies bedeutet jedoch auch, dass diese Softwareart nicht frei von bestimmten Zeitanforderungen ist. Diese Anwendungen, die *partiell*[3] harte Zeitanforderungen besitzen, werden im Folgenden unter dem Begriff *Quality of Service* Software subsummiert, die durch eine bestimmte Spezifikationsbandbreite für das Realzeitverhalten beschrieben wird. Eine Systemauslegung mit existierenden Ansätzen für harte Realzeitsysteme würde bei diesen Systemen zu überdimensionierten Systemarchitekturen führen. Gängige Auslegungsverfahren sind daher nur eingeschränkt für hybride Systeme anwendbar.

Statistische Verfahren hingegen eignen sich für Systeme, in denen Aussagen über durchschnittliche Metriken (Antwortzeit, Rechenzeitanforderung) zur funktionalen Verifikation ausreichend sind. Harte und *QoS* Realzeitaufgaben können mit diesen Verfahren jedoch nicht zuverlässig modelliert werden.

1.1.2 Werkzeugunterstützung

Das Einbeziehen neuer Technologien in den Entwurfsprozess führt vielfach zu einer Erhöhung der Komplexität in den einzelnen Entwurfsphasen[4]. Daher erfordert die Anbindung eines neuen Laufzeitsystems an einen Softwareentwurfsprozess ein den Entwickler unterstützendes Rahmenwerk. Das Ziel ist es dabei, den Softwareentwurfsprozess soweit wie möglich frei von Fragestellungen der Systemarchitektur und der Laufzeitsystemparameter zu halten. Ein ideales Framework muss daher sowohl die Extraktion von Modellparametern aus dem Systemmodell als auch einen automatisierten Realzeitnachweis sowie die anschließende Konfiguration des Laufzeitsystems ermöglichen.

1.2 Zielsetzung und Aufbau dieser Arbeit

Das Ziel dieser Arbeit ist es, ein erweitertes transaktionsbasiertes Schedulingverfahren (*Transaktionsbasiertes EDF*) zu entwickeln, das sowohl einen effizienten Realzeitnachweis durch Berücksichtigung von Präzedenzrelationen als auch eine Spezifikation von zeitlichen Anforderungen für harte, *QoS*– und *Soft*–Tasks gleichermaßen ermöglicht. Im Unterlastfall sollen die zur Laufzeit entstehenden Slack Zeiten (durch Variation von Ausführungszeiten, nicht entstehenden Blockierungszeiten und Variation von Ereignissabständen) zur Erzielung einer hohen *Qualität* von *QoS*–/*Soft*–Tasks genutzt werden. Im Überlastfall hingegen soll für die letztgenannten Taskklassen eine vorab definierte Reduktion der Qualität mit *QoS*–Pattern angewendet

[2] Sicherheitskritische Systeme sind redundand ausgelegt, so dass selbst in diesem Fall eine Deadlineüberschreitung nicht zur Katastrophe führen darf.

[3] Partiell bedeutet in diesem Zusammenhang, dass ein bestimmter Teil der Ausführungen der Tasks mit einer verlängerten Deadline abgearbeitet werden darf während der restlichen Teil harten Zeitanforderungen unterliegt.

[4] Ein Beispiel wäre hier die Anwendung der formalen Verifikation von Spezifikationen, die in der Praxis allerdings selten zum Einsatz kommt.

werden, so dass ein robustes Verhalten gewährleistet ist. Dabei soll mit *QoS*–Pattern definiert werden, wie oft hintereinander eine Task mit einer verlängerten Deadline (*Delta*) bearbeitet werden darf, um anschließend wieder mit einer Mindestanzahl an Ausführungen mit ihrer primären Deadline (Firm) abgearbeitet zu werden.

Sowohl die Slacknutzung als auch die Umschaltung in den Überlastmodus sollen mit einer Überlastüberwachung, die aufgrund des dynamischen Schedulings zur Laufzeit durchgeführt wird, realisiert werden. Es soll des Weiteren eine implementierungstechnische Realisierung der Überlastüberwachung auf einem parallel laufenden Coprozessor untersucht werden.

Das in dieser Arbeit erarbeitete Konzept sieht zudem eine nahtlose Integration des Schedulingverfahrens in einen *State–of–the–Practice* Entwurfsfluss mit heutigen CASE Werkzeugen vor.

Der Aufbau dieser Arbeit orientiert sich an dieser Zielsetzung und ist wie folgt strukturiert.

In Kapitel 2 werden zunächst relevante Projekte aus dem Stand der Forschung aus den Bereichen Scheduling, hybride Systeme, Überlastbehandlung, Entwurfsfluss und hardwaregestützte Laufzeitsysteme diskutiert. Zu den einzelnen Bereichen werden jeweils die in dieser Arbeit motivierten Ansätze beschrieben. Die Basis bildet die Motivation für das *Transaktionsbasierte EDF Scheduling*. Anschließend wird auf das Robuste TEDF Scheduling eingegangen, was eine Erweiterung des Schedulings darstellt, um hybride (harte, *Quality of Service* und weiche) Transaktionen auf einem Laufzeitsystem zu kombinieren. Es wird die Überlastüberwachung und –behandlung beschrieben, ebenso wie die Motivation zur Auslagerung dieser Funktionalität auf einen externen *loosely coupled* Coprozessor.

In Kapitel 3 wird die formale Beschreibung für das *Transaktionsbasierte Earliest Deadline First (TEDF)* Schedulingverfahren gegeben. Der zugehörige Realzeitnachweis sowie die Berücksichtigung von Blockierungszeiten durch Protokolle zur Vermeidung von Prioritätsinversion wird beschrieben.

Die in dieser Arbeit entwickelte Überlasterkennung und –behandlung „*Robustes Transaktionsbasiertes EDF (RTEDF)*“ wird in Kapitel 4 dargestellt. Dazu wird der Algorithmus zur Erkennung sowie die Strategie zur Behandlung der Überlast zur Laufzeit beschrieben. Das entsprechende offline Nachweisverfahren sowie abschließende Bewertungen durch Simulationen werden in Kapitel 5 präsentiert.

In Kapitel 6 wird die prototypische Implementierung der Arbeit beschrieben. Dabei wird sowohl auf die Implementierung des transaktionsbasierten Ansatzes im Realzeitbetriebssystem *eCos* als auch auf die Entwicklung und Implementierung des RTEDF Coprozessors auf eine SOPC (System–on–a–Programmable–Chip) Architektur eingegangen. Messungen der auf der Plattform laufenden Algorithmen, sowie Overheadbestimmungen des Laufzeitsytems werden dargestellt.

Kapitel 7 schließt mit einer Zusammenfassung und einem Ausblick auf künftige Entwicklungen.

2 Stand der Technik und Ansätze dieser Arbeit

Die Basis eines jeden Laufzeitsystems bildet die zur Ressourcenverteilung gewählte Schedulingstrategie. In der Realzeitliteratur finden sich dazu zahlreiche Schedulingverfahren und –erweiterungen, die in den letzten 30 Jahren entwickelt wurden. Während in den ersten Arbeiten der Fokus auf rein harten Realzeitsystemen lag, so hat sich der Forschungsschwerpunkt der letzten Jahren mehr und mehr auf sogenannte *hybride*[1)] Realzeitsysteme verlagert. Diese Entwicklung zeigt die sich ändernden Anforderungen an Realzeitsysteme, die mittlerweile nicht nur für harte Realzeitaufgaben sondern auch für die Bewältigung von *Quality of Service (QoS)* Aufgaben eingesetzt werden.

Hybride Realzeitsysteme unterscheiden sich vor allem in zwei Punkten von den klassischen harten Realzeitsystemen: *Ausführungszeit* und *Ereignisabstand*.

Die Aktivierungssequenzen sind nicht immer als rein periodisch zu betrachten, sondern häufig auch aperiodischer oder sogar sporadischer Natur. Des Weiteren basieren Softwaremodelle vielfach sowohl auf reaktiven Modellen für ereignisgesteuerte Anteile als auch auf transformierenden Modellen für daten– und signalverarbeitende Anteile. Dies hat zur Folge, dass sich in *QoS* Software Modellen entsprechend der eingesetzten Algorithmen größere Variationen der Ausführungszeiten zur Laufzeit ergeben.

2.1 Schedulingverfahren

Die Klassifikation für Schedulingverfahren erfolgt aufgrund der Art und Weise der Prozessorzuteilungsstrategie. In Abbildung 2.1 ist eine darauf basierende Taxonomie dargestellt.

2.1.1 Statische Verfahren

Eine statische Prozessorzuteilungsstrategie bedeutet, dass zur Analysezeit bereits die Zeitpunkte festgelegt werden, zu denen eine Task des Systems Rechenzeit zur Verfügung gestellt wird. Systeme, die eine statische Prozessorzuteilungsstrategie besitzen, werden unter dem Begriff der *zeitgesteuerten* Systeme zusammengefasst. Dazu gehört das in der Automobilindustrie eingesetzte Laufzeitsystem OSEKtime® [27][2)]. Zur Analysezeit wird bereits eine Tabelle mit der sequentiellen Reihenfolge der zu bearbeitenden Tasks festgelegt. Ein Nachteil dieses Verfahrens

[1)] siehe Definition in Abschnitt 2.2

[2)] OSEKtime® bezeichnet ein Konzept und dazugehöriges API eines zeitgesteuerten Laufzeitsystems. In Anlehnung an den Industriestandard OSEK/VDX® kann OSEKtime® somit als Spezifikation eines zeitgesteuerten Laufzeitsystems betrachtet werden.

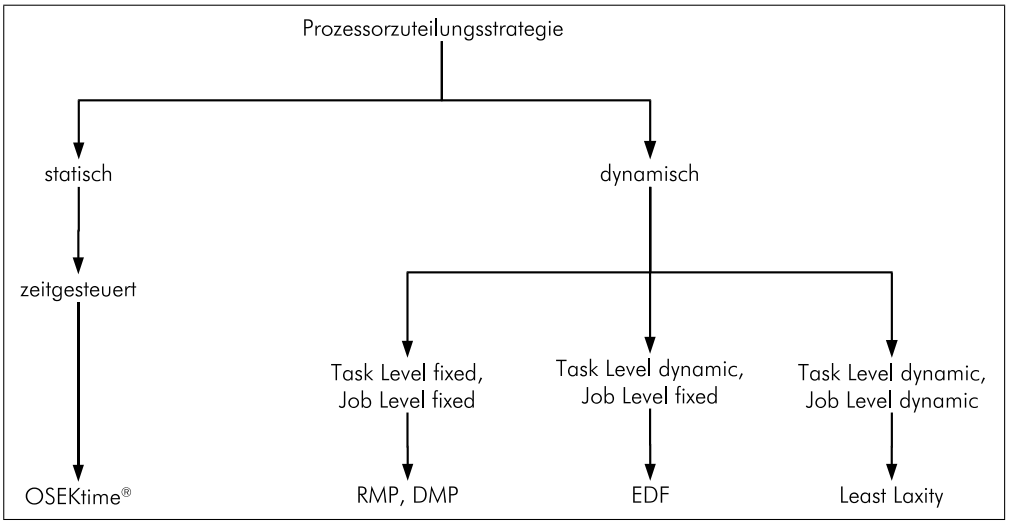


Abbildung 2.1: Prozessorzuteilungsstrategie

ist die aufwendige offline Analyse des Laufzeitsystems: Zugriffe auf Peripheriekomponenten und die Kommunikation zwischen den einzelnen Tasks müssen bereits bei der Erstellung der Ablaufplanung bekannt sein, damit alle Anforderungen erfüllt werden können. Eine solche Ablaufvorschrift ist bei jeder Änderung des Tasksets neu zu erstellen. Das optimierte Mapping von Tasks zu CPU Zeit stellt gerade bei komplexen Softwarearchitekturen und einer großen Anzahl an voneinander abhängigen Tasks hohe Anforderungen an die Optimierungsalgorithmen.

Der Vorteil des statischen Mappings ist die zur Laufzeit vorhersagbare Reihenfolge der Abarbeitung, was bei vielen echtzeitkritischen Systemen zu einer einfachen Wartbarkeit und Analysierbarkeit führt.

2.1.2 Dynamische Verfahren

Dynamische Verfahren vergeben die Prozessorleistung zur Laufzeit. Dabei lassen sich dynamische Systeme in drei unterschiedliche Klassen[3)] einteilen. Abbildung 2.2 zeigt die Task– und Job–Level Prioritätszuordnung dieser drei Klassen zur Spezifikationszeit und zur Laufzeit. Als Job wird in diesem Zusammenhang die einmalige Aktivierung einer Taskinstanz verstanden.

Task–Level fixed, Job–Level fixed Zu dieser Klasse gehören Schedulingverfahren wie *Rate Monotonic Priorities (RMP)* und *Deadline Monotonic Priorities (DMP)*. Eine zur Spezifikationsszeit bestimmte Priorität einer Task führt zur Laufzeit immer zu einer identischen Prioritätsverteilung der Taskinstanzen (Jobs).

Task–Level dynamic, Job–Level fixed Zu dieser Klasse gehört das Earliest Deadline First Scheduling. Tasks werden durch eine einzuhaltende Deadline charakterisiert. Die Priorität, die die einzelnen Jobs zur Laufzeit zueinander haben, kann nicht a priori angegeben werden. Die Priorität entspricht der zur Laufzeit ermittelten absoluten Deadline d_i. Die

[3)] Die Klassen entsprechen der Klassifizierung in [59].

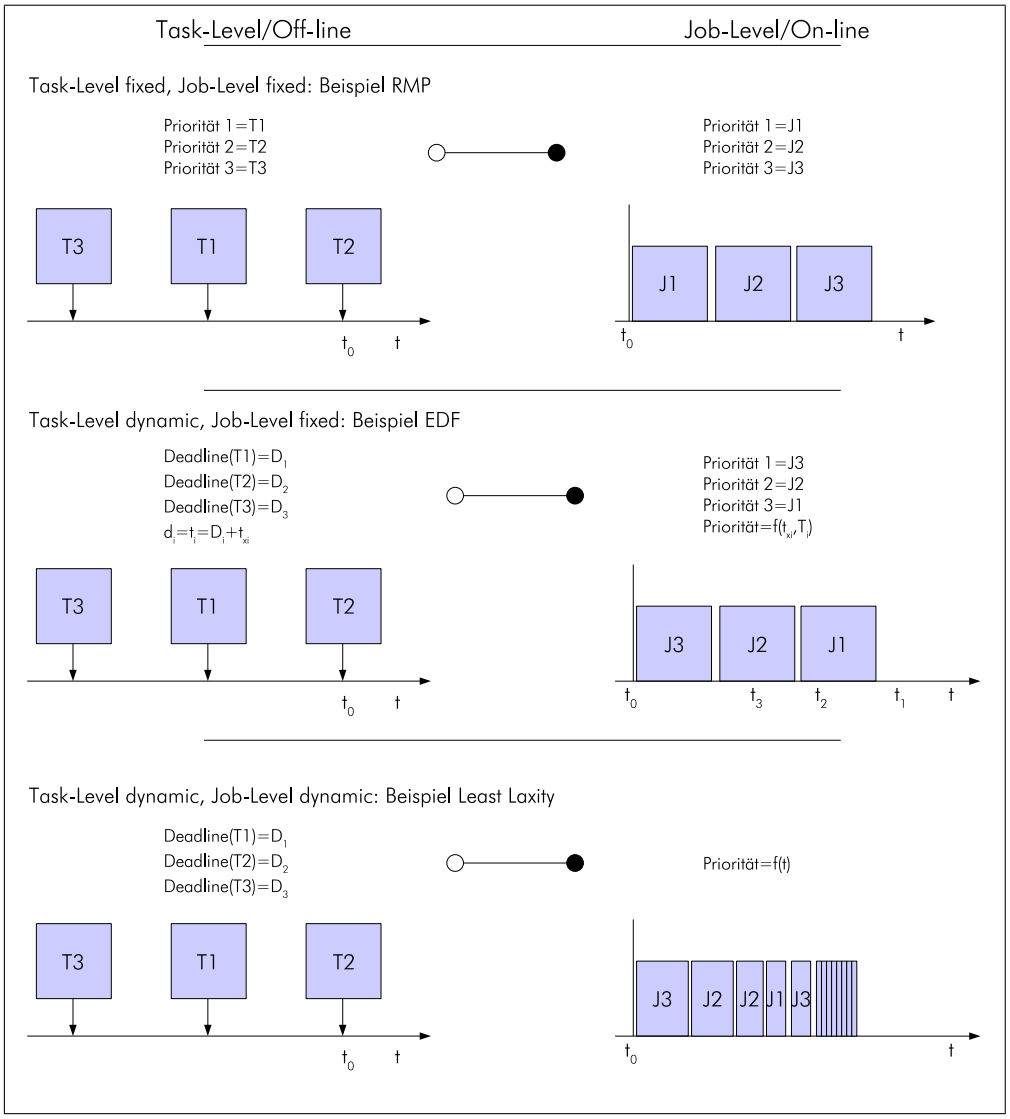


Abbildung 2.2: Klassen dynamischer Schedulingverfahren

absolute Deadline d_i hängt von dem Ereigniszeitpunkt t_{xi} ab, zu dem der Job einer Taskinstanz aktiviert wurde. Das Verhältnis dieser dynamisch ermittelten Priorität zu den anderen Prioritäten der bereits im System befindlichen rechenbereiten Jobs bleibt während der gesamten Rechenzeit dieses Jobs konstant.

Task–Level dynamic, Job–Level dynamic Zu dieser Klasse gehört das Least Laxity Scheduling. Die Laxity bestimmt sich aus dem Zeitintervall vom aktuellen Zeitpunkt bis zur absoluten Deadline einer Task, abzüglich der noch zu leistenden Rechenzeit. Dieses Maß ist abhängig von der aktuellen Systemzeit, da durch eine Abarbeitung eines Jobs auch dessen Laxity verändert wird und es damit zu einer Veränderung der Reihenfolge der Taskinstanzen kommt.

Mit ein Grund dafür, dass zu den Verfahren *Rate Monotonic* und *Deadline Monotonic Priorities* zahlreiche Arbeiten existieren, liegt darin begründet, dass die offline festgelegten Prioritäten zur Laufzeit ebenfalls Bestand haben. Dieser Zusammenhang vereinfacht die Analyse, da in den mathematischen Nachweisen immer von einer prioritätsorientierten Reihenfolge der Tasks zur Laufzeit ausgegangen werden kann.

2.1.3 Ansatz dieser Arbeit – Transaktionsbasiertes EDF

Die Motivation EDF als Basisscheduling in dieser Arbeit zu verwenden, leitet sich von der zunehmenden Dynamik der Softwaresysteme ab. EDF kann als optimales *Task–Level fixed* und *Job–Level fixed* Schedulingverfahren angesehen werden. Daher ist das *Earliest Deadline First Scheduling* als Schedulingverfahren für ereignisgesteuerte Systeme zur Bewältigung der unterschiedlichen Zeitanforderungen prädestiniert [37]. Wenn überhaupt ein gültiger Schedule existiert, so kann dieser stets mit dem EDF Scheduling erzeugt werden.

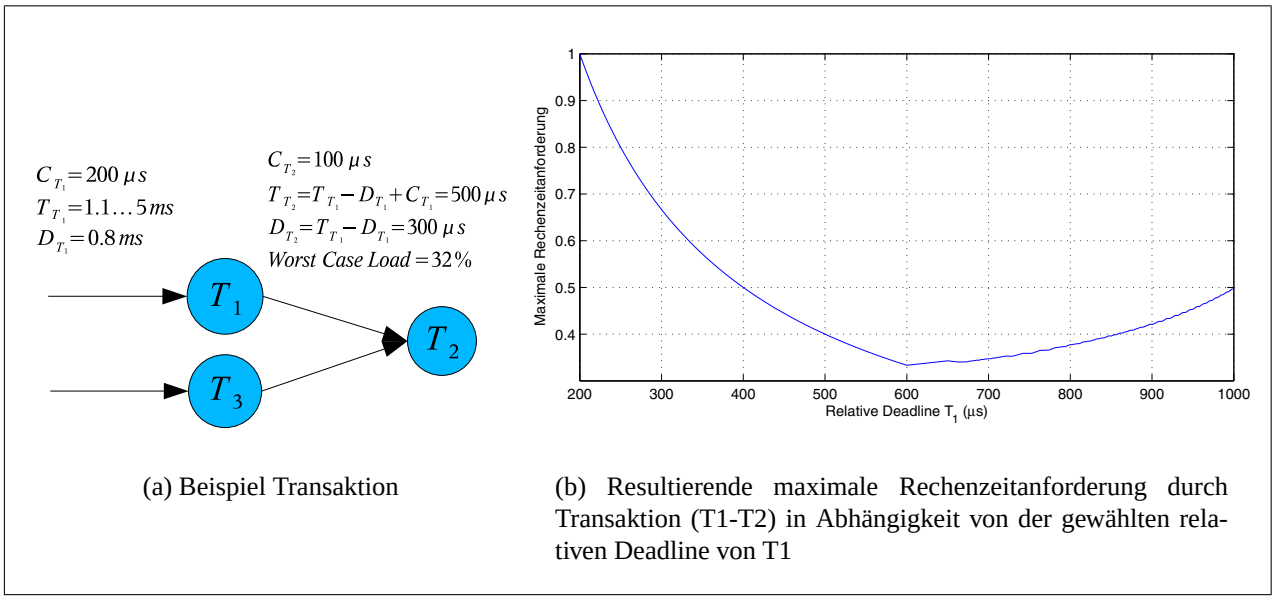
Das EDF Scheduling ist robust gegenüber Änderungen der Systemaktivierung und der Rechenzeiten von Tasks, solange der maximale Rechenzeitbedarf von 100% nicht überschritten wird. EDF Scheduling wurde erstmals von Liu und Layland 1973 in [58] vorgestellt.

Obwohl die Eigenschaft, Deadlines für Aufgaben festzulegen, Analogien zur Spezifikation von eingebetteten Realzeitsystemen nahelegt, ist eine Anwendung von EDF in Realzeitbetriebssystemen eher im akademischen Umfeld zu finden. Das hat unter anderem auch mit dem hohen Verifikationsaufwand zu tun, da eine Reihenfolge der Jobs zur Laufzeit nicht vorhergesagt werden kann und auch keine kommerziellen Werkzeuge bereitstehen.

Ein weit größerer Nachteil zeigt sich in heutigen Laufzeitsystemen bei der effizienten Realisierung von Präzedenzrelationen. Auf der einen Seite bieten Präzedenzrelationen Potential für einen realistischeren Realzeitnachweis. Auf der anderen Seite führen sie jedoch zu einer komplexen Abbildung der Spezifikationsdeadlines auf Taskdeadlines. In Beispiel 2.1 ist ein solcher Fall dargestellt.

Beispiel 2.1
Abbildung 2.3(a) zeigt ein Beispiel zu einer Präzedenzrelation mit drei Tasks. In der Spezifikation ist für die Bearbeitung der Transaktion bestehend aus Task 1 und Task 2 eine relative Deadline entsprechend der kürzesten möglichen Periode von $D = 1.1$ ms vorgeschrieben. Die Rechenzeiten der beiden Tasks betragen $C_1 = 200\ \mu s$ und $C_2 = 100\ \mu s$. Die Bestimmung der relativen Deadlines ist nicht eindeutig, da beide Deadlines in bestimmten Bereichen frei wählbar sind. In Abbildung 2.3(b) ist die Abhängigkeit der Rechenzeitanforderung der Transaktion in Abhängigkeit von der gewählten relativen Deadline von T1 gezeigt. Das Minimum liegt bei 32% des maximalen Rechenzeitbedarfes und $D_{T1} = 600\ \mu s$[4]. Das Beispiel zeigt, dass je nach Wahl der Deadlines zur Einhaltung der Taskpräzedenzen ein deutlicher Einfluss auf die Realzeitanalyse zu erwarten ist. Die Deadlinebestimmung wird zusätzlich erschwert, wenn weitere abhängige Transaktionen zu berücksichtigen sind ($T_3 \succ T_2$).

[4] Die Berechnung der Prozessorauslastung erfolgte mit Gleichung $h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) \cdot C_i$ aus [79].



(a) Beispiel Transaktion

(b) Resultierende maximale Rechenzeitanforderung durch Transaktion (T1-T2) in Abhängigkeit von der gewählten relativen Deadline von T1

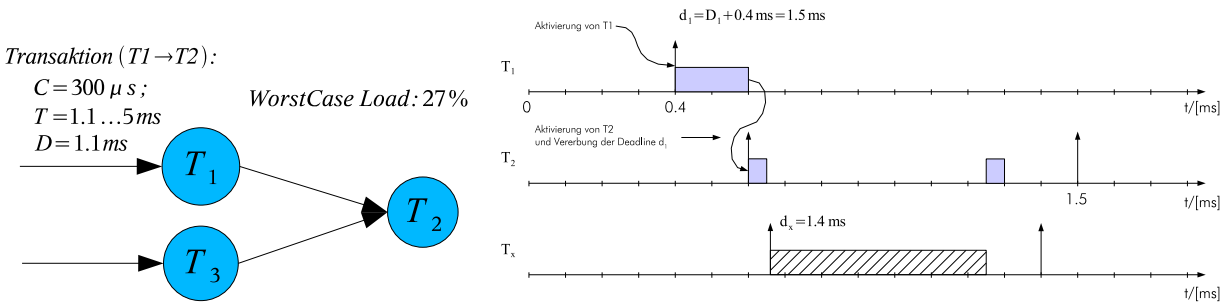
Abbildung 2.3: Beispieltransaktion und Prozessorauslastung in Abhängigkeit von D_{T_1}

Der Ansatz dieser Arbeit zur effizienten Berücksichtigung von semantischen Präzedenzrelationen im Laufzeitsystem besteht in der dynamischen Weiterleitung der absoluten Deadline während der Abarbeitung einer Transaktion. Dieses Verfahren sorgt dafür, dass abhängige Tasks in einer Transaktion immer die optimale und faire Priorität (Deadline) erhalten und wird als *Transaktionsbasiertes EDF Scheduling* definiert.

Beispiel 2.2
In Abbildung 2.4(a) ist das Tasksystem aus Beispiel 2.1 mit TEDF Laufzeitsemantik dargestellt. Bei einer Anwendung der TEDF Laufzeitsemantik wird die zum Ereigniszeitpunkt ermittelte absolute Deadline beim Übergang zur Task 2 weitervererbt, so dass Task 2 jederzeit die zur Abarbeitung optimale Deadline besitzt. Das bedeutet insbesondere auch, dass zur Designzeit nur die End–to–End Deadline initial vergeben werden muss, während Tasks, die ausschließlich von anderen Tasks angestoßen werden, bei jeder Aktivierung eine absolute Deadline erben. Eine Unterbrechung der Task 2 durch eine zu einer anderen Transaktion gehörenden Taskaktivierung ist genau dann möglich, wenn diese Transaktion eine kürzere absolute Deadline besitzt. Somit ergibt sich in diesem Fall eine Prozessorauslastung durch den Realzeitnachweis von 27 %.

Der Begriff der Transaktion wurde erstmals von Kopetz [42] bei der Definition von zeitgesteuerten Aktivitäten bei verteilten Systemen verwendet.

Als Transaktion wird eine semantische Abhängigkeit der zeitlichen Ausführung von Aktionen verstanden. So könnte beispielsweise das Empfangen einer CAN Nachricht mit einem Messwert durch einen Thread *GET_DATA*, die Eintragung dieses Wertes in eine zentrale Datenstruktur durch einen weiteren Thread *CENTRAL_DATA* und ein auf diesem Wert arbeitender Algorithmus in einem dritten Thread *FILTER_DATA*, als eine Transaktion angesehen werden. Alle Aktionen in diesem Beispiel stehen in einer zeitlichen und semantischen Abhängigkeit zueinander und setzen sich aus mehreren Threadaktivierungen zusammen. Diese Art der semantischen



(a) Beispiel Transaktion (b) Laufzeitverhalten für Transaktionsbasiertes EDF

Abbildung 2.4: Beispieltransaktion und Laufzeitverhalten bei Transaktionsbasiertem EDF

Taskabhängigkeiten ist vor allem bei modellbasierten und objektorientierten Entwurfsmethoden zu finden, die asynchron kommunizierende Prozesse verwenden. Funktionalität wird in Form von einzelnen Algorithmen entwickelt und anschließend zu einem funktionierenden Systemdesign miteinander verbunden. Vielfach sind einzelne Funktionsblöcke als Diensteerbringer für andere Funktionsblöcke zuständig.

Tindell[81][82][83] definiert den Begriff der Transaktion für die Spezifikation von Taskabhängigkeiten mit Offsets im Kontext prioritätsbasierten Schedulings. Das Transaktionsmodell dient hier zur weniger pessimistischen Berechnung von Antwortzeiten für harte prioritätsbasierte Realzeitsysteme. In [49] und [63] sind laufzeitoptimierte Versionen dieses Verfahrens mit dem Ziel beschrieben, es für eine online Analyse anwendbar zu machen. Das Wissen über semantische und zeitliche Abhängigkeiten im Modell ist von wesentlicher Bedeutung sowohl für die Laufzeiteffizienz des Schedulings als auch bei der Auslegung des Laufzeitsystems bei Realzeitsystemen.

Gresser hat in [48] Methoden präsentiert, wie eine Präzedenzrelation durch geschickte Wahl der Deadlines bei normalen Tasksystemen realisiert werden kann.

Weitere vergleichbare Ansätze existieren für das dynamische Scheduling von Nachrichten in Netzwerken für verteilte Systeme [62][43][44].

In [53] wurde von Kolloch ein Schedulingverfahren mit *Message Deadlines* entwickelt, dass eine vorhersagbare und analysierbare Abbildung von SDL Spezifikationen für harte Realzeitsysteme realisiert. Der Fokus liegt dabei in der Bereitstellung eines Methodik– Frameworks, das die nahtlose Integration der Realzeitnachweisverfahren in einen automatisierten Entwurfsprozess für harte Realzeitsysteme implementiert. Die resultierende Laufzeitsemantik ist dem TEDF Verfahren vergleichbar.

Ansätze zur Realisierung von Taskpräzedenzen wurden auch bei prioritätsbasierten Systemen in [46] und [35] vorgestellt.

2.2 Hybride Realzeitsysteme

Zukünftige Laufzeitsysteme müssen die Koexistenz von Softwaremodellen mit unterschiedlichsten Zeit- und Rechenanforderungen auf einem Kern garantieren und effizient realisieren können. Realzeitsysteme, die in der Lage sind sowohl Softwaremodelle mit unterschiedlichen zeitlich varrierenden Zeitanforderungen als auch Überlastszenarien robust und definiert zu handhaben, werden im Folgenden als *Hybride Realzeitsysteme* bezeichnet.

Softwaremodelle für hybride Realzeitsysteme können auf der obersten Ebene in harte und *nicht*–harte Realzeitsoftwaremodelle unterschieden werden. Während für den Begriff der harten Realzeitsysteme in der Fachliteratur eine eindeutige Definition existiert, gibt es zu der Klasse von *nicht*–harten Realzeitsystemen unterschiedlichste Definitionen. Für die Einordnung der folgenden Verfahren wird eine Einteilung der *nicht*–harten Realzeitsysteme in *nicht*–realzeit (NRT) und *Quality of Service* (*QoS*) durchgeführt. Letztere fasst sämtliche Taskspezifikationen zusammen, die ihre Deadline unter bestimmten Voraussetzungen überschreiten dürfen. Eine mögliche Klassifikation für hybride Realzeitsysteme kann mit den in Tabelle 2.1 aufgestellten Attributen erfolgen. Eine detaillierte Definition der *QoS*–Taskklassen wird für diese Arbeit in Abschnitt 2.2.3 gegeben.

Auch die in der der Fachliteratur bestehenden Definitionen zu den Systemaktivierungsklassen sind teilweise nicht eindeutig. Daher wurde zur Einordnung der im Folgenden beschriebenen Verfahren eine Definition der Systemaktivierungsklassen aufgestellt (siehe Tab. 2.2).

Es existieren zahlreiche wissenschaftliche Arbeiten zur Integration und Kombination von harten und *nicht*–harten Softwaremodellen. Dabei unterscheiden sich die Ansätze teilweise grundlegend voneinander im Hinblick auf die Voraussetzungen für die verwendeten Softwaremodelle. Im Folgenden wird nur auf solche Ansätze eingegangen, die für *Task–Level dynamic, Job–Level fixed* Schedulingklassen geeignet sind.

2.2.1 Kombination von HRT und NRT Taskklassen

2.2.1.1 Background Scheduling

Background Scheduling stellt die einfachste Art der Integration von unterschiedlichen Softwaremodellen dar. Während das Hauptscheduling für die harten Realzeittasks auf einer hohen Prioritätsebene angesiedelt ist, werden NRT Tasks auf einer niedrigeren Prioritätenebene abgearbeitet. Das bedeutet insbesondere, dass eine Abarbeitung der NRT Tasks nur dann durchgeführt wird, wenn in der höherprioren Ebene die *Idle* Task läuft. Die Implementierung dieses Verfahrens ist vergleichsweise einfach. Eine Unterstützung von *QoS*–Tasks ist nicht möglich.

2.2.1.2 Server–basierte Verfahren

Server–basierte Verfahren sind ein Spezialfall der hierarchischen Scheduler [71] [47] [72]. Hierarchische Scheduler besitzen einen globalen Scheduler, der ein Top–Level Taskset verwaltet.

Taskklassen	**HRT:**Harte Tasks haben definierte Deadlines. Alle Instanzen müssen zur Laufzeit ihre definierte Deadline einhalten. **NRT:***Nicht*–Realzeittasks haben keine Anforderungen an die Einhaltung von Deadlines. Sie werden abgearbeitet, wenn sie die übrigen Taskklassen nicht beeinflussen. **QoS:** *QoS*–Tasklassen stellen eine Komplementärmenge zu den beiden anderen Taskklassen dar. Existieren zeitlich variierende Anforderungen an die Einhaltung der Deadline so werden diese Eigenschaften als *QoS* Anforderungen definiert.
Systemaktivierung	Bei der Systemaktivierung wird in periodischen, aperiodischen und sporadischen Aktivierungen unterschieden. Eine Definition der drei Klassen ist in Tabelle 2.2 gegeben.
Optimierungsziel in Unterlast	Solange nicht mehr als 100% des Rechenzeitbedarfes für alle Intervalle angefordert wird, arbeitet der Scheduler im Unterlastmodus. Ein Optimierungsziel könnte eine möglichst kurze Antwortzeit für eine bestimmte Art von Tasks oder die Einhaltung der spezifizierten Deadlines aller Tasks sein.
Optimierungsziel in Überlast	In einer Überlastsituation können die Deadlines nicht mehr eingehalten werden. Es werden daher andere Optimierungsparameter verwendet, die es zu maximieren gilt.

Tabelle 2.1: Klassifikationsmerkmale für hybride Systeme

Dabei kann in mehreren Top–Level Tasks ein weiterer Scheduler integriert werden, der wiederum einen untergeordneten Satz an Tasks verwaltet. Der Verschachtelungstiefe sind keine Grenzen gesetzt. Allerdings nimmt die Analysierbarkeit mit zunehmender Verschachtelungstiefe ab.

Bei server–basierten Verfahren hingegen existiert eine einzige Verschachtelungsstufe. Top–Level Tasks implementieren einen weiteren Scheduler, der immer dann seine Tasks abarbeitet, wenn er vom globalen Scheduling Rechenzeit erhält.

Server–basierte Laufzeitsysteme sind für die Realisierung verschiedener Strategien entwickelt worden.

Ein Optimierungsziel der server–basierten Verfahren ist die Integration von aperiodischen Nicht–Echtzeittasks (NRT) in ein Laufzeitsystem mit periodischen harten Realzeittasks (HRT). Ziel dieser Verfahren ist es, in einem periodischen System für harte Realzeitaufgaben auch Nicht–Realzeitaufgaben mit aperiodischem Verhalten zu integrieren. Der Fokus liegt auf folgenden zwei Aspekten:

periodisch	Eine periodische Systemaktivierung bedeutet, dass entweder über interne oder externe Timer eine stetig periodische Aktivierung des Systems stattfindet. Das Intervall der Aktivierungen kann während der gesamten Laufzeit als konstant angesehen werden und wird mit dem Parameter der Periode t beschrieben.
aperiodisch	Aperiodische Aktivierungen sind durch den minimalen Ereignisabstand I_{min} definiert. Jitterbehaftete periodische Aktivierungen lassen sich als aperiodische Systemaktivierungen modellieren.
sporadisch	Sporadische Aktivierungen sind ein Sonderfall der aperiodischen Aktivierung, mit der Eigenschaft $I_{min} \rightarrow 0$. Der minimale Ereignisabstand hängt von der Systemarchitektur und der Laufzeit der Interrupt Service Routinen ab.

Tabelle 2.2: Klassifikation der Systemaktivierungen

1. Die harten Realzeittasks sind vor einer Deadlineverletzung zu schützen und
2. für die aperiodischen NRT Tasks ist ein möglichst optimales Antwortzeitverhalten zu erzielen.

Ansätze dieser Kategorie sind von Spuri und Buttazzo veröffentlicht worden. Dazu gehören der *Dynamic Priority Exchange Server (DPE)*, der *Dynamic Sporadic Server (DSS)*, der *Total Bandwidth Server (TBS)* sowie der *Constant Bandwidth Server (CBS)* [28] [78] [56] [79]. Als Basisscheduling wird EDF verwendet. Eine periodische EDF Task hat dabei die Rolle des Servers. Dieser Task wird eine Prozessorzuteilung, auch Budget genannt, garantiert. Wird eine aperiodische Task rechenbereit und hat die Servertask noch genügend Rechenzeit zur Verfügung, so kann diese NRT Task abgearbeitet werden. Dabei wird der NRT Task eine Deadline entsprechend des dem Servertask zustehenden Budgets zugeteilt. Damit wird sichergestellt, dass eine Beeinflussung der HRT Tasks nicht auftritt. Unterschiede zwischen den einzelnen Servern bestehen in der Art und Weise wie das Budget wieder aufgefüllt wird. Neuere Ansätze sind in der Lage die Budgets zwischen unterschiedlichen CB Servern dynamisch zur Laufzeit zu verteilen [39]. Da die Servertask wie eine normale periodische EDF Task betrachtet werden kann, ist ein a priori Realzeitnachweis des Gesamtsystems möglich.

Ein optimales server–basiertes Verfahren in Bezug auf Antwortzeiten ist der *Earliest Deadline as Late as Possible Server (EDL)*, der zuerst von Chetto [40] und Silly [75] [76] beschrieben wurde. Die periodischen HRT Tasks werden nach EDF verwaltet. Trifft eine NRT Task ein, so wird zunächst der vorhandene Slack berechnet, der zu diesem Zeitpunkt zur Verfügung steht, wenn man zu diesem Zeitpunkt für die periodischen HRT Tasks auf die Strategie *Earliest Deadline as Late as Possible (EDL)* umschalten würde. Durch EDL wird dem System die maximal mögliche Zeit pro Intervall entzogen und den NRT Tasks zur Verfügung gestellt. Der EDL Server ist ein optimaler Server zur Minimierung der Antwortzeiten für die aperiodischen NRT Tasks [77][79].

Alle Verfahren zeigen ein besseres Antwortzeitenverhalten für sporadische und aperiodische NRT Tasks als das *Background Scheduling*.

Eine Unterstützung von *QoS*–Taskklassen kann mit diesen Verfahren nicht realisiert werden.

2.2.2 Überlasterkennung und –behandlung

Bei den bisherigen Betrachtungen wurde davon ausgegangen, dass weder die HRT noch die NRT Tasks ihre vorgesehenen Rechenzeitanforderungen überschreiten. Gerade für Software aus dem nicht–HRT Bereich sind jedoch größere Unterschiede zwischen Best Case und Worst Case Execution Time zu beobachten. Wird ein System nicht mit Berücksichtigung der möglichen auftretenden Worst Case Ausführungszeit ausgelegt, so kann zur Laufzeit im ungünstigsten Fall eine Überlast durch *Overruns*[5)] auftreten. Als Überlast versteht man einen Systemzustand, bei dem selbst mit einem optimalen Schedulingverfahren kein gültiger Schedule gefunden werden kann.

In der Literatur sind zahlreiche Strategien zur Erkennung und Behandlung dieser Überlastsituationen dokumentiert. Die Unterschiede zwischen den einzelnen Verfahren liegen im Optimierungsziel beim Auftreten einer Überlast.

Während im Normalbetrieb die Deadline als Schedulingrichtgröße dient, so muss im Überlastfall eine andere Richtgröße verwendet werden. Einige Verfahren verwenden für die Richtgröße im Überlastfall eine Wertefunktion (*Time Utility Funktion*), die angibt, welchen Wert diese Berechnung in Abhängigkeit von der Fertigstellung nach der Deadline für die Gesamtfunktionalität noch besitzt. Für diese *TUFs* gibt es unterschiedliche zeitliche Verläufe (Abb. 2.5).

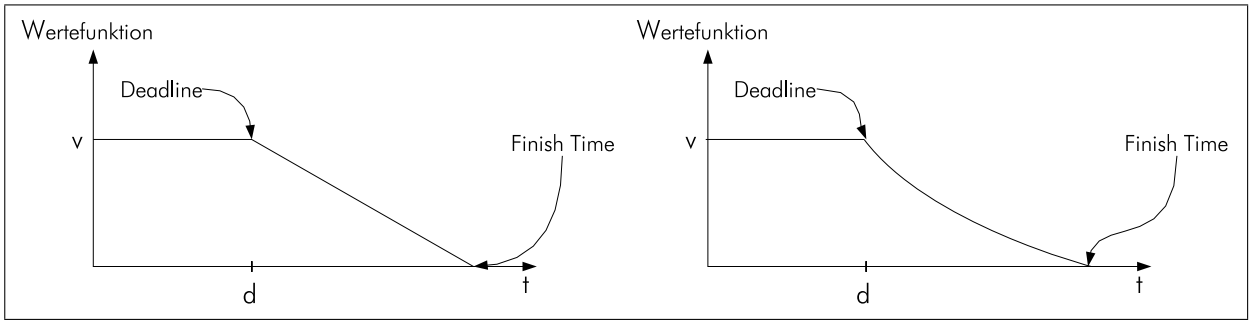


Abbildung 2.5: Time Utility Funktionen

Dabei werden die Algorithmen mit einem optimalen, d. h. vorausschauenden Scheduler (Clairvoyant Scheduler) verglichen. Der optimale, vorausschauende Scheduler hat das Wissen über zukünftige Auftrittszeitpunkte aller Tasks und die exakten Ausführungszeiten.

In [34] wurde nachgewiesen, dass kein online Algorithmus im Überlastfall einen besseren Optimalitätsfaktor als 0.25 besitzt. Der Optimalitätsfaktor bezeichnet das Verhältnis der erzielbaren Funktionswerte vom realen Algorithmus zum vorausschauenden Algorithmus. Der Funktionswert ergibt sich aus dem der Task zugeordneten Wertefunktion (Abb. 2.5).

[5)] Als *Overrun* wird im Folgenden die Überschreitung der für die Analyse verwendeten maximalen Ausführungszeit definiert.

Im Folgenden werden die existierenden Verfahren anhand von Optimierungkriterien klassifiziert.

2.2.2.1 Optimierung der Antwortzeiten

Server–basierte Verfahren, wie der *Constant Bandwidth Server (CBS)*, sind in der Lage Überlastsituationen zu behandeln, da sie einen festen Teil der Prozessorbandbreite für harte Realzeittasks reservieren. Durch dieses Budget sind sie in der Lage, Overruns zu kontrollieren [38][45][61]. Das Optimierungsziel ist die Minimierung der Antwortzeiten für die aperiodischen NRT Tasks. Dabei wird vorausgesetzt, dass ein Teil des Tasksets aus periodischen HRT Tasks besteht, während die überlastverursachenden Tasks entweder aperiodische oder periodische NRT Tasks sind.

2.2.2.2 Effektive Prozessorauslastung (EPA)

Dynamische Scheduling Algorithmen wie EDF erreichen im Überlastfall eine EPA von 25% [34]. Die EPA ist definiert als der Anteil der betrachteten Prozessorzeit, die für Tasks aufgewendet wurde, die ihre Deadline einhalten konnten. Verfahren, welche die effektive Prozessorauslastung im Überlastfall optimieren, gehen von weichen Realzeittasks aus, die nur dann einen positiven Qualitätsbeitrag leisten, wenn sie innerhalb ihrer Deadline abgearbeitet wurden.

Baruah und Haritsa konnten diese Grenze in ihrem Scheduling Algorithmus *ROBUST (Resistance to Overload by Using Slack Time)* durch die Annahme eines Slack Faktors[6)] von 2 für alle Tasks auf 50 % erhöhen [33][32][31]. Mit EPA als Optimierungsziel in einer Überlastphase kann jedoch keine Aussage getroffen werden, welche Taskaktivierung eine Deadlineverletzung haben wird und um welchen Betrag die Deadline überschritten werden wird.

2.2.2.3 Skipping

Ein weiterer Ansatz der in [55] von Koren und Shasha untersucht wurde, ist das *Skippen* (Überspringen) von Taskaktivierungen. Dabei wird nicht eine Task, deren Deadline nicht eingehalten werden kann, verworfen, sondern es werden Tasks, die ihre Deadline zur Laufzeit nicht eingehalten haben, als *Übersprungen* betrachtet. Mit einem Skip Parameter wird der Mindestabstand zwischen zwei übersprungenen Tasks spezifiziert.

2.2.2.4 (m,k) Firm Taskmodell

Eine durch ein (m, k) Tupel definierte periodische Task genügt genau dann der Spezifikation, wenn bei k konsekutiven Taskaktivierungen m Taskaktivierungen innerhalb der Deadline abgearbeitet werden. Der Ansatz von Ramanathan, der in [69] und [70] vorgestellt wird, ist auf

[6)] Der Slack Faktor bezeichnet das Verhältnis von relativer Deadline zu Ausführungszeit.

prioritätsbasiertes Scheduling beschränkt. Zu dem Schedulingverfahren existiert ein Realzeitnachweis, mit dem a priori die Einhaltung der (m, k) *Firm* Zeitanforderungen analysiert werden kann.

Ansätze, um das Konzept auf dynamische Schedulingverfahren zu erweitern, werden in [70] und in [84] vorgestellt. Das in [70] definierte *Distance Based Protocol* (DBP) wird in [67] als Erweiterung (*Matrix Distance Based Protocol* (MDBP)) für EDF basierte Laufzeitsysteme vorgestellt. Dabei definiert sich die Priorität einer Task dynamisch aus der Differenz $k - m$. Diese stellt das Verhältnis zwischen der Anzahl der überschrittenen Deadlines bei k Aktivierungen dar.

Das *Dynamic Window–Constraint Scheduling (DWCS)*[84] nutzt eine ähnliche Strategie zur Laufzeit, um die *m* Taskaktivierungen mit eingehaltener Deadline innerhalb eines Zeitfensters zu realisieren.

2.2.2.5 Best–Effort Algorithmen

Es gibt Anwendungen, bei denen das Ausmaß der Qualitätsreduktion von der Länge der Deadlineüberschreitung abhängt. Die Spezifikation des Qualitätswertes erfolgt dabei mit Hilfe von *Time Utility* Funktionen, die unterschiedliche Verläufe haben können. Verfahren, die in Überlastsituationen online versuchen einen Schedule zu finden, der im Sinne der *Time Utility* Funktionen zu einem Maximum des Qualitätswertes führt, werden *Best–Effort* Algorithmen genannt.

RED *Robust Earliest Deadline* steht für ein Verfahren, das Tasks beim Eintreffen in das System aufnimmt, wenn ihre zeitlichen Anforderungen garantiert werden können. Die Form des *Acceptance Testing* wird mit jeder neu hinzukommenden Taskaktivierung durchgeführt. Dabei besitzt jede Task einen weiteren Parameter (Tardiness), der angibt, ob eine verspätete Abarbeitung der Task zulässig ist. Kann die Task zum aktuellen Zeitpunkt nicht bearbeitet werden, so wird entweder die eintreffende Task oder eine bereits zugelassene Task aus der Ready Queue entfernt bzw. verzögert. Algorithmen, die zur Laufzeit entscheiden, welche Task entfernt und verzögert werden soll, werden in [79] nicht explizit genannt.

Dieses Verfahren garantiert die Einhaltung der Deadlines nur auf „per Job Basis“. Es existiert dazu kein Realzeitnachweis für ein Taskset mit *QoS*– oder weichen Tasks als Ganzes.

LBESA, DASA *Locke's Best Effort Scheduling Algorithm (LBESA)*[60] und das *Dependent Activity Scheduling Algorithm (DASA)*[41] von Clark gehören beide zur Kategorie der Best–Effort Algorithmen. Im Standardfall verhalten sie sich wie ein EDF Scheduler und sind daher optimal. Im Überlastfall werden Schedulingentscheidungen basierend auf den spezifizierten *Time Utility* Funktionen realisiert.

In neueren Arbeiten [57] werden laufzeitoptimierte Implementierungen (MDASA, MLBESA) beschrieben.

D^{over} In [54] wird ein optimales Best–Effort Scheduling D^{over} vorgestellt, das einen Optimalitätsfaktor von $(1 + \sqrt{k})^2$ besitzt. Der Faktor k beschreibt den Quotienten aus der maximalen und minimalen auftretenden Dichte der Wertefunktion für ein Taskset. Die Dichte ist der Quotient aus einem für die Task festgelegten Wert und seiner zu erwartenden Rechenzeitanforderung. Kann eine Task seine Deadline nicht einhalten, so werden auch sogenannte *Value Functions* für den Werteverlauf der Task in Abhängigkeit von der *Finish Time* angegeben.

2.2.2.6 Imprecise Computation Model

Das *Imprecise Computation Model (ICM)* setzt voraus, dass Algorithmen in verschiedenen Qualitätsstufen konzipiert werden können. Die erste Stufe ist für eine Basisfunktionalität des Gesamtsystems nötig und wird als *Mandatory Part* bezeichnet. Weitere Teile des Algorithmus führen bei Bearbeitung zu einer Verbesserung des algorithmischen Ergebnisses und sind daher als *Optional Part* definiert. Dieses stufenweise Softwaredesign kann in Überlastsituationen genutzt werden, um die Last durch Weglassen der optionalen Anteile zu verringern.

Pfefferl stellt in [66] geeignete Verfahren vor, die für Realzeitalgorithmen in der Mechatronik eingesetzt werden können.

Ein EDF–basiertes Realzeitbetriebssystem mit einer Unterstützung des *ICM* wird in [52] beschrieben.

Eine Implementierung dieser Modelle setzt ein angepasstes Software– und Algorithmendesign voraus und stellt somit einen wesentlichen Eingriff in die Entwurfsmethodik dar.

2.2.3 Ansatz dieser Arbeit – Robustes Transaktionsbasiertes EDF

Für den Ansatz dieser Arbeit wird die in Tabelle 2.1 aufgestellte Taxonomie näher präzisiert.

Klassifikation der Softwaremodelle

In dieser Arbeit wird zwischen harten (HRT), *Quality of Service* (*QoS*) und weichen (SRT) Softwaremodellen unterschieden.

In Abbildung 2.6 ist ein Softwareschichtenmodell dargestellt, dass eine grobe Einordnung der zeitlichen Anforderungen von Softwaremodellen erlaubt. Dabei ergeben sich für Softwaremodelle aus den oberen Ebenen größere Variationen der Rechen– und Ereigniszeiten, während hardwarenahe Software nur eine geringe Abweichung zur Laufzeit aufweist.

Die Klassifikation der Realzeitbereiche, die für diese Arbeit herangezogen werden, sind im Folgenden definiert.

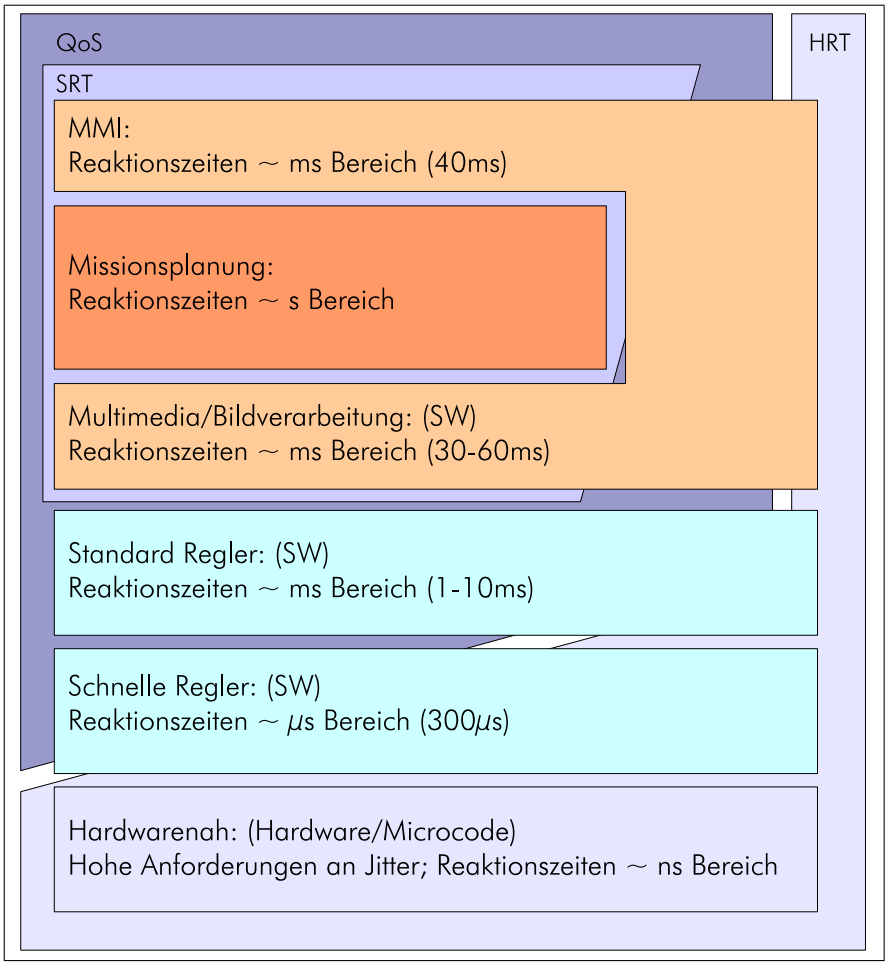


Abbildung 2.6: Softwareschichtenmodell und Einordnung HRT, *QoS* und SRT

Hart (HRT)

Harte Realzeittasks besitzen eine Deadline, die unter keinen Umständen überschritten werden darf. Nach einer Aktivierung einer harten Realzeittask muss deren Berechnung innerhalb der festgelegten Deadline abgeschlossen sein. Übertretungen dieser Deadline zur Laufzeit sind nicht erlaubt. Verfahren zur Auslegung von harten Realzeitsystemen sind in der Literatur dokumentiert. Ihre Anwendung ist jedoch auf harte Realzeitanwendungen beschränkt, da eine Anwendung auf Softwaremodelle mit variierenden Rechen– und Ereigniszeiten zu einer Überspezifikation der Systemarchitektur führen würde. Harte Anforderungen können in allen Softwareschichten auftreten.

Quality of Service (*QoS*)

Es existieren zahlreiche Anwendungsgebiete, die nur teilweise eine harte Abarbeitung der Softwaremodelle benötigen. Ein Teil dieser Applikationen verhält sich robust gegenüber partiellen[7)]

[7)] Partiell bedeutet hier, dass nach n Ausführungen mit einer verlängerten Deadline m Ausführungen ohne Deadlineverlängerung stattfinden müssen.

Deadlineverlängerungen sofern ein bestimmtes Maß nicht überschritten wird. Ein anderer Teil wiederum erfährt bei einer partiellen Deadlineverlängerung eine Degradierung der in der Spezifikation festgehaltenen Funktionalität, die jedoch nicht zu einer Beeinträchtigung der Gesamtfunktionalität führt. Inwiefern eine Degradierung tolerierbar ist, hängt von den individuellen Anforderungen ab. *Quality of Service (*QoS*)* Tasks haben Freiheitsgrade in den zeitlichen Anforderungen, die zur Laufzeit für eine Systementlastung bei Worst Case Szenarien eingesetzt werden können. Die Freiheitsgrade werden über drei Parameter hinreichend definiert:

1. Die Verlängerung einer Transaktionsdeadline zur Laufzeit ist möglich. Der Betrag, um den die Deadline verlängert werden kann, ist a priori definiert. Eine Transaktion, deren Deadline verlängert wurde, wird *Delta*–Transaktion genannt.
2. Die Anzahl der konsekutiv erlaubten *Delta*–Transaktionen ist begrenzt und ebenfalls a priori definiert.
3. Nach einer maximalen Anzahl von *Delta*–Transaktionen, wird die Ausführung mit der primären Deadline gefordert. Die Anzahl dieser *Firm*–Transaktionen ist ebenfalls a priori definiert.

Im Folgenden werden diese drei Attribute zu einem drei–tupel zusammengefasst und stellen ein *QoS*–Pattern dar, dass vom Laufzeitsystem zur Systementlastung in Worst Case Szenarien verwendet werden kann.

Soft (SRT)

Weiche Realzeittasks besitzen zeitliche Anforderungen, ähnlich der für *QoS*–Tasks. Es sind im Unterschied zu *QoS*–Transaktionen keine *Firm*–Transaktionen notwendig, so dass die Anzahl der *Delta*–Transaktionen beliebig ist.

Non–Real–Time (NRT)

Nicht–Realzeitsysteme stellen keine zeitlichen Anforderungen bezüglich der Abarbeitung und werden im Folgenden nicht weiter betrachtet.

Eine detailliertere Betrachtung der drei Bereiche erfolgt in Kapitel 4, Abschnitt 4.2.

Die Motivation dieser Arbeit besteht darin, die laufzeittechnische Kombination von Tasksystemen mit harten, *QoS* und weichen Zeitanforderungen gleichermaßen zu ermöglichen. Im Unterlastfall sollen die zur Laufzeit entstehenden Slack Zeiten (durch Variation von Ausführungszeiten, nicht entstehenden Blockierungszeiten und Variation von Ereignisabständen) zur Erzielung einer hohen *Qualität* von *QoS*–/*Soft*–Transaktionen genutzt werden. Im Überlastfall hingegen soll für diese Taskklassen eine vorab definierte Reduktion der Qualität mit *QoS*–Pattern angewendet werden, so dass ein robustes Verhalten gewährleistet ist. Dabei soll mit *QoS*–Pattern definiert werden, wie oft hintereinander eine Task mit einer verlängerten Deadline (*Delta*) bearbeitet werden darf, um anschließend wieder mit einer Mindestanzahl an Ausführungen mit ihrer primären Deadline (*Firm*) abgearbeitet zu werden.

Der RTEDF Ansatz kombiniert die online Analyse aus den *Best–Effort* Ansätzen mit einem

offline Nachweis für *QoS*–Pattern. Im Gegensatz zu *DBP* und *(m,k) Firm*, werden definierte Pattern spezifiziert, die mit einem Nachweisverfahren zur Laufzeit garantiert werden können.

Im Unterlastfall wird durch eine verzögerte Umschaltung in den Überlastmodus die Nutzung von Slackzeiten anderer Transaktionen realisiert. Das EDF Verhalten wird bis zur real auftetenden Überlast beibehalten und führt zu einem optimalen Scheduling in der Unterlastphase.

Ein weiteres Ziel ist es, den Rechenaufwand für die online Analyse soweit wie möglich durch eine Verlagerung in die offline Analyse zu minimieren.

Die Verwendung von definierten Pattern hat vor allem für den Entwurf von Softwaresystemen einen großen Vorteil. Systemtheoretische Modelle technischer Software gehen zunächst von einer idealisierten Realisierung der Algorithmen aus (Fließkomma Darstellung und kontinuierliche Zeitmodellierung). Ist der funktionale Nachweis mit dem idealen Modell erbracht, so erfolgt eine Diskretisierung sowohl der Datentypen als auch der Abtastzeitpunkte (Festkommadarstellung und zeitdiskrete Abbildung). Dieser Schritt im Entwurfsfluss wird gegen das ideale Modell in Bezug auf Robustheit und Korrektheit überprüft. Geht man nun bei Überlastsituationen davon aus, dass einzelne Berechnungsschritte verzögert werden, so ist es für eine Überprüfung der Robustheit und Korrektheit gegenüber dem idealen Modell von entscheidender Bedeutung, dass die Simulation mit fest definierten Pattern durchgeführt werden kann. Eine statistische Betrachtung über die Einhaltung von Pattern liefert für diesen Entwurfsfluss hingegen keine konkrete Aussage.

2.3 Integration in den Entwurfsprozess

Das „Engineering“ von eingebetteten Realzeitsystemen stellt besondere Anforderungen an alle Phasen der Softwareentwicklung. Dabei kann der Entwurfsfluss aus Sicht der Entwurfssemantik in eine *Operationale* und eine *Denotationale* Entwurfsebene abstrahiert werden.

Entwurfsebene

Die Entwurfsebene umfasst die Anforderungsanalyse, den Entwurf, die funktionale Verifikation, die Codegenerierung und Anbindung an Embedded Targets. Das Hauptaugenmerk liegt hier in der Programmierunterstützung des Designers für imperative Programmiersprachen wie C und C++. Die funktionale Verifikation der Softwaremodelle sowie die prototypische Implementierungen für *Hardware in the Loop* Tests sind Teil dieser Ebene.

Meta–Entwurfsebene

Die Meta–Entwurfsebene kann als eine parallel zum klassischen Entwurfsfluss verlaufende Ebene gesehen werden. Hier werden Metadaten, die entweder aus der Anforderungsanalyse oder erst beim Entwurf enstanden sind, aufgenommmen und analysiert. Die Einhaltung der Vorgaben wird im Endsystem garantiert. Unter Metadaten werden Daten verstanden, die Aussagen über das funktionale Verhalten machen, jedoch nicht durch das Softwaremodell selbst realisiert

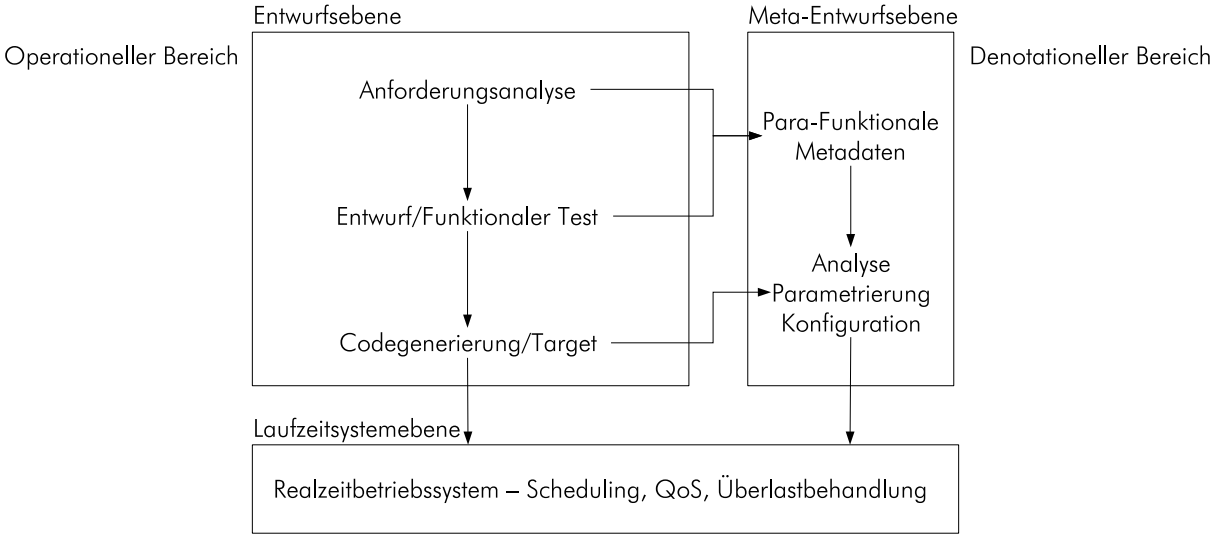


Abbildung 2.7: Abstraktion und Taxonomie des Entwurfsflusses

werden können oder sollen. Letzteres ergibt sich aus der Notwendigkeit, ein möglichst erweiterbares und wiederverwendbares Softwaredesign zu erstellen. Wird die Implementierung der Metadatenfunktionalität mit dem Softwaremodell verwoben, so können diese Ziele nicht erreicht werden. Die Spezifikation von Metadaten hat den Charakter von denotationellen Semantiken: Es wird etwas über die Funktionalität, die im Endsystem erbracht werden muss, ausgesagt, nicht jedoch wie diese zu realisieren ist.

Beispiel 2.3
Größen wie Programmspeicherbedarf, Stackgröße, End–to–End Deadline oder periodische Aktivierung mit einer Mindestperiodizität sind parafunktionale Parameter, die in vielen Softwareprojekten spezifiziert werden. Diese Größen können nicht durch funktionale Programmierung ausgedrückt werden, sind jedoch ebenfalls für eine korrekte Funktionalität des Systems notwendig.

2.3.1 Entwurfsebene

2.3.1.1 Werkzeuggestützter Entwurf eingebetteter Systeme

Entwurfswerkzeuge unterstützen den Entwickler beim Entwurf in nahezu allen Phasen des Entwurfes. Dabei unterscheiden sich die Ansätze in der dem Werkzeug zugrundeliegenden Modellierungssprache.

Während Programmiersprachen durch eindeutige Metamodelle in Form von EBNF Grammatiken und Standards definiert sind, ist dies bei einigen Modellierungssprachen nicht der Fall. Sie stellen in der Regel einen Aufsatz für imperative Programmiersprachen wie C und C++ dar, um durch grafische Notationen den Entwurf der Software zu vereinfachen. Dabei sind die meisten

Werkzeuge über die Jahre gewachsen und stellen mittlerweile Notationen für zustandsbasierte und strukturelle Beschreibungselemente zur Verfügung.

Eine wesentliche Eigenschaft dieser Werkzeuge besteht darin, Simulationsmöglichkeiten zu integrieren, um so eine funktionale Verifikation der Softwaremodelle in der gleichen Umgebung zu ermöglichen. Sowohl die Option zum Rapid–Prototyping auf schnellen Rechnern (xPC, dSpace) als auch die Codegenerierung für Embedded Targets mit Realzeitbetriebssystemen gehören zu den Stärken dieser Werkzeuge. Beispiele für diese Werkzeuge sind ASCET–SD [21] von ETAS und Matlab/Simulink [24] von The Mathworks.

2.3.1.2 Laufzeitsystem–Anbindung

Der direkte Weg zur Implementierung führt bei der Anwendung bestehender Werkzeuge zu einer speziellen Art der Abbildung des Softwaremodells auf das Laufzeitsystem. Dabei gibt es große Unterschiede in der Abbildungseffizienz.

Einige Werkzeuge nutzen spezifische Codegeneratoren für ein domänenspezifisches Targetsystem (z. B. OSEK®). Dieses ist durch die spezialisierte Abbildung des Softwaremodells auf das Target gekennzeichnet. Der Nachteil dieser Lösung ist die Fixierung des Softwaremodells auf das Werkzeug und die Anwendungsdomäne.

Die große Mehrheit der Hersteller verfolgt das Ziel, möglichst viele Zielsysteme mit den eigenen Werkzeugen unterstützen zu können. Dieses lässt dem Entwickler die Freiheit sowohl das Hardwaretarget als auch das Realzeitbetriebssystem frei zu wählen. Diese Freiheit wird durch eine generische Art der Abbildung des Softwaremodells auf das Target erkauft. Es werden nur einfache Betriebssystemprimitive zur Realisierung verwendet, die auf fast jedem Realzeitbetriebssystem zu finden sind. Der Nachteil dieser Lösung ist die für den Embedded Bereich nicht optimale Effizienz der Abbildung. Die Unterstützung zur Spezifikation von Laufzeiteigenschaften als *parafunktionale* Anforderungen ist in den meisten Werkzeugen nicht vorhanden.

In [65] wird eine Methodik vorgestellt, die eine effiziente Implementierung auf ein Embedded Target mit Matlab/Simulink erlaubt.

Grafische Modellierungssprachen, die ein formales Metamodell besitzen sind beispielsweise Rhapsody[23], ARTiSAN RealTime Studio [20], Telelogic Tau [25] und IBM Rational Rose RT (Technical Developer).

2.3.2 Meta–Entwurfsebene

In der Meta–Entwurfsebene werden Systemanforderungen auf denotationale Art spezifiziert. Eine erfolgreiche Einhaltung dieser Spezifikation muss durch eine nahtlose Integration der einzelnen Werkzeuge sichergestellt werden.

Neuere Entwicklungen setzen verstärkt auf eine entwurfsbegleitende Spezifikation von parafunktionalen Eigenschaften. Parafunktionale Eigenschaften sind Eigenschaften, die für eine

korrekte Implementierung des Systems erforderlich sind, jedoch *orthogonal* zum entwickelten Softwareentwurf gesehen werden.

Da es sich bei diesen Sprachmitteln um Spezifikationen handelt, die im Allgemeinen nicht rein in Implementierungscode umgesetzt werden können, ist eine Behandlung dieser *parafunktionalen* Sprachmittel nur durch einen parallel verlaufenden Meta–Entwurfsfluss darstellbar. Da sämtliche Erweiterungen und Spezifikationen erst kürzlich standardisiert worden sind, wird es einige Zeit dauern bis eine geeignete Werkzeugunterstützung verfügbar sein wird. Allerdings ist ein Entwicklungstrend hin zu denotationellen Spracherweiterungen wie SysML [19] oder UML Profiles[89][90] bereits erkennbar.

2.3.3 Ansatz dieser Arbeit – Meta–Entwurfsfluss

Der Ansatz, der in dieser Arbeit verfolgt wird, sieht die Extraktion der während der Modellierung und Spezifikation entstehenden Metadaten (Transaktionspfade, Deadlines, Ressourcennutzung) in geeigneter Form vor. Eine softwaretechnische Berücksichtigung unterschiedlicher Systemzustände, wie dies beim *Imprecise Computational Model* der Fall ist, ist somit nicht erforderlich. Die Beschreibung der Vorgehensweise ist in Kapitel 6, Abschnitt 6.1, beschrieben.

2.4 Hardwaregestützte Schedulingverfahren

Die Idee das Scheduling auf eine zusätzliche Hardwareeinheit auszulagern, ist nicht neu. Erste Ansätze sind in der Arbeit von Tempelmeier [80] zu finden, in der bereits die Auslagerung des gesamten Betriebssystems auf einen weiteren Hardwarekern behandelt wird.

Im Spring Kernel Projekt [64][36] wurde das Scheduling für eine Multiprozessorarchitektur auf einen dedizierten Schedulingprozessor ausgelagert. Dieser berücksichtigt die Abhängigkeiten zwischen den Tasks und versucht beim Eintreffen neuer Jobs einen gültigen Schedule mit den bereits „garantierten Tasks“ zu finden. Dabei kommen Heuristiken zum Einsatz.

Die Auslagerung des gesamten Schedulingverfahrens für EDF und LLF in programmierbare Hardware wurde in [51] behandelt. Der Vorteil dieser Lösung ist das exakt vorhersagbare Laufzeitverhalten des Schedulingalgorithmus. Allerdings muss die Hardware für jede Änderung der Softwarearchitektur erneut programmiert werden.

In der Praxis und in den wissenschaftlichen Arbeiten stellen Untersuchungen zur hardwarebasierten Unterstützung von Schedulingalgorithmen dieser Art keinen eigenen Schwerpunkt dar.

Coprozessorgestützte Architekturen werden vor allem bei anwendungsspezifischen Applikationen mit hohem Parallelisierungsgrad verwendet. Spezielle Hardwarefunktionen (für die digitale Signalverarbeitung) werden ebenfalls auf heterogenen Multiprozessorarchitekturen verarbeitet. In der Sicherheitstechnik werden bei *Fail–Safe Systemen* unterschiedliche physikalische Hardwarearchitekturen verwendet, um einen Ausfall oder eine Fehlfunktion einer Software- oder Hardwarekomponente zu erkennen und abzufangen.

2.4.1 Ansatz dieser Arbeit – Parallele Überlastüberwachung

In dieser Arbeit wurde die Auslagerung der Überlastüberwachung auf einen Coprozessor untersucht. Das EDF–basierte TEDF Scheduling wird weiterhin vom Realzeitbetriebssystem des Anwendungsprozessors durchgeführt. Der Coprozessor wird über neu eintreffende Transaktionen informiert und besitzt somit ein Abbild des aktuellen Laufzeitsystems. Führt eine neu eintreffende Transaktion zu einer Überlast, so kann dieser den Hauptscheduler zu Eingriffen in das laufende Transaktionsset auffordern. Durch die Verlängerung von Deadlines aktuell im System befindlicher Transaktionen soll eine Überlast verhindert werden. In diesem Überlastmodus werden alle neu eintreffenden Transaktionen mit vorab spezifizierten *Pattern* in das System aufgenommen.

Die Auslagerung ist durch die folgende Betrachtung motiviert: In [85] wird von Zlokapa ein Verfahren zur Berechnung des *Punctual Point* vorgestellt. Der *Punctual Point* bezeichnet dabei einen optimalen Zeitpunkt, an dem eine Task nach ihrem Ankunftszeitpunkt in das Laufzeitsystem zugelassen wird. Das Optimierungsziel ist die Minimierung der Context Switches im Laufzeitsystem. In [79] wird die Aussage unterstrichen, dass der Ereigniszeitpunkt nicht mit dem Eingriffszeitpunkt zusammenfallen muss.

In dieser Arbeit wird eine Umschaltung in den Überlastmodus solange verzögert, wie das System durch eine Umschaltung in den Überlastmodus noch in einen spezifizierten und robusten Zustand überführt werden kann. Der Algorithmus auf dem Coprozessor kann daher asynchron zum Hauptprozessor die Überprüfungsintervalle wählen.

Im Unterlastfall wird der Hauptprozessor somit nicht durch eine ständige Überprüfung des Systemzustandes belastet.

Die für eine Überlastüberwachung benötigte Rechenzeit auf einer parallelen Architektur ist unabhängig von der Taktfrequenz und Leistungsfähigkeit des Anwendungsprozessors. Die entscheidenden Größen hierfür sind die Aktivierungsfrequenz der Softwaremodelle durch die Interaktion mit dem einbettenden System und die Anzahl der zu überwachenden Transaktionen. Wie in Abbildung 2.6 dargestellt ist, liegen die minimalen Ereignisabstände für die relevanten Klassen *QoS* und SRT zwischen $300\mu s$ und mehreren Sekunden. Das bedeutet insbesondere auch, dass auf dem Coprozessor zwischen $150\mu s$ und $500\mu s$ für eine Analyse zur Verfügung stehen.

Desweiteren sprechen noch folgende Punkte für eine hardwarebasierte Überlastüberwachung:

- Sehr genau prädizierbare Laufzeiten für den Überlastüberwachungsalgorithmus bei einfacher Coprozessorarchitektur.
- Optimierte Implementierung der Überlastüberwachung, da der Coprozessor ausschließlich für die Überwachung zur Verfügung steht.
- Software– und hardwaretechnische Trennung von funktionalen (TEDF) und parafunktionalen Aspekten (*QoS*).
- Verwendung von einfachen und günstigen Ausführungseinheiten (CPUTime) für die *QoS* Überwachung.

Die Einbindung eines Coprozessor in ein Multiprozessor *SOC* Entwurf ist technisch kein Problem. Der erhöhte Hardwareaufwand ist jedoch im Einzelfall zu prüfen.

2.5 Zusammenfassung der Ansätze

Das in dieser Arbeit beschriebene Konzept setzt sich dabei aus vier Kernpunkten zusammen:

- Eine effiziente Berücksichtigung von Präzedenzrelationen in Softwaremodellen sowohl für das Nachweisverfahren als auch zur Laufzeit (TEDF).
- Ein Laufzeitsystem, dass die Integration von harten, *QoS*– und *Soft*–Transaktionen ermöglicht und ein robustes und eindeutig spezifziertes Verhalten im Überlastfall realisiert (RTEDF).
- Eine Integration des Laufzeitsystems in ein bestehenden Entwurfsfluss durch einen parallel verlaufenden Meta–Entwurfsfluss zur Extraktion und Implementierung parafunktionaler Anforderungen.
- Die Auslagerung der Überlastüberwachung auf eine externe Ausführungseinheit, die für eine effiziente und getrennte Implementierung des RTEDF Algorithmus dient.

Die effiziente Berücksichtigung von semantischen Präzedenzrelationen im Laufzeitsystem besteht in der dynamischen Weiterleitung der absoluten Deadline während der Abarbeitung einer Transaktion. Dieses Verfahren sorgt dafür, dass abhängige Tasks in einer Transaktion immer die optimale Priorität (Deadline) erhalten und wird als *Transaktionsbasiertes EDF Scheduling* definiert. Ziel dieser Arbeit ist es, ein Laufzeitsystem für hybride Realzeitsysteme mit sporadischen Aktivierungen und variierenden Rechenzeitanforderungen zur Verfügung zu stellen.

Die Unterstützung von zeitlichen Anforderungen für harte, *QoS*– und *Soft*–Tasks gleichermaßen, wird durch den RTEDF Algorithmus bereitgestellt. Im Unterlastfall sollen die zur Laufzeit entstehenden Slack Zeiten (durch Variation von Ausführungszeiten, nicht entstehenden Blockierungszeiten und Variation von Ereignisabständen) zur Erzielung einer hohen Qualität von *QoS*–/*Soft*–Tasks genutzt werden. Im Überlastfall hingegen soll für diese Taskklassen eine vorab definierte Reduktion der Qualität mit *QoS*–Pattern angewendet werden, so dass ein robustes Verhalten gewährleistet ist. Dabei soll mit *QoS*–Pattern definiert werden, wie oft hintereinander eine Task mit einer verlängerten Deadline (*Delta*) bearbeitet werden darf, um anschließend wieder mit einer Mindestanzahl an Ausführungen mit ihrer primären Deadline (*Firm*) abgearbeitet zu werden. Der RTEDF Ansatz kombiniert die online Analyse aus den *Best–Effort* Ansätzen mit einem offline Nachweis für *QoS*–Pattern. Im Gegensatz zu *DBP* und *(m,k) Firm*, werden definierte Pattern spezifiziert, die mit einem Nachweisverfahren zur Laufzeit garantiert werden können. Im Unterlastfall wird durch eine verzögerte Umschaltung in den Überlastmodus die Nutzung von Slackzeiten anderer Transaktionen realisiert. Das führt zu einem EDF Verhalten und somit optimalen Scheduling in der Unterlastphase. Ein weiteres Ziel ist es, den Rechenaufwand für die online Analyse soweit wie möglich durch eine Verlagerung in die offline Analyse zu minimieren.

Die Extraktion der während der Modellierung und Spezifikation entstehenden Metadaten (Transaktionspfade, Deadlines, Ressourcennutzung) in geeigneter Form, ist die Aufgabe eines parallelen Meta–Entwurfsflusses. Der in der Arbeit realisierte Ansatz ist in Kapitel 6 beschrieben.

Die Auslagerung der Überlastüberwachung (RTEDF) auf einen *loosely coupled* Coprozessor soll eine Entkopplung des Meta–Entwurfsflusses vom funktionalen Entwurf ermöglichen. Die Implementierung dieser Lösung ist in Kapitel 6 beschrieben.

3 Transaktionsbasiertes EDF Scheduling

In diesem Kapitel wird das Transaktionsbasierte Scheduling (TEDF) vorgestellt. Zunächst erfolgt eine allgemeine einleitende Betrachtung, die durch eine formale Definition vervollständigt wird. Abschnitt 3.4 behandelt sowohl die ressourcenbasierte als auch die transaktionsbasierte Blockierung. Es wird gezeigt, dass das *Deadline Inheritance Protokoll* und das *Transaction Deadline Inheritance Protokoll* hierfür eingesetzt werden können. Der korrespondierende Realzeitnachweis wird in Abschnitt 3.5 behandelt.

3.1 Transaktionsbasiertes EDF Scheduling (TEDF)

Wie in Kapitel 2 angeführt, zeigt sich die Komplexität der Spezifikation beim Deadline Scheduling vor allem bei Tasksets mit Abhängigkeiten zwischen den einzelnen Tasks.

Spezifikation relativer Deadlines

Die Spezifikation relativer Deadlines von abhängigen Tasks fällt bei einer statischen a priori Analyse pessimistisch aus. Jeder Thread erhält dabei die kürzeste relative Deadline, die er zur Laufzeit durch ein Ereignis von anderen Tasks erhalten würde. Das bedeutet insbesondere auch, dass wenn Taskabhängigkeiten berücksichtigt werden sollen, eine entsprechende Unterstützung durch Werkzeuge vorhanden sein muss.

Ineffizientes Laufzeitverhalten

Auch die statische Deadlinevergabe führt zu einem pessimistischen Laufzeitverhalten. Threads, die nicht von Ereignissen mit der kürzesten relativen Deadline angestoßen werden, müssen trotzdem mit dieser kürzesten Deadline ausgeführt werden. Das führt zu einer Beeinträchtigung der Laufzeiteffizienz, da Tasks mit unnötig *„hoher Priorität“* abgearbeitet werden und dadurch eine künstliche Verknappung der Laxity verursachen.

Realzeitnachweis

Die pessimistische Vergabe von Deadlines führt letztendlich auch zu einer pessimistischen Realzeitanalyse. Der analysierte Bedarf an Leistungsfähigkeit der Hardware liegt daher weit über dem im Durchschnitt benötigten.

3.2 Einleitende Betrachtung

Die grundlegende Idee beim Transaktionsbasierten EDF Scheduling ist die explizite Integration der Threadabhängigkeiten im Laufzeitsystem. Abhängigkeiten im Tasksystem, die zu Taskpräzedenzen führen, sind durch das Softwaremodell explizit vorgegeben. Für die Taskpräzedenzen wird ein vordefiniertes Dienstprimitiv zur Verfügung gestellt. Mit diesem Primitiv wird dann die zur Laufzeit erfolgende dynamische Vererbung der absoluten Deadline an die einzelnen Threads des Präzedenzsystems realisiert. Die beschriebene Einflussnahme auf das Laufzeitsystem wird als *transaktionsbasierte* Änderung bezeichnet, da durch die Vererbung der Deadline die semantische Sichtweise der Abarbeitung im Laufzeitsystem geändert wird. Nicht die Threads, sondern die Ereignisse, die einen Thread anstoßen, sind die Träger der Aufgabe. Threads werden somit zu Ressourcen, die von Ereignissen zur Erfüllung einer übergeordneten Aufgabe verwendet werden.

In Abbildung 3.1[1)] ist beispielhaft ein Ausschnitt eines Transaktionssystem dargestellt. Ereignisquellen wie beispielsweise Interrupt Service Routinen können Ereignisse in Form von Nachrichten generieren und an die zuständigen Threads senden. Dabei besitzt jeder Thread eine einzige Nachrichtenwarteschlange, in der die Nachrichten nach ihrer absoluter Deadline einsortiert werden. Die Funktion der Threads kann dabei mit Zustandsautomaten modelliert sein, die jederzeit von den in der Nachrichtenwarteschlange vorhandenen Nachrichten aktiviert werden können.

Für die formale Behandlung des Transaktionsmodells, wird eine abstrakte Abbildung auf einen Transaktionsgraphen verwendet (Abb. 3.2). In dem Graphen werden Ereignisse auf Kanten und Threads auf Knoten abgebildet. Die Ausführungszeit einer Aktion eines Threads hängt dabei von dem Ereignis ab, das zur Aktion geführt hat. Eine Transaktion besteht aus mehreren Pfaden durch ein Transaktionssystem, ist jedoch durch ein ausgezeichnetes Startereignis eindeutig definiert.

3.3 Formale Beschreibung

Die Verwendung der Begriffe *Task, Thread, Transaktion* und *Aktion* sollen für die weiteren Ausführungen definiert werden.

Definition 1 (Task)
Task und Aufgabe können synonym für die in der Spezifikation des Systems festgelegte Funktionalität genutzt werden[2)]. Eine Task (Aufgabe) kann aus mehreren Aufrufen verschiedener Threads zusammengesetzt sein.

Definition 2 (Thread)
Als Thread wird ein leichtgewichtiger Prozess im Kontext eines Realzeitbetriebssystems ver-

[1)] Es wird eine kombinierte Darstellung von Zustandsbeschreibungen in UML Notation mit Strukturbeschreibungen von Betriebssystemprimitiven verwendet.
[2)] Im Gegensatz zu anderen Arbeiten wird mit *Task* keine Betriebssystemtask bezeichnet.

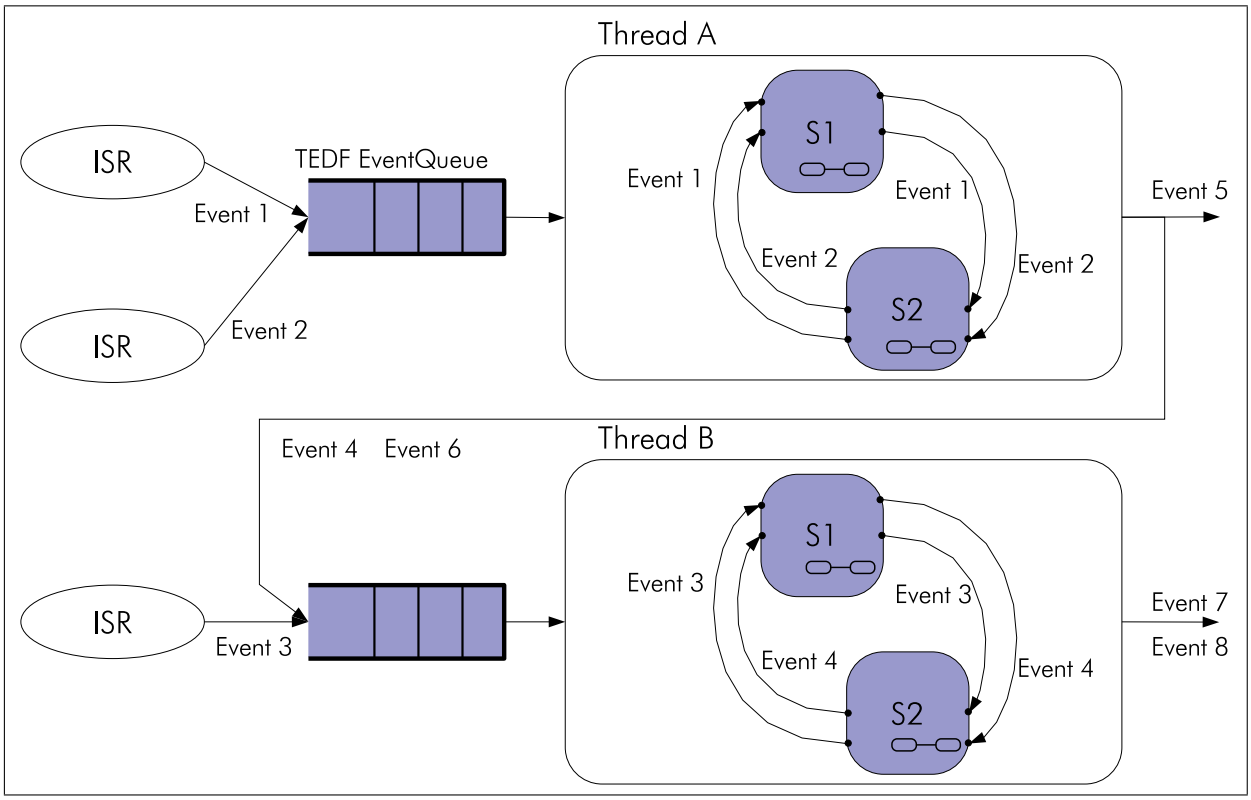


Abbildung 3.1: Schematisches Beispiel für ein Transaktionssystem

standen. Ein Thread hat in Abhängigkeit des Aktivierungsereignisses unterschiedliche Aktionen.

Definition 3 (Aktion)
Eine Aktion ist die funktionale Aktivität des Programmcodes aufgrund einer dedizierten Aktivierung eines Threads durch ein Ereignis.

Definition 4 (Transaktion)
Eine Transaktion ist die Abarbeitung einer Task durch eine Folge von Threadaufrufen im Laufzeitsystem eines Betriebssystems. Sie ist durch ein ausgezeichnetes Startereignis charakterisiert und kann mehrere Folgen von Threadaufrufen zur Laufzeit besitzen. Eine Transaktion bezeichnet daher die implementierungsrelevante Sicht einer Task.

Im Folgenden wird daher der Begriff *Task* im Kontext der Spezifikation und der Begriff *Transaktion* im Kontext der Modellierung und Implementierung verwendet.

3.3.1 Transaktionsmodell

Im Folgenden wird eine formale Beschreibung des Transaktionsmodells gegeben.

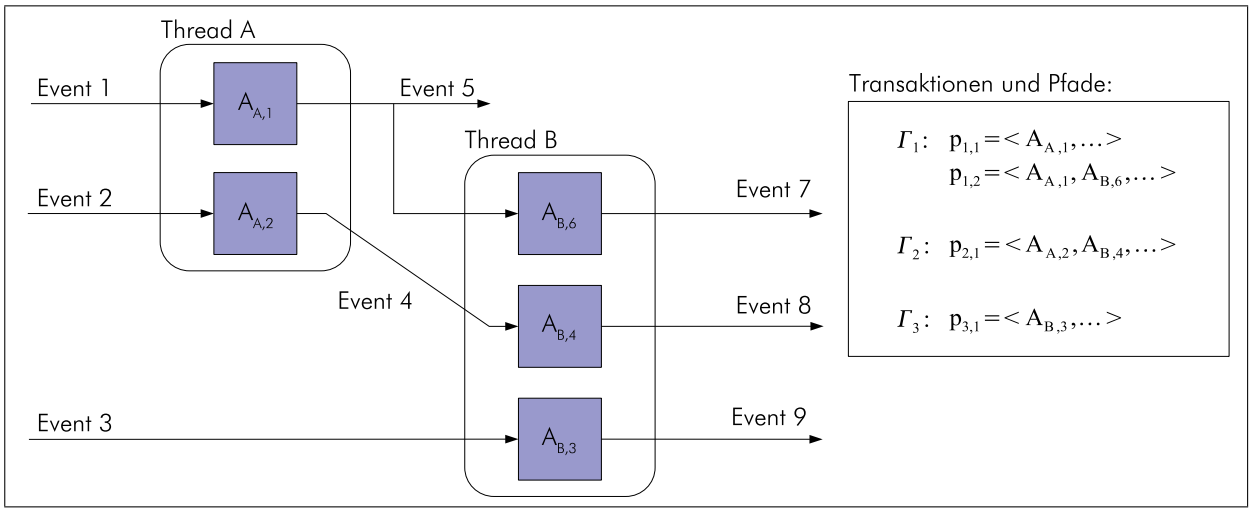


Abbildung 3.2: Abstrahiertes Transaktionssystem

Definition 5 (Transaktion)
Eine Transaktion Γ_i wird durch ein fünf–Tupel definiert

$$\Gamma_i = \{C_i, D_i, E_{Fi}, T_i, \boldsymbol{S_i}\} \qquad \textit{mit}$$

$$\begin{cases} C_i & \textit{Worst–Case Transaktionsausführungszeit} \\ D_i & \textit{relative Transaktionsdeadline} \\ E_{Fi} & \textit{Transaktionsereignisfunktion} \\ T_i & \textit{Transaktionsgraph, der alle möglichen Pfade beschreibt} \\ \boldsymbol{S_i} & \textit{Menge aller verwendeten Ressourcen} \end{cases} \tag{3.1}$$

3.3.1.1 Transaktionsgraph

Definition 6 (Transaktionsgraph)
Ein Transaktionsgraph wird definiert als ein gerichteter Graph $T = (\boldsymbol{A}, \boldsymbol{E})$, in dem $\boldsymbol{A}$ die endliche Menge aller Threadaktivierungen und $\boldsymbol{E}$ die Menge aller binärer Relationen (Ereignisse) auf $\boldsymbol{A}$ darstellen. Die Menge $\boldsymbol{E}$ enthält alle Ereignisse, die bei der Abarbeitung von Threads entstehen können und zu weiteren Threadaktivierungen führen. Ein Transaktionsgraph ist eindeutig durch sein Startereignis definiert.

Threadaktivierungen werden durch $A_{m,k}$ beschrieben, wobei m den Index des physikalischen Threads bezeichnet, zu dem diese Aktivierung gehört, und k den Index des aktivierenden Ereignisses E_k bezeichnet.

Ereignisse werden durch E_k definiert, wobei k die inzidenten Knoten bezeichnet.

Definition 7 (Transaktionspfad)
Ein einzelner Pfad $p_{i,j}$ durch den Transaktionsgraphen T_i wird als eine geordnete Sequenz von

Threadaktivierungen definiert

$$p_{i,j} =< A_{k,source}, \ldots, A_{l,m} > \quad \textit{mit} \quad A_{k,source} \prec A_{l,m}. \tag{3.2}$$

Definition 8 (Transaktionspfadmenge)
Die Anzahl aller möglichen Transaktionspfade v in einem Transaktionsgraphen wird als Transaktionspfadmenge definiert:

$$\boldsymbol{P}_i = \{p_{i,1}, \ldots, p_{i,v}\}. \tag{3.3}$$

3.3.1.2 Worst–Case Ausführungszeit

Eine feinere granulare Aufteilung der Threadaktivierungen wird in diesem Modell nicht benötigt, da Threadaktivierungen immer in Abhängigkeit vom aktivierenden Ereignis betrachtet werden. Jede Aktivierung besitzt eine *Worst–Case Ausführungszeit*, die durch eine binäre Relation definiert ist.

Definition 9 (Worst–Case Ausführungszeit)

$$\begin{aligned} WC : A \longrightarrow \mathbb{T} \quad & \textit{mit} \quad WC = \{(A, b) : A \in \boldsymbol{A} \textit{ und } b \in \mathbb{T}\} \\ & \textit{und} \quad \mathbb{T} = \{(a * b) : a \in \mathbb{N}^+ \textit{ und } b = \frac{1}{f}\,(f : \textit{Taktfrequenz}^{3)})\}. \end{aligned} \tag{3.4}$$

Die maximale Ausführungszeit einer Transaktion hängt von allen Transaktionspfaden ab, die zur Laufzeit abgearbeitet werden. Die Worst–Case Transaktionsausführungszeit wird durch denjenigen Transaktionspfad $p_{i,j}$ definiert, der in der Summe seiner Threadaktivierungen die größte Ausführungszeit verursacht.

Definition 10 (Worst Case Transaktionsausführungszeit (WCTET))
Die WCTET ist wie folgt definiert

$$C_i = \underset{\forall p_j \in \boldsymbol{P}_i}{max} \sum_{\forall A \in p_j} WC(A) \; + B_i. \tag{3.5}$$

B_i bezeichnet hier die Blockierungszeit, die vom verwendeten Prioritätsvermeidungsprotokoll abhängt. Eine Definition erfolgt in Abschnitt 3.4.

3.3.1.3 Relative Deadline D

Die Deadline definiert das vom technischen Prozess vorgegebene Zeitintervall, in dem die Bearbeitung der Transaktion abgeschlossen sein muss.

[3)] Für die Menge $\mathbb{T}$ wird von idealen Taktfrequenzen und damit von diskreten Werten für die Ausführungszeiten ausgegangen.

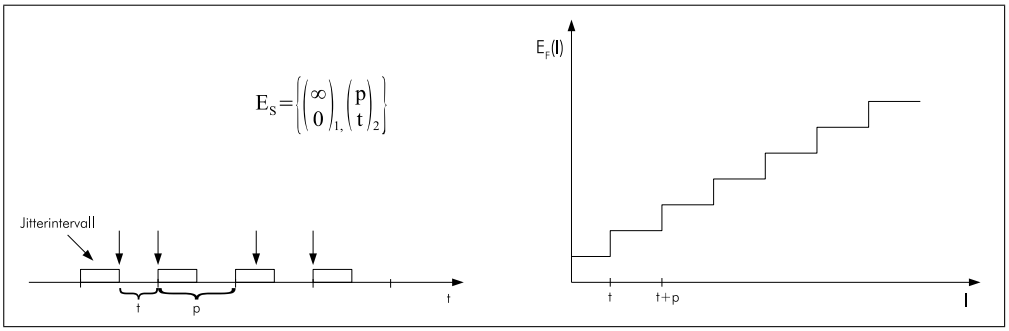


Abbildung 3.3: Beispiel für einen jitterbehafteten periodischen Ereignisstrom

3.3.1.4 Ereignisfunktion E_F

Zu jeder Transaktion wird eine Ereignisfunktion E_{F_i} definiert, die die maximale Anzahl der Ereignisse pro Intervall angibt. Die Ereignisfunktion definiert sich aus dem Ereignisstrom, der durch Ereignistupel beschrieben wird.

Definition 11 (Ereignisstrom)
Ein Ereignisstrom wird durch eine geordnete Menge von Ereignistupeln

$$E_S = \left\{ \left(\begin{pmatrix} p_k \\ a_k \end{pmatrix}_k \right) \right\} \quad \textit{mit} \quad k \in \mathbb{N}^+$$

dargestellt, deren Ordnungsindex k die Anzahl der im Interval a_k auftretenden Ereignisse definiert.

In Abbildung 3.3 ist ein periodischer und jitterbehafteter Ereignisstrom dargestellt. Aufgrund der angestrebten Periodisierung der Ereignisgruppen, gibt es mehrere Abbildungen in den Intervallbereich. Komplexere Ereignisströme führen zu einer größeren Anzahl an Ereignistupeln.

Definition 12 (Ereignisfunktion)
Die Ereignisfunktion gibt an, welche Anzahl an Ereignissen der Transaktion Γ_i pro Zeitintervall I auftreten können.

$$E_{F_i}(I) = \sum_{k=1}^{n} E_k(I) = \sum_{k=1}^{n} \begin{cases} 0 & : I < a_k \\ \left\lfloor \frac{I - a_k}{p_k} + 1 \right\rfloor & : I \geq a_k \ \wedge \ p_k < \infty \\ 1 & : I \geq a_k \ \wedge \ p_k = \infty. \end{cases} \tag{3.6}$$

Die Inverse zur Ereignisfunktion ist die Intervallfunktion.

Definition 13 (Intervallfunktion)
Die Intervallfunktion gibt das minimale Ereignisintervall an, in dem n Ereignisse auftreten können.

$$I_{F_i}(n) = E_{F_i}^{-1} \tag{3.7}$$

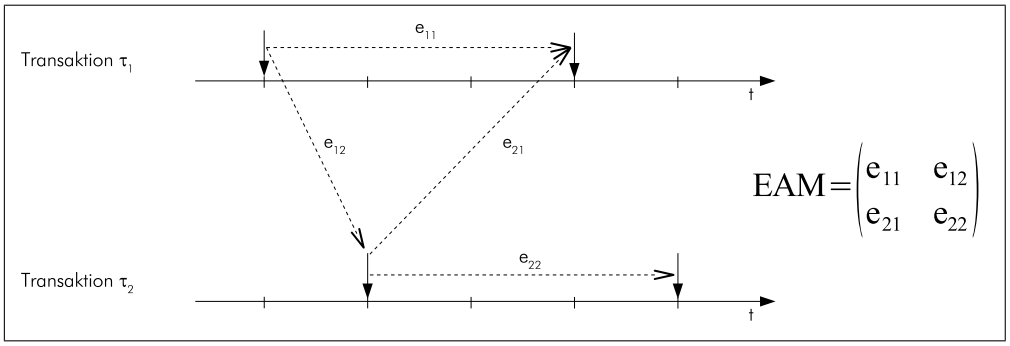


Abbildung 3.4: Beispiel für eine Abhängigkeitsmatrix zweier zeitlich abhängiger Transaktionen

3.3.2 Transaktionsabhängigkeiten

Bestehen über die Threadabhängigkeiten hinaus noch weitere Beziehungen zwischen den verschiedenen Transaktionen, werden diese Abhängigkeiten mit Hilfe einer Ereignisabhängigkeitsmatrix beschrieben. Die Definition ist analog zu der von Gresser [48] publizierten Definition.

Definition 14 (Ereignisabhängigkeitsmatrix)
Die Ereignisabhängigkeitsmatrix EAM beschreibt die zeitlichen Mindestabstände zwischen Transaktionsinstanzen

$$EAM = \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix}, \tag{3.8}$$

in der die Diagonalelemente e_{ii} die minimalen Ereignisabstände für die Transaktionen selber definieren und die Elemente e_{iy} die minimalen Ereignisabstände von Transaktion i zu Transaktion y definieren.

3.3.3 Transaktionsinstanz

Eine Instanz einer Transaktion i ist die zur Laufzeit stattfindende Abarbeitung eines bestimmten Transaktionspfades und ist definiert durch

Definition 15 (Transaktionsinstanz)

$$\tau_{i,j} = \{C_j, D_i, p_j\}. \tag{3.9}$$

C_j bezeichnet dabei die zum Pfad p_j gehörende Worst Case Ausführungszeit, die jedoch nicht mit der WCTET für die Transaktion übereinstimmen muss.

3.4 Blockierung

Als Blockierung bezeichnet man den Zustand eines Laufzeitsystems, in dem eine höherpriore Transaktion durch eine niederpriore Transaktion verzögert wird. Diese Blockierung kann bei

Realzeitsystemen genau dann auftreten, wenn exklusive Betriebsmittel von mehreren Transaktionen verwendet werden. Das ist beim Zugriff auf Ressourcen wie z. B. Bussysteme, Coprozessoren und Peripheriebausteine der Fall. Die so entstehende Blockierungszeit stellt keine aktive Rechenzeit für die blockierte Task dar, muss jedoch als solche in einem Nachweis berücksichtigt werden. Für diese Art von Blockierungsszenarien existieren bereits Prioritätsinversionsvermeidungsprotokolle wie *Deadline Inheritance* oder *Deadline Ceiling*, die den Zugriff im Konfliktfall effizient regeln und es erlauben, eine maximale obere Grenze für auftretende Blockierungszeiten zu berechnen.

Eine Blockierung, die durch die Nutzung von gemeinsamen Ressourcen entsteht, wird im Folgenden als *Ressourcenbasierte Blockierung* bezeichnet.

Im Gegensatz dazu ergeben sich bei TEDF aufgrund der Transaktionssemantik weitere Prioritätsinversionszenarien, die auf Transaktionskonflikte zurückzuführen sind. Das folgt aus der Eigenschaft von TEDF, dass Threads als Ressourcen angesehen werden, so dass ein Thread von mehreren Transaktionen aktiviert werden kann und es somit zur klassischen Prioritätsinversion kommt. Blockierungen, die durch gemeinsame Nutzung von Threads durch Transaktionen entstehen, werden im Folgenden als *Transaktionsbasierte Blockierung* bezeichnet.

In der Realzeitliteratur [79] sind Protokolle zur Vermeidung von Prioritätsinversion bei ressourcenbasierter Blockierung für EDF Scheduling bereits hinreichend beschrieben.

Im Folgenden Abschnitt werden zunächst die für die *Ressourcenbasierte Blockierung* relevanten Protokolle vorgestellt. Dabei soll zunächst das *Deadline Inheritance Protocol* für TEDF detailliert beschrieben werden. Anschließend erfolgt eine kurze Abhandlung zum *Deadline Ceiling Protokoll*.

Im Abschnitt zur *Transaktionsbasierten Blockierung* wird das *Transaction Deadline Inheritance* Protokoll beschrieben.

3.4.1 Ressourcenbasierte Blockierung

Die detaillierte Betrachtung des *Deadline Inheritance Protocol* ergibt sich aus der Notwendigkeit die bestehenden Arbeiten auf das TEDF Laufzeitsystem zu übertragen.

Für die Ressourcennutzung mit TEDF gelten folgende Voraussetzungen:

1. Ressourcen S_i können von nur einer Threadaktivität A gleichzeitig belegt werden.
2. Der k-te kritische Abschnitt einer Transaktion i wird mit $z_{i,k}$ bezeichnet. Die Rechenzeit im kritischen Abschnitt wird mit $c_{i,k}$ definiert.
3. Bei Belegung mehrerer Ressourcen müssen diese entweder geordnet verschachtelt oder sequentiell angeordnet sein.
   - Eine geordnete Schachtelung zweier kritischer Abschnitte liegt genau dann vor, wenn $z_{i,k} \subset z_{i,m} \vee z_{i,m} \subset z_{i,k}$ gilt.
   - Eine sequentielle Anordnung liegt dann vor, wenn $z_{i,k} \cap z_{i,m} = \emptyset$ gilt.

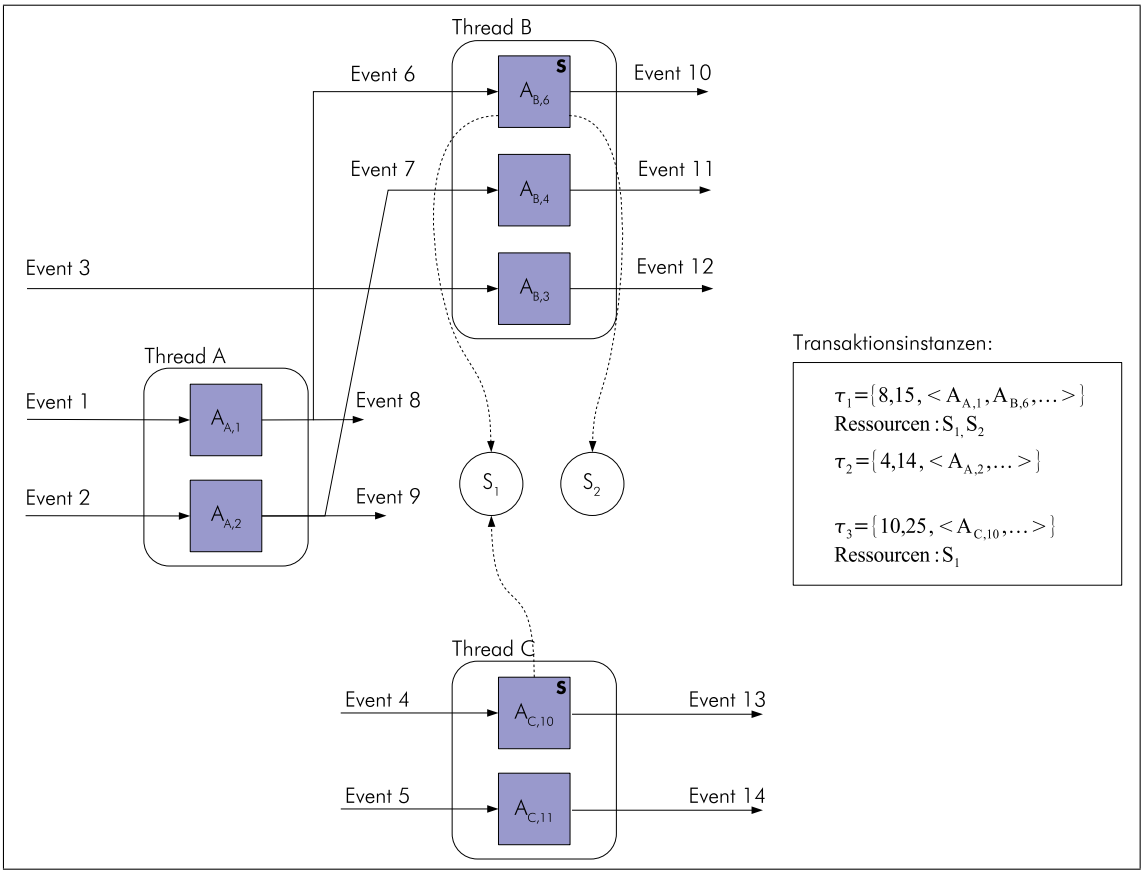


Abbildung 3.5: Transaktionssystem mit Ressourcennutzung

4. Die Reihenfolge der Freigabe der Ressourcen ist genau umgekehrt zur Belegungsreihenfolge.
5. Ressourcen können nicht über Threadgrenzen hinaus belegt werden.
6. Obwohl es sich um ein dynamisches Scheduling mit dynamischen Prioritäten handelt, ist für die Analyse die statische Betrachtung von *Unterbrechungsebenen* [30] sinnvoll. Die Unterbrechungsebene π einer Transaktion definiert sich dabei durch die Relation der relativen Deadline zur Deadline anderer Transaktionen. Je kleiner die relative Deadline, umso höher ist demnach die *Unterbrechungsebene*.

$$\pi_i > \pi_j \quad \Longleftrightarrow \quad D_i < D_j. \tag{3.10}$$

3.4.1.1 Deadline Inheritance

Eine unkontrollierte Ressourcennutzung kann dazu führen, dass sehr große Blockierungszeiten auftreten und somit Deadlineverletzungen wahrscheinlicher werden. Für EDF lässt sich mit den Ereignisfunktionen eine maximale Blockierungszeit ohne jegliche Protokolle zur Vermeidung

von Prioritätsinversion bestimmen [48] . Diese ist sowohl in der Analyse als auch zur Laufzeit sehr pessimistisch, so dass eine praktische Anwendung nur bei sehr einfachen Laufzeitsystemen sinnvoll ist.

Das *Deadline Inheritance* Protokoll reduziert sowohl die zu berücksichtigende Blockierungszeit zur Analysezeit als auch die real auftretende Blockierungszeit zur Laufzeit.

In den Abbildungen 3.6 und 3.7 ist der Vorgang einmal ohne und einmal mit Deadline Vererbung für die in Abbildung 3.5 dargestellten Transaktionsinstanzen abgebildet.

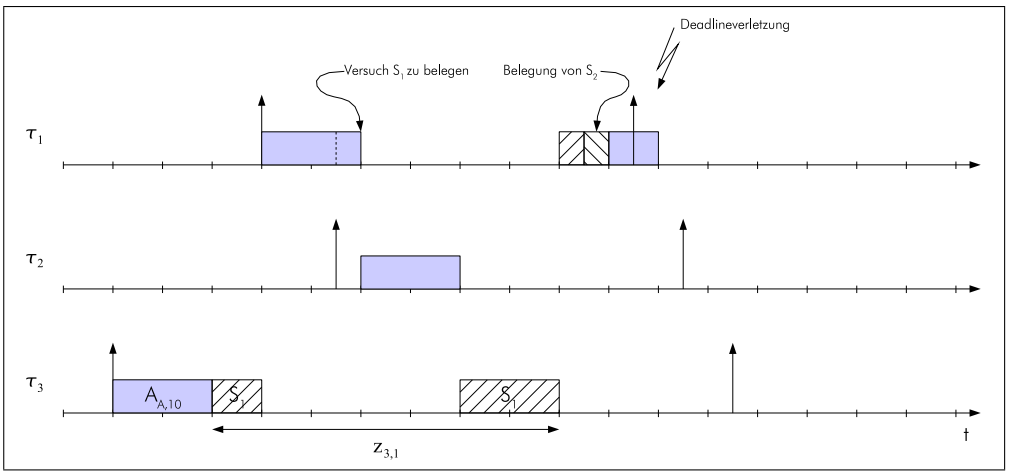


Abbildung 3.6: Deadlineverletzung durch Prioritätsinversion

Beispiel 3.1
Eine Prioritätsinversion ergibt sich bei folgendem Szenario (Abbildung 3.6): Transaktionsinstanz τ_3 befindet sich im kritischen Abschnitt $z_{3,1}$ und wird von Instanz τ_1 unterbrochen. Bei Eintreffen der Transaktionsinstanz τ_2 wird τ_1 nicht unterbrochen, da es sich um die Instanz mit der kürzesten absoluten Deadline handelt. Beim Versuch von τ_1 die Ressource S_1 zu belegen wird τ_1 blockiert und τ_2 bekommt die Rechenzeit zugewiesen. Diese Inversion führt letztendlich zur Deadlineverletzung von τ_1.

In Abbildung 3.7 ist das Szenario mit Deadlinevererbung dargestellt.

Algorithmus

Der *Deadline Inheritance* Algorithmus ist für TEDF wie folgt definiert:

- Wenn eine Transaktionsinstanz τ_i mit absoluter Deadline d_i eine Ressource S belegen möchte, so darf sie diese belegen, sofern sie nicht von einer Instanz verwendet wird. Sie wird jedoch blockiert, falls diese Ressource bereits von einer anderen Transaktionsinstanz τ_k verwendet wird.
- Im Falle einer Blockierung wird die absolute Deadline d_i an die Transaktionsinstanz τ_k vererbt, die gerade die Ressource S verwendet. Sobald τ_k die Ressource wieder freigibt, erhält τ_k wieder die Deadline vor der Vererbung und τ_i darf die Ressource belegen.

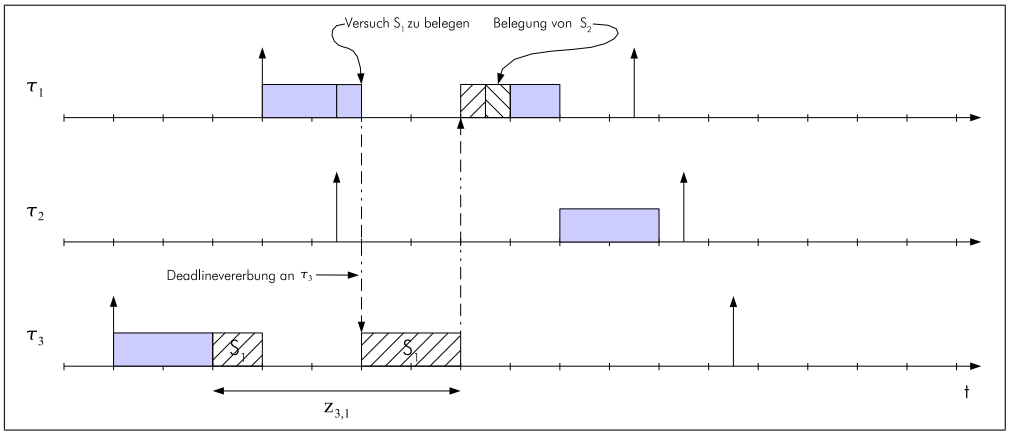


Abbildung 3.7: Deadlinevererbung

- Deadline Inheritance ist transitiv, d. h. falls während der Blockierung eine weitere Instanz auf die Ressource S zugreifen möchte, so wird wiederum ihre absolute Deadline auf die gerade belegende Transaktioninstanz τ_k vererbt.

Für eine Analyse muss offline bekannt sein, welche Transaktionen sich gegenseitig blockieren können. Obwohl die Prioritäten der Transaktionen erst zum Ereigniszeitpunkt festgelegt werden, kann trotzdem a priori eine Aussage zu Blockierungsbeziehungen zwischen Transaktionen getroffen werden. Im Folgenden werden die Startzeitpunkte mit dem Symbol s_i beschrieben.

Betrachtet man Transaktionsinstanz τ_1 und τ_3 aus Abbildung 3.6, so kann nur eine Blockierung von τ_1 durch τ_3 stattfinden.

Wenn die dynamische Priorität der Transaktionen mit $Pr(\tau_i)$ definiert wird, so können die folgenden drei Fälle unterschieden werden:

$\mathbf{Pr(\tau_1) < Pr(\tau_3)}$ Die Unterbrechungsebene π_1 ist kleiner als π_3. Daher ergeben sich zwei Fallunterscheidungen:

- $\mathbf{s_1 < s_3}$**:** τ_1 kann durch das Eintreffen von Transaktion τ_3 nicht unterbrochen und somit auch nicht im kritischen Abschnitt blockiert werden.
- $\mathbf{s_1 > s_3}$**:** Wenn τ_1 eintrifft, so wird die Ausführung von τ_3 unterbrochen. Eine Blockierung kann nun nur dann auftreten, wenn τ_3 zu diesem Zeitpunkt bereits im kritischen Abschnitt war.

$\mathbf{Pr(\tau_1) > Pr(\tau_3)}$ Hier gibt es nur den Fall, dass das Eintreffen von τ_3 zur Laufzeit von τ_1 stattfindet. τ_3 wird aber aufgrund der geringeren Priorität von τ_1 nicht unterbrochen und somit kann keine Blockierung stattfinden.

Diese Fälle führen zu folgenden Lemmata:

Lemma 1
Eine Transaktion τ_h auf einer höheren Unterbrechungsebene kann von einer Transaktion τ_l auf einer niedrigeren Unterbrechungsebene nur dann blockiert werden, wenn sich τ_l zum Zeitpunkt s_h bereits im kritischen Abschnitt befunden hat.

Beweis
Gesetzt den Fall, dass sich τ_l zum Zeitpunkt s_h nicht in einem kritischen Abschnitt befindet. Somit würde τ_l sofort durch τ_h unterbrochen. τ_l befindet sich nicht in einem kritischen Abschnitt, so dass auch keine Deadline vererbt werden könnte und somit wird τ_l erst dann weiter bearbeitet, wenn τ_h seine Abarbeitung abgeschlossen hat.

□

Lemma 2
Eine Transaktion τ_h kann von einer Transaktion τ_l nur dann blockiert werden, wenn $\pi_h > \pi_l$ gilt.

Beweis
τ_h kann durch τ_l nur dann blockiert werden, wenn τ_l sich im kritischen Abschnitt befindet. Das bedeutet, dass τ_l bereits gestartet wurde, während τ_h gerade erst eintrifft. Damit eine Unterbrechung stattfinden kann, muss $\pi_h > \pi_l$ gelten.

□

Blockierungsarten Für die Bestimmung der Dauer der kritischen Abschnitte sind zwei mögliche Blockierungsarten von Bedeutung

Direkte Blockierung Direkte Blockierung findet statt, wenn, wie in Abbildung 3.6 dargestellt, zwei Transaktionsinstanzen involviert sind, die die gleiche Ressource belegen können. τ_1 wird dabei direkt durch τ_3 blockiert.

Push Through Blockierung Dieser Fall kann für Transaktionsinstanzen entstehen, wenn sie keine gemeinsame Ressourcen mit anderen Transaktionen nutzen. τ_2 erfährt in Abbildung 3.7 eine solche Blockierung. Obwohl τ_2 keine Ressourcen verwendet, entsteht eine Blockierung, wenn τ_3 die Deadline von τ_1 im kritischen Abschnitt erbt.

Kritische Abschnitte Für die Bestimmung der Blockierungszeiten ist die Berücksichtigung aller auftretenden kritischen Abschnitte entscheidend.

Dabei muss unterschieden werden, um welche Art der Blockierung es sich handelt. Während eine direkte Blockierung relativ einfach bestimmt werden kann, ist die Analyse von *Push Through Blockierung* komplizierter.

Beispiel 3.2
Eine Transaktionsinstanz τ_1 kann aufgrund von Lemma 2 durch eine Transaktionsinstanz τ_3 in direkter Art und Weise blockiert werden, da beide Transaktionen Ressource S_1 belegen können.

Die Berücksichtigung einer möglichen Push Through Blockierung durch Transaktionsinstanz τ_3 auf τ_2 ist zunächst nicht deutlich erkennbar, da in diesem Fall sogar $\pi_2 > \pi_1 > \pi_3$ gilt. Daher gilt es auch die kritischen Abschnitte zu berücksichtigen, die alle anderen Transaktionen im System blockieren können.

Die Menge aller kritischen Abschnitte einer Transaktion Γ_m, die eine Transaktion Γ_k blockieren können, setzt sich aus einem direkten und einem *push through* Anteil zusammen:

$$\begin{aligned}\beta_{k,m} &= \beta^d_{k,m} \cup \beta^p_{k,m} \\ &= \{z_{m,p} : \pi_m < \pi_k \wedge S_{m,p} \in \boldsymbol{S_k}\} \cup \{z_{m,q} \in \beta_{r,m} : \pi_r > \pi_m\}. \end{aligned} \tag{3.11}$$

Aus der vorhergehenden Betrachtung folgt, dass die Blockierung durch den längsten kritischen Abschnitt aus $\beta_{k,m}$ erfolgen kann. Wird nämlich ein kritischer Abschnitt von einer niederprioren Transaktionsinstanz verlassen, so wird deren Abarbeitung sofort durch die höherpriore Instanz unterbrochen.

Satz 1
Jede Transaktionsinstanz einer Transaktion Γ_k kann maximal durch einen kritischen Abschnitt aus $\beta_{k,m}$ mit $1 \leq m \leq k$ mit $\pi_m \leq \pi_k$ blockiert werden.

Beweis
Der Beweis folgt aus Lemma 1 und 2.

□

Satz 1 liefert eine obere Grenze (hinreichend) für die maximal zu erwartende Blockierungszeit, wenn aus der Menge $\beta_{k,m}$ die längsten kritischen Abschnitte berücksichtigt werden.

Die maximale Blockierungszeit einer Transaktion Γ_i, die durch Ressourcennutzung von anderen Transaktionen Γ_k $(k \neq i)$ verursacht werden kann, ergibt sich aus

$$B^\beta_i = \sum_{\pi_k > \pi_i} max\{c_{k,m} : z_{k,m} \in \beta_{i,k}\}. \tag{3.12}$$

Dabei kann es vorkommen, dass in zwei Mengen $\beta_{k,m}$ und $\beta_{k,l}$ der längste kritische Abschnitt durch ein und dieselbe Ressource verursacht wird. Dieses würde in diesem Fall eine pessimistische Blockierungszeit liefern, da eine Transaktion nur ein einziges Mal aufgrund einer bestimmten Ressource blockiert werden kann.

Daher kann analog zur Gleichung 3.12 eine ressourcenbasierte Betrachtung aufgestellt werden.

$\zeta_{k,m,l}$ bezeichnet die Menge der kritischen Abschnitte aus $\beta_{k,m}$, die allein durch Belegung der Ressource S_l verursacht werden.

$$\zeta_{k,m,l} = \{z_{m,h} : z_{m,h} \in \beta_{k,m} \wedge S_{k,m} = S_l\} \tag{3.13}$$

Die Menge aller kritischen Abschnitte, die durch eine Ressource S_l verursacht werden können, sind definiert durch

$$\zeta_{k,*,l} = \bigcup_{\pi_m < \pi_k} \zeta_{k,m,l}. \tag{3.14}$$

Satz 2
Eine Transaktionsinstanz τ_i kann nur durch einen einzigen kritischen Abschnitt, verursacht durch die gemeinsame Ressourcennutzung von Ressource S_l, blockiert werden.

Beweis
Nach Lemma 1 und 2 gilt, dass eine Transaktionsinstanz τ_k nur durch eine andere Transaktionsinstanz τ_m blockiert werden kann, wenn $\pi_k > \pi_m$ gilt und wenn τ_m sich zum Zeitpunkt s_k gerade im kritischen Abschnitt befindet. Ohne Einschränkung der Allgemeinheit kann angenommen werden, dass τ_m im kritischen Abschnitt gerade Ressource S_l reserviert hat. Wird der kritische Abschnitt verlassen, so wird die Ressource entweder an τ_k gegeben oder aufgrund von *push through* Blockierungan an andere niederpriore Transaktionsinstanzen. Allerdings kann nach diesem Zeitpunkt τ_k nicht mehr durch Belegung der Ressource S_l blockiert werden.
□

Die Blockierungszeit bei einer reinen Betrachtung der Ressourcen, die sowohl zu direkter als auch *push through* Blockierung führen können, liefert die Gleichung

$$B_i^{\zeta} = \sum_{l=1}^{p} max\{c_{q,m} : z_{q,m} \in \zeta_{k,*,l}\}. \tag{3.15}$$

Dieser Ansatz liefert genau dann einen zu pessimistischen Wert, wenn eine Transaktion mehrere Ressourcen belegen kann.

Die Blockierungszeit für eine Transaktion Γ_i ergibt sich daher aus dem Minimum der beiden Gleichungen 3.12 und 3.15:

$$B_i = min\left(B_i^{\beta}, B_i^{\zeta}\right) \tag{3.16}$$

Ein synthetisches Beispiel soll die Bestimmung der Blockierungszeiten verdeutlichen.

Beispiel 3.3
Für die in Tabelle 3.1 dargestellten Transaktionen gilt $D_1 \leq D_2 \leq D_3$. Die Bestimmung der

Transaktion τ_i	Ressource S_j	kritischer Abschnitt $z_{i,k}$
τ_1	S_1, S_3	$z_{1,1}, z_{1,2}$
τ_2	S_1, S_2	$z_{2,1}, z_{2,2}$
τ_3	S_3, S_2	$z_{3,1}, z_{3,2}$

Tabelle 3.1: Beispieltransaktionsset

Anteile β^d und β^p ergibt dabei folgende Mengen.

$$\beta_{1,2}^d = \{z_{2,1}\} \qquad \beta_{1,3}^d = \{z_{3,1}\} \qquad \beta_{2,3}^d = \{z_{3,2}\}$$
$$\beta_{1,2}^p = \{\} \qquad \beta_{1,3}^p = \{z_{3,2}\} \qquad \beta_{2,3}^p = \{z_{3,1}\}$$

Daraus ergibt sich

$$\beta_{1,2} = \{z_{2,1}\} \qquad \beta_{1,3} = \{z_{3,1}, z_{3,2}\} \qquad \beta_{2,3} = \{z_{3,1}, z_{3,2}\}$$

Die ressourcenbasierte Betrachtung liefert dann

$$\begin{array}{lll} \zeta_{1,2,1} = \{z_{2,1}\} & \zeta_{1,2,2} = \{\} & \zeta_{1,2,3} = \{\} \\ \zeta_{1,3,1} = \{\} & \zeta_{1,3,2} = \{z_{3,2}\} & \zeta_{1,3,3} = \{z_{3,1}\} \\ \zeta_{2,3,1} = \{\} & \zeta_{2,3,2} = \{z_{3,2}\} & \zeta_{2,3,3} = \{z_{3,1}\} \end{array}$$

Für die zu berücksichtigenden Blockierungszeiten zeigt sich, dass der Ansatz mit den Unterbrechungsebenen die optimalen Zeiten liefert.

$$\begin{array}{ll} B_1^\beta = z_{2,1} + \{z_{3,1}|z_{3,2}\} & B_1^\zeta = z_{2,1} + z_{3,2} + z_{3,1} \\ B_2^\beta = \{z_{3,1}|z_{3,2}\} & B_2^\zeta = z_{3,2} + z_{3,1} \end{array}$$

Ein Nachteil, der durch sorgfältiges Softwaredesign und Codeanalyse ausgeschlossen werden muss, ist die Möglichkeit von Deadlocks bei Verwendung mehrerer Ressourcen. Ein weiterer Nachteil sind die in der Analyse ermittelten Blockierungszeiten. Diese fallen länger aus als die zur Laufzeit erzwungenen Blockierungszeiten beim *Deadline Ceiling*.

3.4.1.2 Deadline Ceiling

Das *Deadline Ceiling* Protokoll [73][74][68] ist ein Protokoll zur Vermeidung von Prioritätsinversion, dass zusätzlich in der Lage ist, Deadlocks zu vermeiden. Dabei wird für jede einzelne Ressource eine *Ceiling Priority* festgelegt, die sich aus den „Prioritäten“ der einzelnen Transaktionen, die diese Ressource belegen, ermittelt. Eine globale Variable *System Ceiling* erhält zur Laufzeit immer genau die *Ceiling Priority* derjenigen Ressource, die gerade verwendet wird. Daher kann eine Transaktion immer nur dann einen kritischen Abschnitt betreten, wenn sie entweder die Ressource, die das aktuelle System Ceiling ausmacht bereits belegt hat oder eine höhere „Priorität“ besitzt als das aktuelle System Ceiling.

Deadline Ceiling erlaubt eine Verringerung der in der Analyse ermittelten Blockierungszeiten gegenüber dem *Deadline Inheritance Protokoll*. Allerdings ergeben sich zur Laufzeit zwei nachteilige Effekte:

1. Die „Priorität“ von Transaktionen zueinander ist nicht offline a priori anzugeben. Daher ergibt sich das Problem, dass die *Ceiling Priorities* der Ressourcen zur Laufzeit angepasst werden müssen, was zusätzliche Rechenzeit kostet. Verfahren wie das Preemption Ceiling Protokoll versuchen diesen Effekt zu minimieren [59][29].

2. Die Blockierungsentscheidung, die auf dem globalen System Ceiling basiert, ist hinreichend aber nicht notwendig. Das bedeutet insbesondere, dass selbst wenn nach dem *Deadline Ceiling* Protokoll eine Blockierung stattfinden könnte, keine Blockierung zur

Laufzeit stattfindet. Dies ist auch der zentrale Unterschied zwischen dem *Deadline Inheritance* Protokoll und dem *Deadline Ceiling*. Während *Deadline Inheritance* erst eine Blockierung bei dem Versuch die Ressource zu belegen realisiert, wird beim *Deadline Ceiling* eine Vermeidungsblockierung durchgeführt, auch wenn die Ressource gerade nicht belegt ist. Dieser Effekt wird dadurch verstärkt, dass bei verschiedenen Pfaden einer Transaktion nicht notwendigerweise bei jeder Instanz eine Ressource belegt werden muss.

3.4.2 Transaktionsbasierte Blockierung

Beim Transaktionsbasierten Scheduling sind Threads ebenfalls als Ressourcen anzusehen. Somit zeigt sich ein „ressourcenbasiertes“ Prioritätsinversionsverhalten.

Beispiel 3.4
In Abbildung 3.8 ist eine Situation dargestellt, in der eine Transaktion τ_1 trotz kürzerer Deadline durch niederpriore Transaktionen blockiert wird. Eine Deadlinevererbung an den gerade aktiven Thread ist notwendig, damit nicht andere Transaktionen mit einer mittleren „Priorität“ eine Inversion hervorrufen.

Die Vererbung der kürzeren absoluten Deadline an den gerade aktiven Thread wird zur Laufzeit vom *Transaction Deadline Inheritance* Protokoll durchgeführt.

Für die Bestimmung der Blockierungszeiten gelten auch für diesen Fall Lemma 1 und 2. Somit sind für die Blockierungszeitbestimmung alle Aktivitäten von Interesse, die zu Transaktionen mit einer geringeren Unterbrechungsebene als die der betrachteten Transaktion gehören.

Für die Blockierungszeiten bei Verwendung des *Transaction Deadline Inheritance* Protokolls ergeben sich für die einzelnen Transaktionen folgende Blockierungszeiten

$$B_i = \max_{\forall p_j \in \boldsymbol{P}_i} \sum_{\forall k \mid A_k \in p_j} max(WC(A_k)) \tag{3.17}$$

mit

$$A_k \in \boldsymbol{P}_l \ \wedge \ l \neq i \ \wedge \ Pr(\tau_i) < Pr(\tau_l).$$

3.5 Realzeitnachweis

Für TEDF gilt ebenso wie für EDF, dass genau dann ein gültiger Schedule existiert, wenn der Gesamtrechenzeitbedarf im Worst–Case bei $U \leq 1$ liegt. Die Auslastung eines Systems definiert sich über das Verhältnis der angeforderten Rechenzeit zum Intervall, in dem die Rechenzeit angefordert wird. Dabei trägt ein Ereignis in einem betrachteten Interval $I = |t_1 - t_2|$

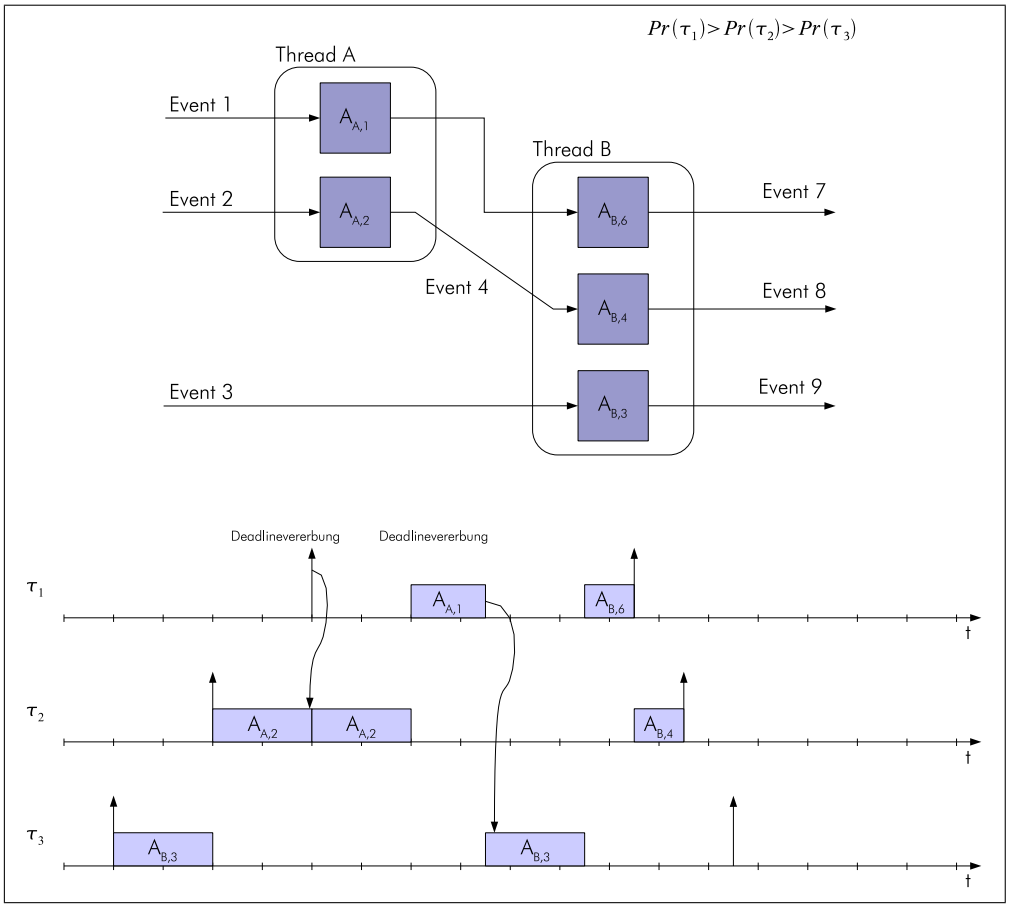


Abbildung 3.8: Beispiel für das *Transaction Deadline Inheritance* Protokoll

nur dann zur Erhöhung der Rechenzeitanforderung bei, wenn für den Ereigniszeitpunkt s und der absoluten Deadline d gilt

$$t_1 < s < t_2 \qquad \wedge \qquad d < t_2. \tag{3.18}$$

Dann gilt für ein Transaktionssystem mit n Transaktionen

$$U = \frac{\sum_{i=1}^{n} D_i(I)}{I} \tag{3.19}$$

mit

$$D_i(I) = E_{F_i}(I - D_i) \cdot C_i. \tag{3.20}$$

Die Rechenzeitanforderungsfunktion $D_i(I)$[4)] gibt dabei an, wieviel Rechenzeit im Worst Case von einer Transaktion in einem Intervall I angefordert wird. Das folgt aus der Definition der

[4)] Um eine Verwechslung mit dem Deadline Symbol zu vermeiden, wird bei der Rechenzeitanforderungsfunktion immer der Intervallparameter mitangegeben.

Ereignisfunktion, die die maximale Anzahl an Ereignissen pro Intervall einer Transaktion beschreibt. In Abbildung 3.9 ist die grafische Interpretation der Gleichung 3.19 dargestellt. Liegt

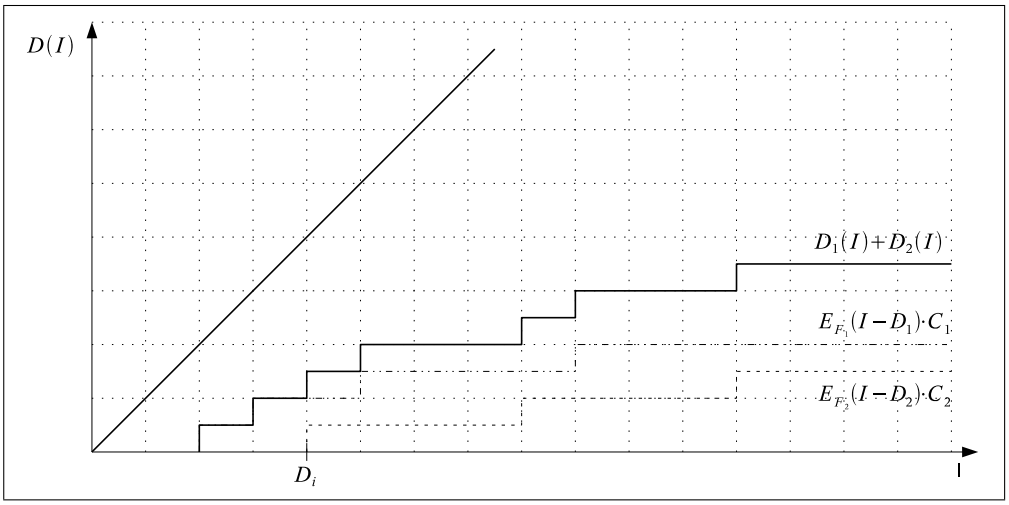


Abbildung 3.9: Grafische Darstellung der Gesamtrechenzeitanforderungsfunktion $D(I)$

die Rechenzeitanforderungsfunktion unterhalb der Winkelhalbierenden, so existiert immer ein gültiger Schedule nach EDF.

Für das transaktionsbasierte System kann das gleiche Nachweisverfahren verwendet werden, wenn Transaktionen in der Analyse wie Tasks angesehen werden. Dies lässt sich durch eine Transformation des Transaktionssystems zeigen.

Werden die einzelnen Aktivitäten einer Transaktion in einem einzigen Thread zusammengefasst, so ergibt sich keine Änderung der Laufzeitsemantik. Die zu berücksichtigenden Blockierungszeiten und die zusätzlichen Systemprimitive zur Sicherung der Datenkonsistenz bei gemeinsamen Daten führen lediglich zur Änderung der zu berücksichtigenden Rechenzeiten der einzelnen Transaktionen.

Beispiel 3.5
In Abbildung 3.10 ist eine Codetransformation eines TEDF Modells in ein EDF Modell dargestellt. Die drei Transaktionen

$$\tau_1 : p_{1,1} =< A_1, B_6 >$$
$$\tau_2 : p_{2,1} =< A_2, B_4 >$$
$$\tau_3 : p_{3,1} =< B_3 >$$

können in einzelne Tasks transformiert werden. Sowohl die Deadline als auch die zu berücksichtigende Rechenzeit bleibt dabei unverändert.

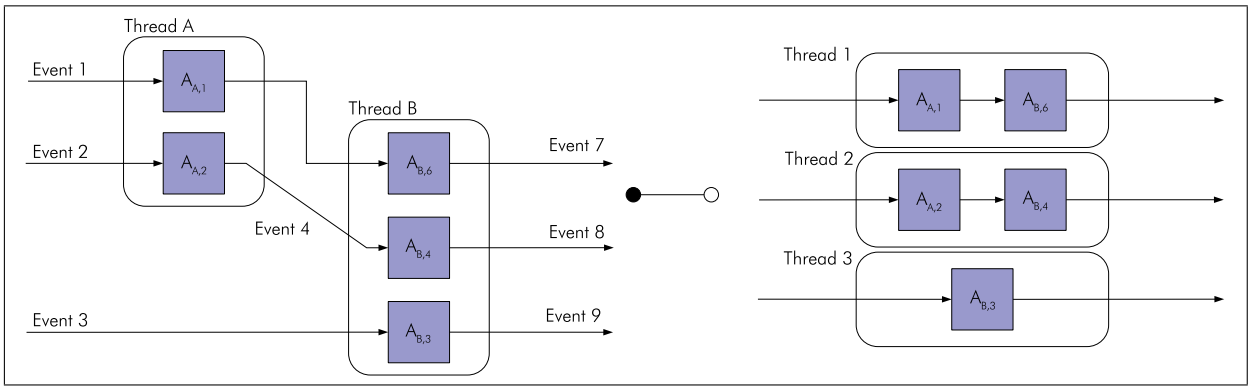


Abbildung 3.10: TEDF zu EDF Transformation

4 Robuste Erweiterung RTEDF

In diesem Kapitel wird die robuste Erweiterung zum TEDF Scheduling behandelt. Nach einer einleitenden Beschreibung zur Funktionsweise des RTEDF Algorithmus wird eine formale Betrachtung und das Nachweisverfahren beschrieben. Anschließend wird auf die Struktur und die Einbettung des Verfahrens in eine Laufzeitsystemumgebung eingegangen. Das Kapitel schließt mit einer Betrachtung der sequentiellen und parallelisierten Überlastbehandlung.

4.1 Einleitende Betrachtung

Robustes Transaktionsbasiertes EDF Scheduling stellt eine Erweiterung für die Behandlung von Überlastszenarien zum TEDF Scheduling aus Kapitel 3 dar. Die Erweiterung realisiert eine robuste Kombination von harten, weichen als auch *Quality of Service (QoS)* Aufgaben in einer Laufzeitsystemumgebung. Rechenleistung, die sowohl von weichen und von *QoS*–Realzeittasks nicht verwendet wird, soll effizient für eine individuelle Erhöhung der *Quality of Service* einzelner Tasks verwendet werden.

Im folgenden wird zunächst eine formale Beschreibung der für den RTEDF Ansatz zugelassenen Transaktionsklassen gegeben.

4.2 Formale Beschreibung

Die Menge aller Transaktionen können in drei semantisch eindeutige und disjunkte Mengen unterteilt werden, so dass gilt

$$\boldsymbol{T} = \boldsymbol{H} \cup \boldsymbol{Q_{qos}} \cup \boldsymbol{Q_{soft}}$$

$$\text{mit} \quad \begin{cases} \boldsymbol{H} = \{\Gamma : \text{harte Realzeittransaktionen}\} \\ \boldsymbol{Q_{qos}} = \{\Gamma : \textit{QoS}\text{–Transaktionen mit verlängerbarer Deadline}\} \\ \boldsymbol{Q_{soft}} = \{\Gamma : \textit{Soft}\text{–Transaktionen}\} \end{cases} \tag{4.1}$$

und

$$\boldsymbol{H} \cap \boldsymbol{Q_{qos}} \cap \boldsymbol{Q_{soft}} = \varnothing.$$

4.2.1 Harte Realzeittransaktionen

Harte Realzeittransaktionen sind durch ihre maximale Rechenzeitanforderung und ihre Deadline spezifiziert. Jede Anforderung muss innerhalb ihrer festgelegten Deadline abgearbeitet werden. Harte Realzeittransaktionen sind durch ein fünf-Tupel gekennzeichnet

$$\Gamma_{hrt,i} = \{C_i, D_i, E_{Fi}, T_i, \boldsymbol{S}_i\}$$

$$\text{mit} \quad \begin{cases} C_i & \text{Worst Case Transaktionsausführungszeit} \\ D_i & \text{relative Transaktionsdeadline} \\ E_{Fi} & \text{Transaktionsereignisfunktion} \\ T_i & \text{Transaktionsgraph, der alle möglichen Pfade beschreibt} \\ \boldsymbol{S}_i & \text{Menge aller verwendeten Ressourcen.} \end{cases} \tag{4.2}$$

4.2.2 *QoS*–Transaktionen

Softwarespezifikationen beschreiben nicht nur eine zu erbringende Funktionalität, sondern definieren auch eine Deadline innerhalb derer die Aufgabe erfüllt sein muss. Vielfach erlaubt die implementierungstechnische Realisierung oder das theoretische Systemmodell entweder eine verzögerte oder eine ausgelassene Abarbeitung einer Anforderung. Diese temporären Änderungen der Abarbeitung werden entweder durch die im Modell vorhandene Redundanz aufgefangen und somit ausgeglichen oder führen zu Einschränkungen der *Quality of Service*. Letztere Einbußen müssen im Einzelfall mit den Anforderungen aus der Spezifikation abgestimmt werden. Die Verlängerung von Transaktionsdeadlines, die gerade bearbeitet werden, entspricht der Verringerung der Priorität dieser Transaktion im Laufzeitsystem. Dadurch können bei einer aufkommenden Überlastsituation Rechenzeitanforderungen verzögert werden, so dass eine temporäre Entlastung des Laufzeitsystems herbeigeführt wird. *Deadline Extension Patterns* λ geben an,

1. um welchen Betrag eine Deadline zur Laufzeit verlängert werden kann,
2. die maximale Anzahl an konsekutiven Transaktionsinstanzen, die mit einer verlängerten Deadline abgearbeitet werden können und
3. die minimale Anzahl an Transaktionen, die anschließend ohne Deadlineverlängerung bearbeitet werden müssen.

Neue Events, die im Intervall der Deadlineerweiterung auftreten, werden nicht in das System aufgenommen sondern verworfen. Dieses Verhalten ist vergleichbar mit der Sperrung von Interrupts in kritischen Abschnitten.

Für die Klasse der Transaktionen, die eine Verlängerung der Deadline zur Laufzeit erlauben, wird folgendes Deadline Erweiterungspattern definiert:

Definition 16
Ein λ–Pattern für eine Transaktion Γ_i wird wie folgt als ein drei–Tupel definiert

$$\lambda_i = \begin{pmatrix} v_i \\ \Delta_i \\ f_i \end{pmatrix}$$

$$\textit{mit} \quad v_i, f_i \in \mathbb{N}^+, \begin{cases} v_i : \textit{Maximale Anzahl von konsekutiven Delta–Instanzen} \\ \Delta_i : \textit{Deadlineerweiterung um Intervall } \Delta_i \\ f_i : \textit{Konsekutive harte Transaktionen nach Delta–Instanzen.} \end{cases} \tag{4.3}$$

Beispiel 4.1
In Abbildung 4.1 ist die Anwendung eines λ–Patterns auf eine Ereignisfolge einer Transaktion dargestellt.

Zum Zeitpunkt a_1 wird eine neue QoS *Anforderung an das Laufzeitsystem gestellt. Das Laufzeitsystem befindet sich entweder bereits in der Überlastphase oder es wird eine Überlastsituation eben durch diese Anforderung verursacht. Aufgrund der Überlastphase wird für diese Transaktion ihr λ–Pattern aktiviert. Die erste Instanz wird zu einer Delta–Instanz, indem zur bestehenden regulären Deadline der Delta Betrag hinzuaddiert wird. Anforderung derselben Instanz innerhalb dieses Delta Fensters werden verworfen. Befindet sich das Laufzeitsystem zum Zeitpunkt a_2 weiterhin im Überlastmodus, so wird auch die nachfolgende Rechenzeitanforderung zu einer Delta–Instanz mit der erweiterten Deadline transformiert. Nach spezifiziertem λ–Pattern sind maximal zwei hintereinanderfolgende Transaktionsinstanzen als Delta–Instanzen zugelassen. Das bedeutet insbesondere, dass die nächste reguläre Anforderung zum Zeitpunkt a_3 eine harte Transaktionsinstanz (firm) sein muss. Dabei spielt es keine Rolle, ob sich das System immer noch in der Überlastphase oder aber bereits im Normalbetrieb befindet. Nach dieser Firm–Instanz ist das Pattern abgearbeitet und wird erst bei einer zukünftigen Überlastphase erneut aktiviert.*

$\boldsymbol{Q_{qos}}$ Transaktionen sind durch ein sechs-Tupel definiert:

$$\Gamma_{qos,i} = \{C_i, D_i, E_{Fi}, T_i, \lambda_i, \boldsymbol{S_i}\}.$$

Ein λ–Pattern kann beliebig oft hintereinander auf einen eintreffenden Ereignisstrom angewendet werden.

4.2.3 *Soft*–Transaktionen

Soft–Transaktionen stellen einen Sonderfall der *QoS*–Transaktionen dar. Die Parameter f_i und v_i sind wie folgt vorgegeben: $f_i = 0$ und $v_i = \infty$. Damit werden keine Anforderungen an eine bestimmte Anzahl oder Pattern von Transaktionsinstanzen mit primärer Deadline gestellt. Allerdings besitzen sie im Gegensatz zu *Non Real–Time Tasks* eine primäre und eine Delta–Deadline.

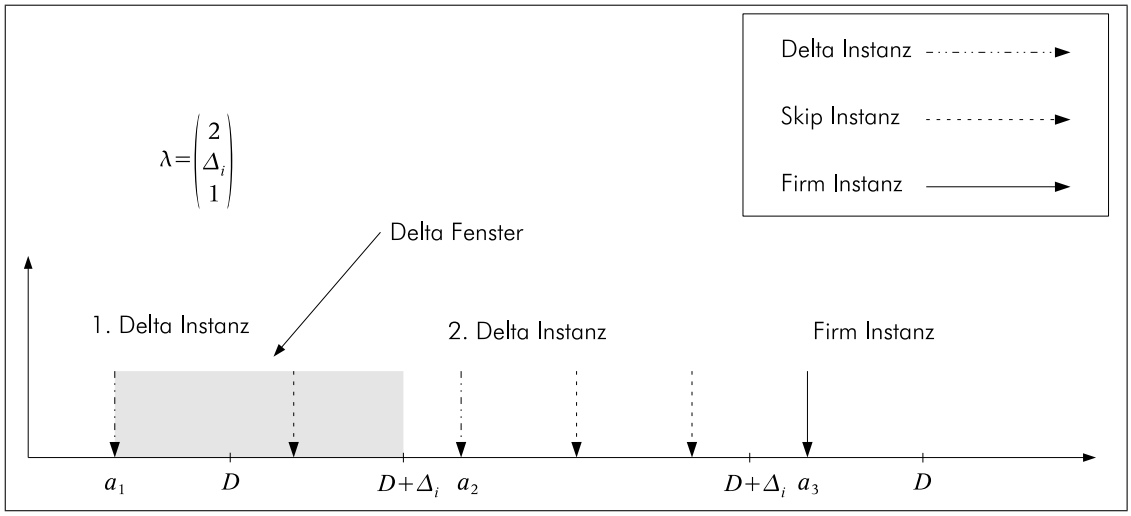


Abbildung 4.1: Ereignisfolge mit Anwendung eines λ–Patterns

Die Spezifikation der Überlastbehandlung von *Soft*– Transaktionsinstanzen wird mit einem ψ–Pattern beschrieben.

Definition 17
Ein ψ–Pattern wird wie folgt als ein drei–Tupel definiert:

$$\psi_i = \begin{pmatrix} \infty \\ \Delta_i \\ 0 \end{pmatrix} \quad \textit{mit} \quad \Delta_i : \textit{Deadlineerweiterung um Intervall } \Delta_i. \tag{4.4}$$

$\boldsymbol{Q_{soft}}$ Transaktionen sind durch ein sechs-Tupel definiert:

$$\Gamma_{soft,i} = \{C_i, D_i, E_{Fi}, T_i, \psi_i, \boldsymbol{S_i}\}.$$

4.3 RTEDF Laufzeitsystem

Das RTEDF Laufzeitsystem erlaubt eine uneingeschränkte Kombination der definierten Transaktionsklassen. Die Behandlung der unterschiedlichen Transaktionsklassen im Realzeitnachweis wird im Folgenden beschrieben.

4.3.1 Harte Transaktionen

Es wird davon ausgegangen, dass Transaktionen, die in keinster Weise ihre Deadline verletzen dürfen, sowohl eine strikte Periodizität als auch eine sehr geringe Ausführungszeitdifferenz zwischen WCTET und BCTET aufweisen. Aufgrund dieser Eigenschaften lassen sich die Rechenzeitanforderungen dieser Transaktionen sehr genau durch Geraden approximieren. Die somit zu berücksichtigende Rechenzeitanforderung pro Intervall ist konstant und führt zu einer

modifizierten maximal möglichen Rechenzeitanforderungsfunktion $D'(I)$. In Abbildung 4.2 ist dies grafisch dargestellt. Die Berücksichtigung der harten Transaktionen ist nur im Nachweis explizit durchzuführen. Im Laufzeitsystem können alle Transaktionsklassen zur selben Zeit koexistieren.

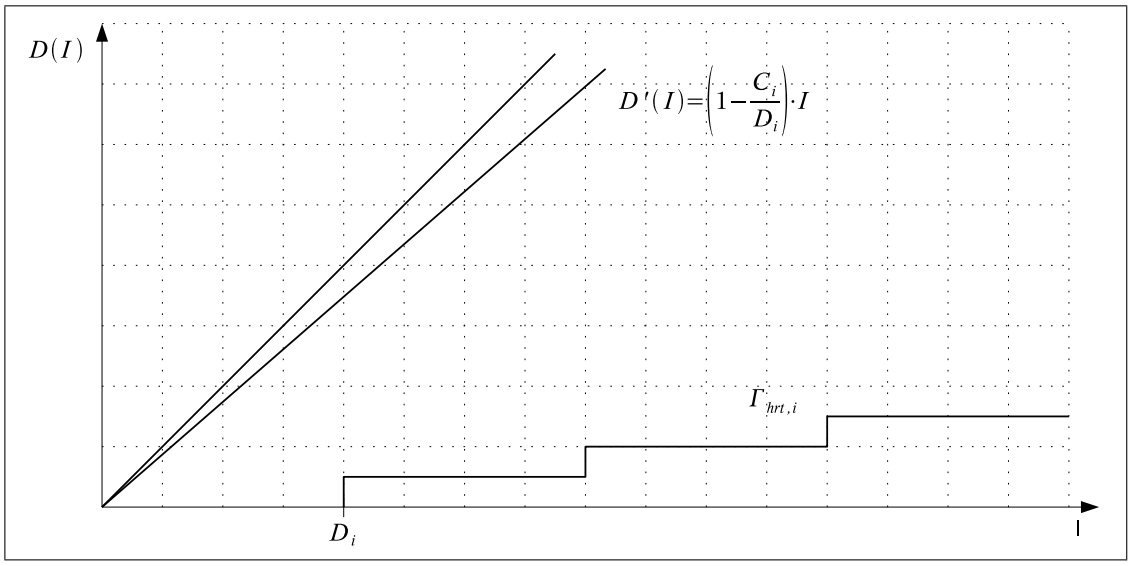


Abbildung 4.2: Berücksichtigung der harten Transaktionen beim Realzeitnachweis

Aufgrund der speziellen Betrachtung der harten Transaktionen im Realzeitnachweis können im Folgenden diese Transaktionen für die Analyse und Definition der Überlastbehandlung ausgeklammert werden.

4.3.2 *QoS–/Soft–*Transaktionen

Bei diesen Transaktionsklassen wird davon ausgegangen, dass der Unterschied zwischen den für die Worst Case Analyse verwendeten Rechenzeiten und den tatsächlich zur Laufzeit anfallenden Anforderungen in der Praxis durchaus groß sein kann. Desweiteren wird davon ausgegangen, dass die Aktivierungszeitpunkte einer stärkeren zeitlichen Variation aufgrund zustandsbasierter Ausführung unterworfen sind. Für beide Transaktionsklassen müssen Pattern (λ, ψ) definiert werden, um bei auftretenden Überlastszenarien zur Laufzeit auf die vordefinierten Qualitätsreduktionen umschalten zu können. Diese Eigenschaft ermöglicht eine Entlastung der Gesamtrechenzeitanforderung in *Worst Case* Szenarien und erlaubt somit die Dimensionierung von Systemen mit weniger leistungsfähigeren Systemressourcen.

Die nachweistechnische Trennung und die laufzeittechnische Kombination der harten und *QoS–/Soft–*Transaktionen ist aus zwei Gründen sinnvoll:

1. Die nachweistechnische Trennung der harten Transaktionen von den *QoS–/Soft–*Transaktionen ist dadurch begründet, dass die harten Transaktionen nicht zu einem nennenswerten nutzbaren Slack beitragen. Das liegt vor allem an der geringen Variation der Ereigniszeitpunkte als auch den fast konstanten Ausführungszeiten.

2. Die Nutzung von gemeinsamen Ressourcen auf dem Laufzeitsystem soll ermöglicht werden, so dass jeweils Blockierungszeiten durch Protokolle zur Vermeidung von Prioritätsinversion berücksichtigt werden können.

4.3.3 RTEDF Ansatz

Der Ansatz, der dazu in dieser Arbeit verfolgt wird, stützt sich auf eine modifizierte Zugangskontrolle (*Acceptance Testing*) für die *QoS–*/*Soft–*Transaktionen.

Die Zugangskontrolle, die durch den RTEDF Algorithmus realisiert wird, überprüft dabei, ob eine neu startende Transaktion sowohl innerhalb ihrer spezifizierten Deadline abgearbeitet werden kann als auch andere bereits im Laufzeitsystem befindlichen Transaktionen nicht über ihre Deadline hinaus verzögert.

Solange keine Überlast detektiert wird, werden alle Rechenzeitanforderungen angenommen und abgearbeitet. Das System befindet sich dabei im *Normalmodus* und verhält sich wie ein normales TEDF System ohne Überlastbehandlung.

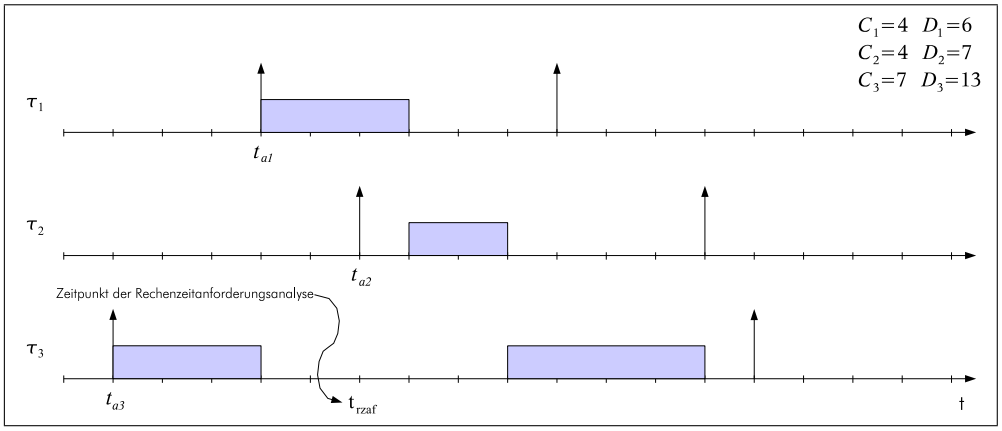
Wird bei neu eintreffenden Rechenzeitanforderungen eine mögliche Überlast detektiert, so wird eine Strategie angewendet, die den enstehenden Slack aus den Variationen der Ausführungszeiten optimal nutzen kann.

Beispiel 4.2 verdeutlicht den Gedankengang.

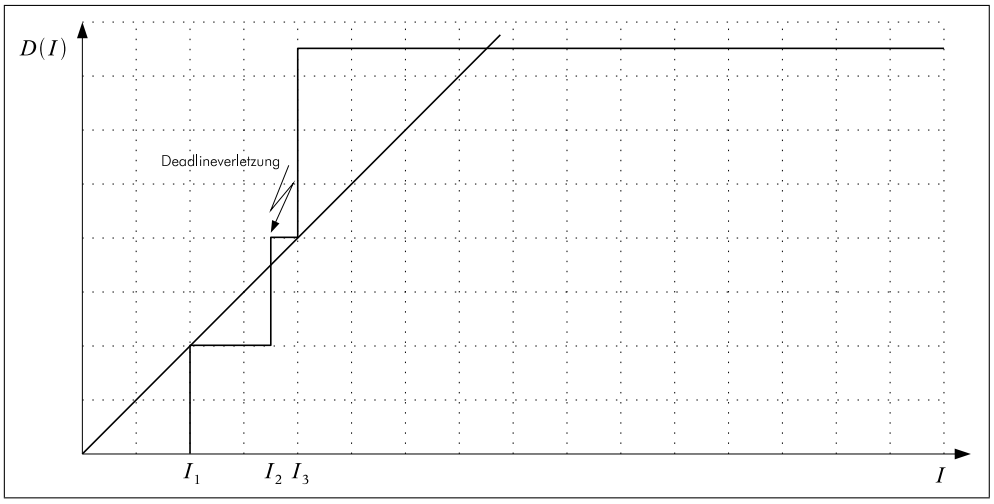
Beispiel 4.2
In Abbildung 4.3(a) sind drei Transaktionen mit Deadline und Worst Case Ausführungszeit dargestellt. Bei jeder neu eintreffenden Transaktion wird eine Rechenzeitanforderungsanalyse durchgeführt. Die Rechenzeitanforderungsanalyse beim Eintreffen der Transaktionsinstanz τ_2 zum Zeitpunkt t_{a2} ist in Abbildung 4.3(b) dargestellt. Die Intervalle, die in der Analyse verwendet werden ergeben sich zu $I_x = t_{ax} + D_x - t_{rzaf}$. Zu diesem Zeitpunkt liefert die Analyse die Information, dass sowohl Transaktionsinstanz τ_2 als auch τ_3 ihre Deadlines nicht einhalten werden können, wenn jeweils die maximalen Ausführungszeiten auftreten werden. Sollten die maximalen Ausführungszeiten nicht auftreten, so wäre eine Umschaltung in den Überlastmodus nicht nötig. Im Beispiel in Abbildung 4.3(a) tritt aufgrund der real kürzeren Ausführungszeiten keine Deadlineverletzung mehr auf.

Die in dieser Arbeit realisierte Variante sieht eine verzögerte Umschaltung in den Überlastmodus vor. Dazu wird zum Zeitpunkt t_{rzaf} aus Beispiel 4.2 zunächst im Idealfall für die Dauer des Intervalls I_2 mit einer Entscheidung gewartet. Dieses Vorgehen ist insofern zulässig, als dass Transaktion t_1 ihre Deadline auf jeden Fall einhalten wird. Falls im Intervall I_2 keine neue Transaktion eingetroffen sein sollte, wird erneut eine Rechenzeitanforderungsanalyse durchgeführt. Geht man davon aus, dass das Ende der Transaktion τ_3 bereits zum Analysezeitpunkt vorliegt, so zeigt sich, dass keine Überlast mehr vorliegt. Eine Umschaltung in den Überlastmodus war somit nicht mehr notwendig.

Sollte nach einem Verzögerungsintervall I_x weiterhin eine Überlast vorliegen, so muss mit den vorab spezifizierten Pattern in das aktuelle Laufzeitsystem eingegriffen werden. Im Falle einer



(a) Aktivierungsszenario für drei Transaktionen



(b) Gesamtrechenzeitanforderungen für das Szenario zum Zeitpunkt t_{rzaf}

Abbildung 4.3: Aktivierungsszenario für drei Beispieltransaktionen und Gesamtrechenzeitbedarf zum Zeitpunkt t_{rzaf}

Überlast werden für die *QoS*–Transaktionen die vorab definierten λ– und für weiche Transaktionen die ψ–Pattern[1)] angewendet. Das Laufzeitsystem befindet sich dann bis zur nächsten *Idle Time* im Überlastmodus.

Die Idee der verzögerten Überlastbehandlung ist verwandt mit dem *Punctual Point* Verfahren, das von Zlokapa in [85] vorgestellt wurde. Der *Punctual Point* bezeichnet dabei einen optimalen Zeitpunkt, an dem eine Task nach ihrem Ankunftszeitpunkt in das Laufzeitsystem zugelassen wird. Das Optimierungsziel ist die Minimierung der Context Switches im Laufzeitsystem.

Nach einer *Idle Time* befindet sich das System wieder im *Normalmodus*. Der Normalmodus

[1)] siehe Abschnitt 4.2.2 und 4.2.3

wird nur verlassen, wenn

1. nach einem Verzögerungsintervall die Überlast noch vorhanden ist oder
2. wenn einige Transaktionen ihre firmen Instanzen laut Patternspezifikation nocht nicht erfüllt haben.

Ziel dieser robusten Überlastbehandlung ist es, für den Systemzustand der Unterlast eine möglichst hohe „Qualität“ zu erreichen, während im *Worst Case* Fall die Qualität auf die vordefinierten Pattern degradiert wird.

TEDF gehört wie EDF zu den dynamischen Schedulingverfahren mit dynamischen Prioritäten. Die Anwendung einer Überlastbehandlungsstrategie ist daher nur zur Laufzeit möglich. Die Prozessbeschreibung des Algorithmus ist in Abbildung 4.5 dargestellt.

4.3.4 Struktur und Verhalten

Für die Realisierung des RTEDF Algorithmus sind Erweiterungen des TEDF Laufzeitsystems erforderlich. Abbildung 4.4 zeigt ein abstraktes Strukturbild der Architektur. Im Folgenden wird auf die strukturellen Elemente und deren Funktionalität eingegangen.

Das RTEDF System ist der zentrale Bereich für die RTEDF Überlastüberwachung. Für eine Detektion von möglichen Überlastszenarien bei neu eintreffenden Transaktionsinstanzen wird ein Abbild des aktuellen Laufzeitsystems benötigt. Die Informationen werden durch die im TEDF System arbeitende Transaktionsverwaltung aufbereitet.

4.3.4.1 Transaktionsverwaltung

Die Transaktionsverwaltung ist dem eigentlichen Laufzeitsystem vorgeschaltet. Dabei hat sie zum einen die Aufgabe, das Auftreten von Ereignissen im TEDF Laufzeitsystem an das RTEDF System zu melden. Zum anderen muss sie bei Eingriffen des RTEDF Systems zur Laufzeit die angeforderten Aktionen an Transaktionsinstanzen im laufenden System durchführen.

4.3.4.2 TEDF zu RTEDF Kommunikation

Es wird zwischen drei Arten von Ereignissen unterschieden. Startereignisse beschreiben Transaktionsereignisse, die neu im System eintreffen, während Ende–Ereignisse das Ende eine Transaktion im TEDF Laufzeitsystem signalisieren. Das *Idle Ereignis* zeigt eine Idle Zeit im Laufzeitsystem an.

Startereignisse Startereignisse von *QoS*– oder *Soft*–Transaktionsinstanzen werden in eine temporäre Warteschlange (*QoS/Soft TempQueue*) gelegt. Das RTEDF System wird über die *QoS/Soft Eventdata* Datenstruktur mit dem Primitiv **`QOS_EVENT`** informiert.

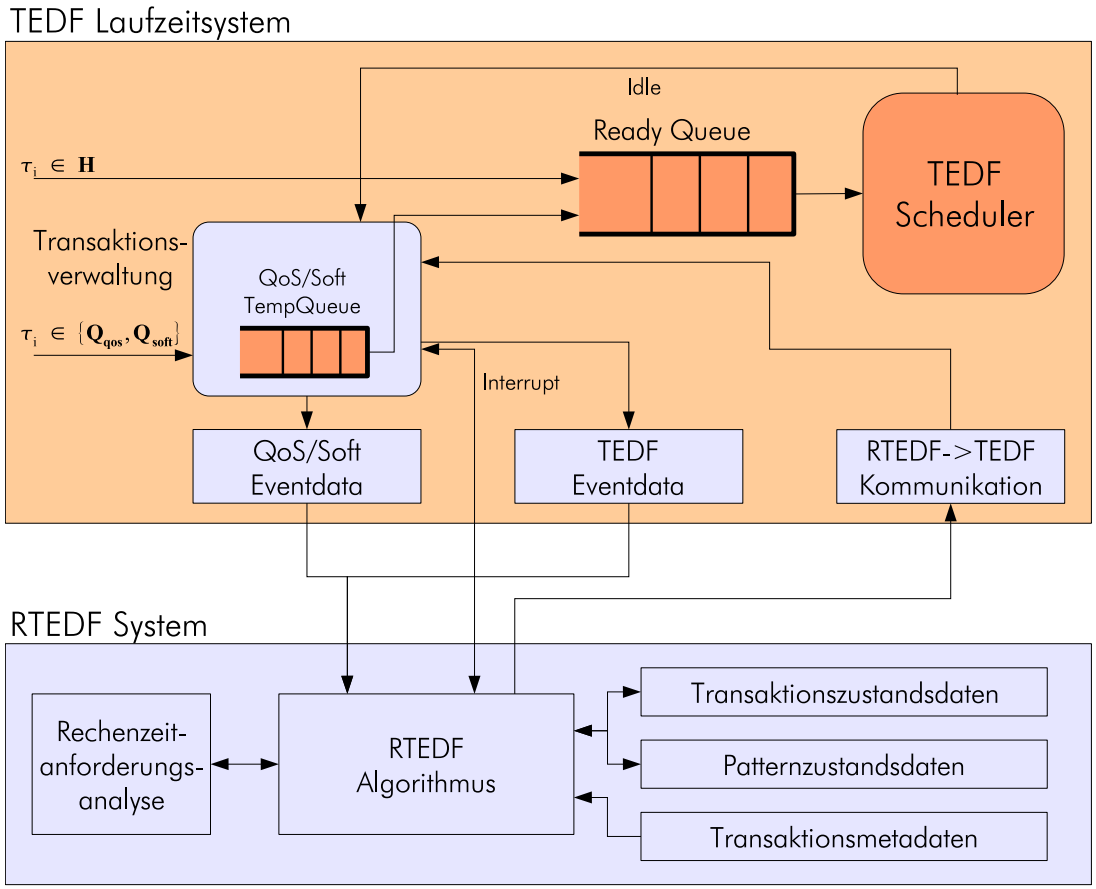


Abbildung 4.4: Strukturbild TEDF Laufzeitsystem mit RTEDF System

Ende–Ereignisse Ende–Ereignisse sowohl von harten als auch von *QoS*– und *Soft*–Transaktionsinstanzen werden mit dem Primitiv **`END_EVENT`** über die *TEDF Eventdata* Datenstruktur gesendet.

Idle Ereignis Die Detektion einer Idle Zeit wird als **`IDLE_EVENT`** über die *TEDF Eventdata* Datenstruktur gesendet.

In Tabelle 4.1 sind die Kommunikationsprimitive, die von der Transaktionsverwaltung an das RTEDF System gesendet werden, aufgestellt.

Kommunikationsprimitiv	Definition
`QOS_EVENT`	Eine neue *QoS*– oder *Soft*–Transaktionsinstanz wurde registriert. Die Zulassung ist zu überprüfen.
`END_EVENT`	Eine Instanz hat ihre Anforderung beendet.
`IDLE_EVENT`	Die *Ready Queue* hat keinen lauffähigen Thread[2)] mehr im System

Tabelle 4.1: Kommunikationsprimitive vom TEDF zum RTEDF System

[2)] Es wird davon ausgegangen, dass es keine *Self Suspension* von Threads gibt, so dass bei einer Idle Phase keine ausstehenden Rechenzeitanforderungen mehr im System vorhanden sind.

4.3.4.3 RTEDF System–Aktionen

Die Transaktionsverwaltung hat zudem die Aufgabe, Einträge aus der *Qos/Soft Eventdata* Datenstruktur zu entfernen und gegebenenfalls die Transaktionsinstanzen aus der *QoS/Soft Temp-Queue* in die *Ready Queue* einzufügen.

4.3.4.4 RTEDF System

Das RTEDF System verarbeitet die Daten aus den Datenstrukturen *QoS/Soft Eventdata* und *TEDF Eventdata* ereignisgesteuert. Liegen neue Daten vor, so werden diese in die Transaktionszustandsdatenstruktur eingetragen und es wird eine Rechenzeitanforderungsanalyse durchgeführt. Neben der RTEDF Funktionalität besteht das RTEDF System aus drei grundlegenden Datenstrukturen:

Transaktionsmetadaten (TMD) Die Transaktionsmetadaten sind Daten, die durch die Spezifikation und durch eine a priori Analyse des System ermittelt worden sind. Dazu gehören beispielsweise die WCTET, die λ– und ψ–Pattern. Die Größe dieser Datenstruktur ist konstant. Es wird zur Laufzeit nur lesend auf sie zugegriffen.

Patternzustandsdaten (PZD) In dieser Datenstruktur wird festgehalten, wie oft eine Transaktionsinstanz noch verlängert werden darf und ob die firmen Transaktionen bereits abgearbeitet wurden. Die Größe dieser Datenstruktur ist ebenfalls konstant. Sie wird zur Laufzeit sowohl gelesen als auch beschrieben.

Transaktionszustandsdaten (TZD) Für eine Rechenzeitanforderungsanalyse wird die absolute Deadline und die zu erwartende Ausführungszeit einer Transaktion benötigt. Zudem müssen diese jederzeit nach Deadline sortiert vorliegen. Die Größe dieser Datenstruktur ist konstant. Sie wird jedoch zur Laufzeit gelesen, beschrieben und muss jederzeit nach der Deadline sortiert vorliegen.

4.3.4.5 RTEDF zu TEDF Kommunikation

Wird anhand einer Rechenzeitanforderungsanalyse eine nicht transiente Überlast detektiert, so kommuniziert das RTEDF System mit der Transaktionsverwaltung über Interrupt. Die durchzuführende Aktion wird in der *RTEDF->TEDF Kommunikation* Datenstruktur abgelegt, und anschließend ein Interrupt ausgelöst. Mögliche Aktionen sind in Tabelle 4.2 definiert.

Wie in Abbildung 4.4 zu sehen ist, wird die Überlastbehandlung für den Scheduler transparent durchgeführt. Sie ist daher nicht mit dem Scheduling Algorithmus verwoben. Es wird vom TEDF Scheduler lediglich die Detektion des Idle Zustandes benötigt.

Auf die implementierungstechnische Realisierung der Struktur wird in Kapitel 6 näher eingegangen.

Kommunikationsprimitiv	Definition
EXTEND_ALL	Alle *QoS*– und *Soft*–Transaktionsinstanzen, die sich aktuell im Laufzeitsystem befinden, müssen erweitert werden, sofern sie nicht bereits erweitert sind. Firm–Instanzen werden nicht verändert.
INSERT_NORMAL	Eine *QoS*– oder *Soft*– Transaktionsinstanz kann mit ihrer primären Deadline ins Laufzeitsystem aufgenommen werden.
INSERT_EXTENDED	Eine Transaktion ist als Delta–Instanz einzufügen.
SKIP	Eine Transaktion ist zu verwerfen.
INSERT_FIRM	Eine Transaktion ist als Firm– Instanz einzufügen.

Tabelle 4.2: Kommunikationsprimitive vom RTEDF zum TEDF System

4.3.5 RTEDF Algorithmus

Der RTEDF Algorithmus wird immer nur abgearbeitet, wenn eine Anforderung von einer der beiden externen Ereignisquellen vorliegt. Dabei wird unterschieden, ob sich das Laufzeitsystem im Überlastmodus (Overload) oder im Normalmodus (Normal) befindet. Dieser Modus wird solange beibehalten, wie keine Idle Time aufgetreten ist. Nach einer Idle Time im Laufzeitsystem werden *QoS*–Transaktionen, die ihrem Pattern genügen, nach einer Laxityüberprüfung in das System aufgenommen. Das geschieht auch dann, wenn es andere Transaktionen gibt, die noch firme Transaktionen ausstehen haben und sich zum gegenwärtigen Zeitpunkt nicht im Laufzeitsystem befinden.

In Abbildung 4.5 ist die Basisstruktur des RTEDF Algorithmus dargestellt. Die Unterroutinen *Overload Handler()*, *QoS Input Handler()* und *QoS Data Handler()* sind in den Abbildungen 4.6, 4.7 und 4.8 dargestellt.

Auf die implementierungstechnische Realisierung des Algorithmus wird in Kapitel 6 näher eingegangen.

4.3.6 Komplexitätsbetrachtung

Komplexitätsbetrachtungen von Algorithmen sind im Kontext von eingebetteten Realzeitsystemen lediglich als notwendige Bedingung zur Bewertung der Einsetzbarkeit anzusehen. Zu groß sind implementierungsspezifische Unterschiede, die selbst einen linearen Algorithmus für den Einsatz disqualifizieren können. In Kapitel 6 werden zu den hier folgenden Betrachtungen Messwerte aus den konkreten Implementierungsvarianten diskutiert.

Ein Großteil der Rechenzeit des RTEDF Algorithmus wird zur Haltung des TEDF Systemzustandes in der Transaktionszustandsdatenstruktur verwendet. Für die Berechnung der Realzeitanalyse werden die Werte in einer nach Deadline aufsteigend sortierten Reihenfolge benötigt. Ein idealer *in Place*[3)] Sortieralgorithmus ist der Heapsort. Die Komplexität dieses Algorithmus

[3)] *in Place* bedeutet, dass nur eine sehr geringe und maximal begrenzte Anzahl an Elementen beim Sortiervorgang

```
   Data Input : TEDFEventdata, QoSdata;
   Data Output : RTEDFdata, RTEDF_IR;
   Data : TMD, TZD, PZD, tempstack;
   Variable(statisch) : SystemState(normal, overload), RTEDFstate(normal, tedfmode);
   Variable(lokal) : QoSAnalysis:=false, tedfdata, Deadline_Intervall,
                     RaiseInterrupt:=false;
 1 while (TEDF Eventdata ≠ ∅) do
 2     Get element tedfdata from TEDFdata;
 3     Update(TZD ∧ PZD, tedfdata);
 4     if (Event(tedfdata) == idle_event) then SystemState:=normal;
 5 if (TZD == ∅ || {∃τ_i : τ_i ∈ TZD ∧ τ_i == firm}) then
 6     RTEDFState:= normal;
 7 if (SystemState == overload) then
 8     Overload Handler();
 9 else
10     QoS Input Handler();
11 if (QoSAnalysis == true) then
12     if (RTAnalysis() == overload) then
13         if (Deadline_Intervall>0) then
14             Set Timer(Deadline_Intervall);
15             Qos Data Handler(normal);
16         else
17             RTEDFdata ⟵ EXTEND ALL;
18             Update(TZD ∧ PZD);
19             SystemState:=overload;
20             RaiseInterrupt:= true;
21             QoS Data Handler(tedfmode);
22     else
23         if RTEDFState == tedfmode then
24             QoS Data Handler(tedfmode);
25         else
26             QoS Data Handler(normal);
27         RaiseInterrupt:=true;
28 if (RaiseInterrupt == true) then RETDF_IR ⟵ raise;
```

Abbildung 4.5 : RTEDF Algorithmus

liegt im Worst Case bei $O(n \cdot lg(n))$. Selbst wenn für den durchschnittlichen Rechenzeitbedarf ein Quicksort günstiger ist, wird in diesem Fall eine maximale Laufzeitgrenze benötigt, die

außerhalb der zu sortierenden Datenstruktur abgelegt werden muss.

```
1  while tempstack ≠ ∅ do
2      Get Element qosdata from tempstack;
3      if qosdata.type == FIRM then
4          RTEDFdata ⟵ INSERT_FIRM(qosdata.id);
5      else if qosdata.type == QoS then
6          if TEDFState == tedfmode then
7              RTEDFdata ⟵ INSERT_EXTENDED(qosdata.id);
8          else
9              RTEDFdata ⟵ INSERT_NORMAL(qosdata.id);
```

Abbildung 4.6 : Unterroutine QoS Data Handler

```
1  while QoSdata ≠ ∅ do
2      Get element qosdata from QoSdata;
3      if PZD[qosdata.id] = ∅ then
4          Update(TZD, qosdata, QoS);
5          RTEDFdata ⟵ INSERT_EXTENDED(qosdata.id);
6      else if PZD[qosdata.id].deltaphase < currenttime && PZD[qosdata.id].extensions
       ≠ 0 then
7          Update(PZD, qosdata, QoS);
8          Update(TZD, qosdata, QoS);
9          RTEDFdata ⟵ INSERT_EXTENDED(qosdata.id);
10     else if PZD[qosdata.id].extensions == 0 && PZD[qosdata.id].firms ≠ 0 then
11         Update(PZD, qosdata, FIRM);
12         Update(TZD, qosdata, FIRM);
13         RTEDFdata ⟵ INSERT_FIRM(qosdata.id);
14     else if PZD[qosdata.id].deltaphase > currenttime then
15         RTEDFdata ⟵ SKIP(qosdata.id);
```

Abbildung 4.7 : Unterroutine Overload Handler

nicht mit quadratischer Komplexität behaftet ist.

Die Komplexität der Berechnung der Systemlaxity mit Hilfe der Rechenzeitanforderungsfunktion ist linear ($O(n)$) und abhängig von der maximalen Anzahl gleichzeitig im System befindlichen Transaktionen.

In Tabelle 4.3 ist eine Zusammenfassung der relevanten Operationen im RTEDF System mit der zughörigen Komplexitätsklasse dargestellt.

```
1  while QoSdata ≠ ∅ do
2      Get element qosdata from QoSdata;
3      if PZD[qosdata.id].deltaphase > currenttime then
4          RTEDFdata ⟵ SKIP(qosdata.id);
5      else
6          if PZD[qosdata.id].firms ≠ 0 then
7              Update(TZD,qosdata,FIRM);
8              tempstack ⟵ qosdata;
9              RTEDFState:= tedfmode;
10         else
11             Update(TZD, qosdata, QoS);
12             tempstack ⟵ qosdata;
13         QoSAnalysis:=true;
```

Abbildung 4.8 : Unterroutine QoS Input Handler

Datenstruktur	Operation	Komplexität
Transaktionszustandsdatenstruktur	Einfügen eines Datums in einen sortierten Heap	$O(lg(n))$
	Ausgabe eines sortierten Heaps	$O(n \cdot lg(n))$
	Berechnen der aktuellen Systemlaxity	$O(n)$
Patternzustandsdaten	Einfügen/Einlesen eines Datums	$O(n)$

Tabelle 4.3: Komplexität der Operationen auf den verwendeten Datenstrukturen

4.4 Realzeitnachweis RTEDF

Der Nachweis der Realzeitfähigkeit gliedert sich in drei Teile. Im ersten Teil wird gezeigt, dass durch eine online Rechenzeitanforderungsanalyse mit einem *Acceptance Queue* Ansatz ein Überlastzustand, der in schlimmsten Fall zu Deadlineverletzungen führen kann, immer detektiert werden kann. Im zweiten Teil wird ein Nachweisverfahren vorgestellt, dass die resultierende Rechenzeitanforderung des Laufzeitsystems bei Anwendung der λ_i– und ψ_i–Pattern ermöglicht. Es wird bewiesen, dass mit RTEDF immer ein gültiger Schedule existiert, wenn durch Berücksichtigung der *QoS*–Pattern die Rechenzeitanforderung nicht größer als eins ist. Die Gültigkeit des Nachweisverfahrens wird mit Hilfe von bestehenden Aussagen zu EDF bewiesen.

Im dritten Teil wird der Realzeitnachweis des RTEDF Schedulers selber geführt. Hier wird nachgewiesen, dass mit der angewendeten Scheduling Strategie immer ein gültiger Schedule erzeugt werden kann.

4.4.1 Rechenzeitanforderungsanalyse

Für die Analyse, ob die im System befindlichen Transaktionen im *Worst Case* zu einer Überlast führen können, wird die bei dynamischen Schedulingverfahren verwendete Rechenzeitanforderungsfunktion verwendet (Abbildung 4.9). Der Abstand zur Winkelhalbierenden ist die für die neue Transaktion zur Verfügung stehende Rechenzeit in dem betrachteten Intervall. Mit den Parametern WCTET und der absoluten Deadline kann bei linearer Komplexität ($O(n)$) die Rechenzeitanforderung zum Zeitpunkt t berechnet werden.

Lemma 3
Gegeben sei eine Menge von Transaktionsinstanzen $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, die nach absoluter Deadline d_i sortiert sind. Die Laxity L_i jeder Transaktion t_i zum Zeitpunkt t kann wie folgt iterativ berechnet werden

$$L_i \;=\; L_{i-1} + (d_i - d_{i-1}) - c_{i,A_{m,k}}. \tag{4.5}$$

Beweis
Die Laxity einer Transaktion ist definiert als $L_i = d_i - z_i$, mit z_i als Endzeit der Transaktion i. Für die Transaktion mit der kürzesten Deadline ergibt sich also eine Laxity von $L_1 = d_1 - z_1$. Durch die nach absoluter Deadline sortierten Reihenfolge der Transaktionen ist auch die Reihenfolge der Abarbeitung gegeben. Daher folgt für die Transaktion mit der zweit–kürzesten Deadline der Zusammenhang $L_2 = d_2 - z_2$. Dabei bestimmt sich z_2 zu $z_2 = z_1 + c_2$, da die zweite Transaktion sofort mit der Abarbeitung weitergeführt wird sobald die erste beendet wurde.

Somit ergibt sich für $L_2 = d_2 - (z_1 + c_2) = d_2 - d_1 + L_1 - c_2$. Ohne Einschränkung der Allgemeinheit kann mit $z_i = z_{i-1} + c_i$ auf die Aussage geschlossen werden

$$L_i = d_i - z_i = d_i - (f_{i-1} + c_i) = L_{i-1} + (d_i - d_{i-1}) - c_i.$$

□

Abbildung 4.9 zeigt die grafische Darstellung einer Rechenzeitanforderungsfunktion.

Satz 3
Wenn eine mögliche Überlast im Laufzeitsystem auftritt, dann ist mindestens ein Laxitywert aus der Rechenzeitanforderungsanalyse kleiner Null.

Beweis
Indirekt: Alle Laxitywerte einer Rechenzeitanforderungsanalyse zum Zeitpunkt t seien größer Null und das System befindet sich in Überlast. Die Laxity entspricht

$$\begin{aligned} L_i &= I_i - \sum_{j=1}^{i} D_j(I_i) \\ &= D_i - \sum_{j=1}^{i} D_j(D_i) \end{aligned}$$

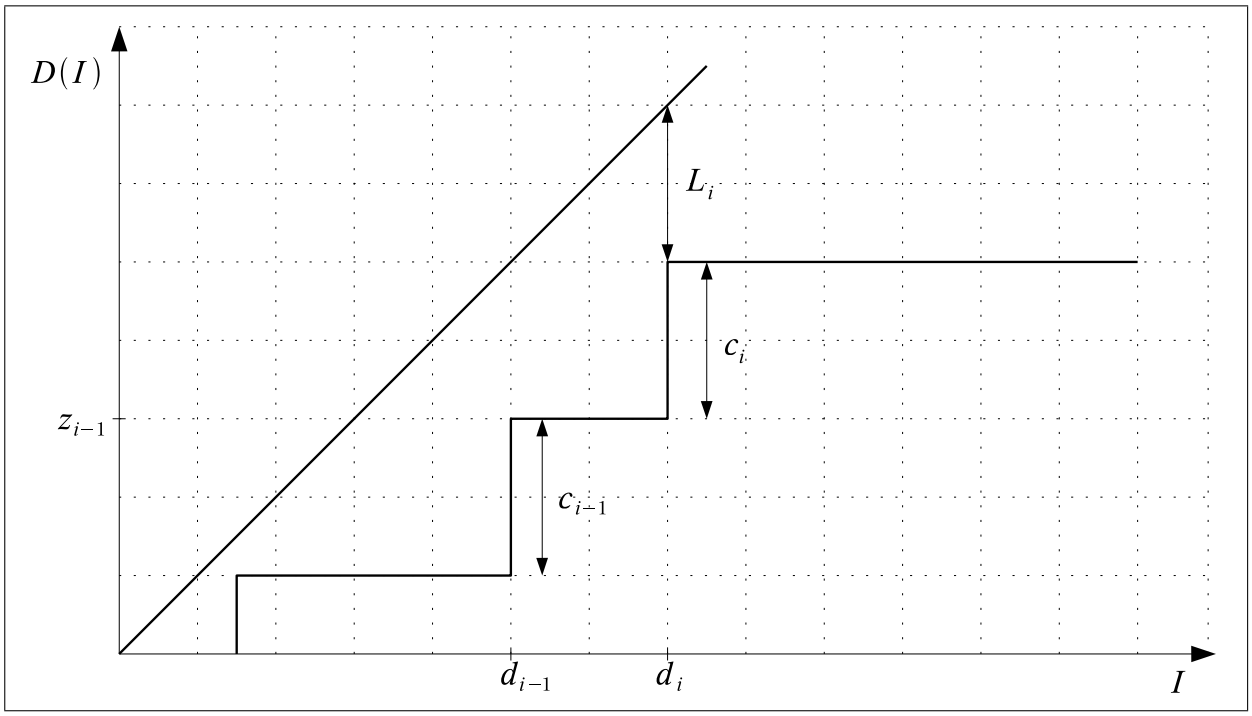


Abbildung 4.9: Rechenzeitanforderungsfunktion

mit

$$D_j(I) = E_{F_i}(I - D_j) \cdot C_i.$$

Wenn aber alle Laxitywerte zum Zeitpunkt t größer Null sind, so bedeutet das, dass

$$L_i = I_i - \sum_{j=1}^{i} D_j(I_i) > 0 \quad \Longrightarrow \quad \frac{\sum_{j=1}^{i} D_j(I)}{I} \leq 1 \qquad \forall i.$$

Der letzte Term entspricht Gleichung 3.19 und liefert für Werte unter 1 die notwendige und hinreichende Bedingung, dass keine Überlast vorliegt. Daraus folgt, dass Satz 3 im Gegensatz zur Annahme wahr ist.

□

4.4.2 Nachweisverfahren RTEDF

Das Nachweisverfahren für RTEDF erlaubt durch Transformation der Ereignisfunktionen und Anwendung der λ– und ψ–Pattern eine offline Analyse der Realzeitfähigkeit eines Transaktionssystems.

In diesem Abschnitt wird zunächst die Korrektheit des Nachweisverfahren nachgewiesen. Dazu werden zwei ausgezeichnete Fälle betrachtet, nämlich die Umschaltung in den Überlastmodus und die dauerhafte Anwendung der spezifizierten Pattern.

4.4.2.1 Umschaltung in den Überlastmodus

Die einzige definierte Aussage, die für einen Nachweis verwendet werden kann, ist die Tatsache, dass die Deadlines aller im System befindlichen Transaktionsinstanzen zu einem definierten Zeitpunkt t_{rzaf} um das Zeitintervall Δ verlängert werden. Diese Deadlineverlängerung kann für einen Nachweis wie eine neue Instantiierung der bereits im System befindlichen Transaktionsinstanzen mit einer neuen Deadline ($D = \Delta$) angesehen werden. Die Verlängerung der Deadline zum Zeitpunkt t_{rzaf} ist in Abbildung 4.10 dargestellt. Die sich ändernden Deadlineintervalle werdem mit I_i' definiert und bestimmen sich zu

$$I_i' = I_i + \Delta_i = t_{ai} + D_i - t_{rzaf} + \Delta_i.$$

Für einen offline Nachweis können nur konstante Parameter verwendet werden. Der Wert von I_i' hängt sowohl vom Ereigniszeitpunkt als auch von dem Analysezeitpunkt ab. Die einzige von der Laufzeit unabhängige Größe ist der Delta Wert. Für den Nachweis wird von einer initialen Transaktionsinstanz mit einer Deadline gleich dem Delta Wert (Δ) ausgegangen.

Wie in Abschnitt 4.3.5 bereits beschrieben, wird vom RTEDF System zur Laufzeit eine Umschaltung in den Überlastmodus durchgeführt. Daher ist es beim Nachweis von Interesse, ob im schlimmsten Fall, d. h. bei dauerhafter Anwendung des λ– Pattern, eine Überlast überhaupt verhindert werden kann. Für die Betrachtung bleiben die *Soft*–Transaktionen zunächst unberücksichtigt. ψ–Pattern stellen einen Spezialfall der λ–Pattern dar, so dass der Nachweis auf diese Pattern identisch angewendet werden kann. Daher werden die ψ–Pattern zunächst nicht betrachtet.

Für die Analyse wird eine modifizierte Ereignisfunktion $E_{F_{qos}i}$ benötigt, die das Worst Case Szenario bei ständiger Anwendung der λ–Pattern beschreibt. Dazu wird der Überlastmodus in drei Phasen aufgeteilt.

Die erste Phase beschreibt das einmalige Auftreten einer Instanz mit einer Deadline, die dem spezifizerten Delta Wert entspricht (Δ). Die Phase wird als γ–Phase bezeichnet.

Die zweite Phase, in der maximal v_i konsekutive Transaktionsinstanzen mit verlängerter Deadline aktiv sein können, wird als δ–Phase bezeichnet. Die darauf folgende σ–Phase bezeichnet den Abschnitt, in dem die konsekutiven und harten Transaktionen f_i vorkommen können.

Diese Trennung ist insofern nötig, als dass man bei Anwendung eines Patterns einen Eingriff in den aktuellen Laufzeitsystemzustand vornimmt. Dadurch erhält man drei Ereignisströme, die zwar in einem wohldefinierten Abhängigkeitsverhältnis zueinander stehen, jedoch aufgrund unterschiedlicher Deadlines separat behandelt werden müssen.

In Abbildung 4.12 ist der Transformationsprozess für eine Ereignisfunktion aufgezeigt.

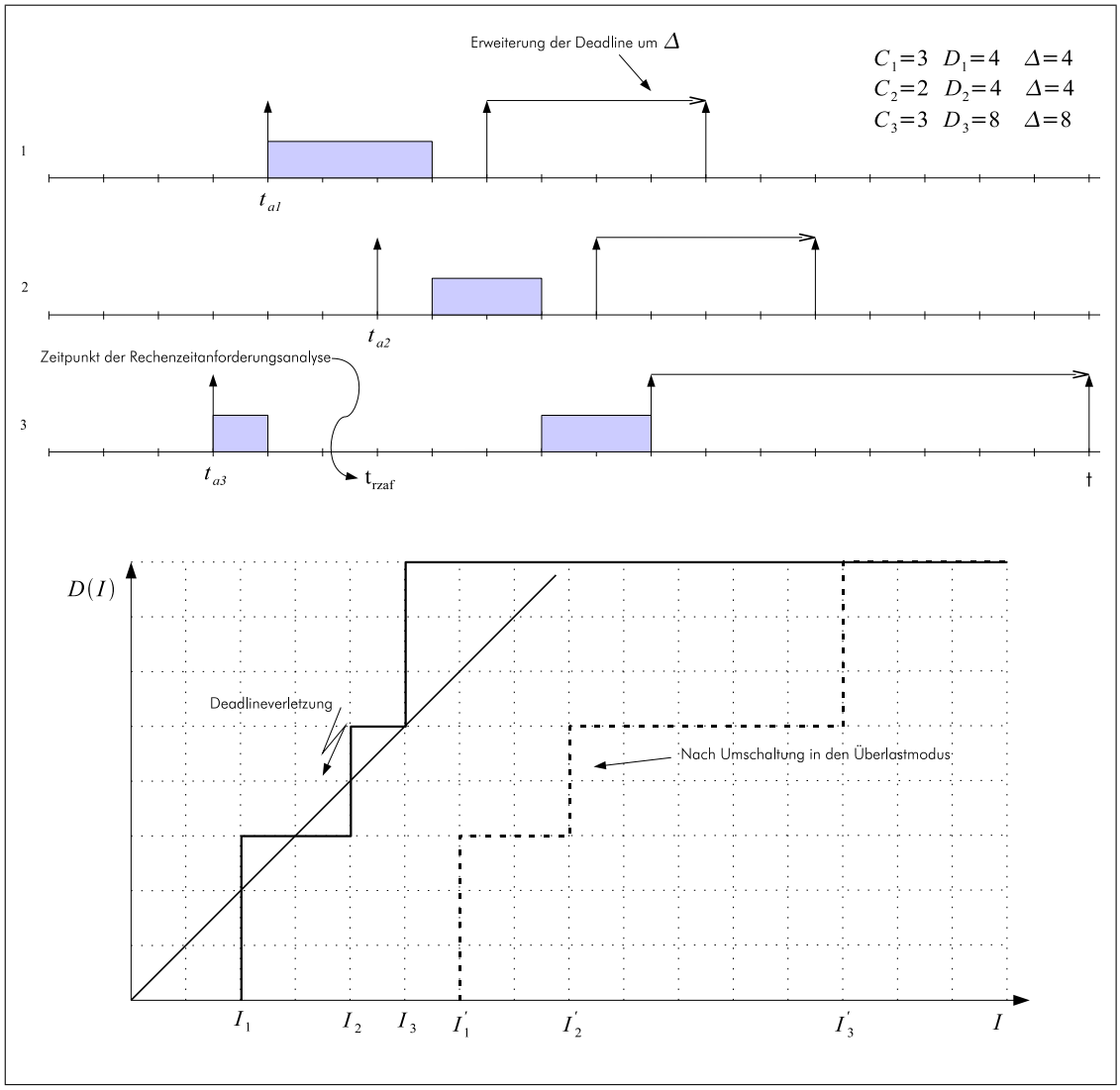


Abbildung 4.10: Drei Transaktionsinstanzen mit einer Umschaltung in den Überlastmodus zum Zeitpunkt t_{rzaf}

Man geht von einer periodischen Anwendung eines λ–Patterns aus. Dabei ergibt sich eine periodische Aneinanderreihung von δ– und σ–Phasen. Für die Transformation in den für die Analyse relevanten Intervallbereich sind nur die Ereignisse pro Intervall und ihre möglichen Periodizitäten von Interesse.

γ–Phase

Ein Ereignis, dass zum Analysezeitpunkt auf seine verlängerte Deadline erweitert wird, befindet sich in der γ–Phase. Die resultierende Ereignisfunktion $E_{F_{\gamma i}}$ ist in Abbildung 4.12 dargestellt.

δ–Phase

Wenn für eine Transaktionsinstanz ein Parameter $v > 1$ definiert ist, so schließt sich an die γ–

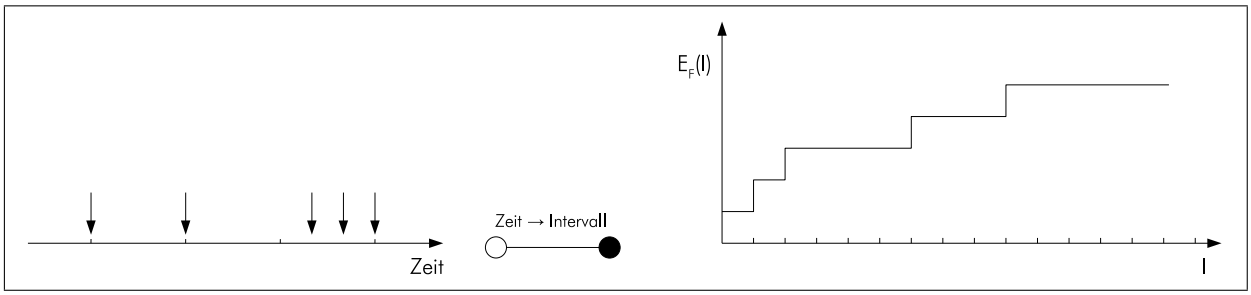


Abbildung 4.11: Zeit — Intervall Transformation

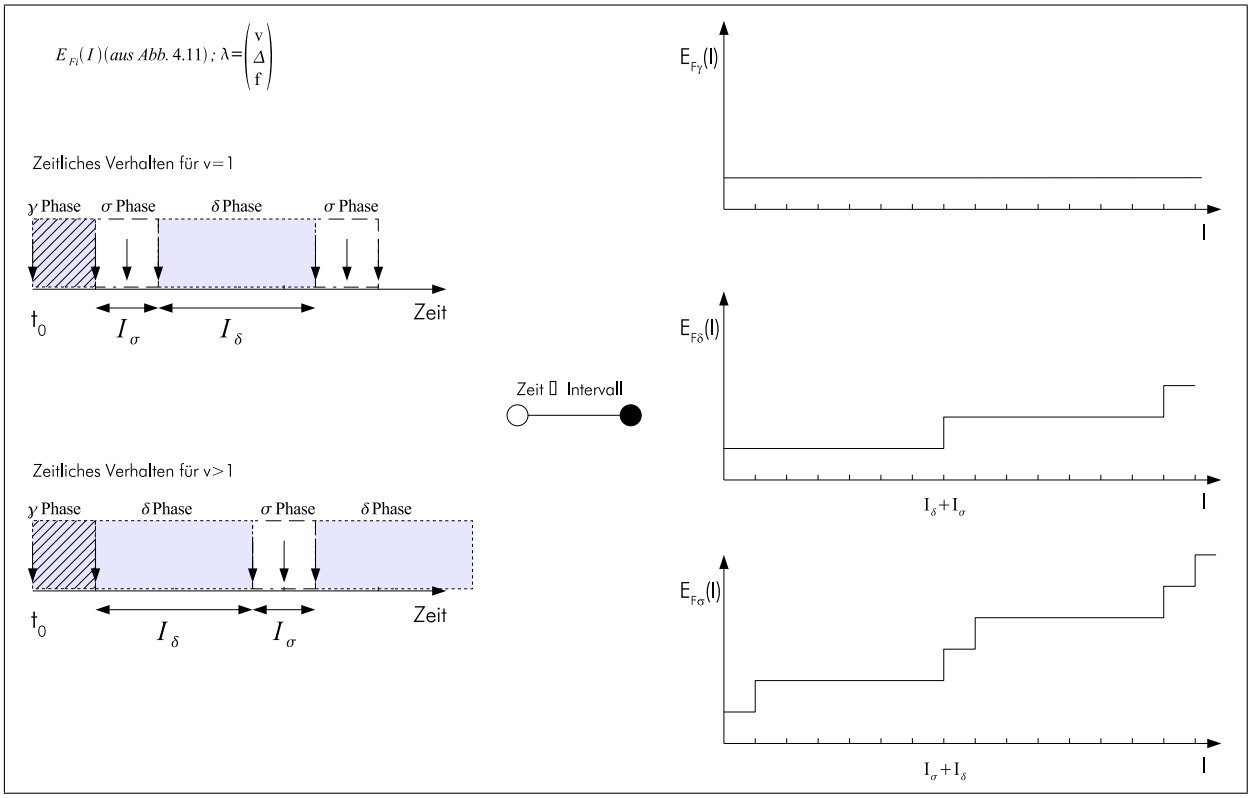


Abbildung 4.12: Anwendung des λ–Pattern

Phase die δ–Phase an. Ereignisse, die auftreten, werden auf ihre verlängerte Deadline erweitert. Im ungünstigsten Fall kann sich diese Phase periodisch mit der Periode $p_k = I_\sigma + I_\delta$ wiederholen. I_x bezeichnet hier das Intervall, in dem sich das System gerade befindet. Die resultierende Ereignisfunktion $E_{F_\delta i}$ ist in Abbildung 4.12 dargestellt.

σ–Phase

Wenn für eine Transaktionsinstanz ein Parameter $v = 1$ definiert ist, so schließt sich an die γ–Phase unmittelbar die σ–Phase an. Für eine *Worst Case* Betrachtung wird angenommen, dass ein Ereignis bereits zum Zeitpunkt $t_0 + \delta$ auftritt und dort auch mit der in der Ereignisfunktion definierten Ereignisdichte für Intervalle der Breite $I = I_{Fi}(f_i) + d_i$ berücksichtigt wird. Die σ–

Phase ist ebenfalls periodisch mit einer Periode $p_k = I_\sigma + I_\delta$. Die resultierende Ereignisfunktion $E_{F_\sigma i}$ ist in Abbildung 4.12 dargestellt.

4.4.2.2 Abhängigkeitsmatrix und Ereignisfunktionen

Die Abhängigkeitsmatrix EAM beschreibt die Abhängigkeiten zwischen den Ereignisfunktionen $E_{F_\gamma i}$, $E_{F_\delta i}$ und $E_{F_\sigma i}$ für den Nachweis. Das minimale Intervall, in dem n Ereignisse auftreten können, ist über die Inverse $I_{F_i}(n)$ (Gleichung 3.7) der Ereignisfunktion definiert

$$EAM = \begin{pmatrix} e_{\sigma\sigma} & e_{\sigma\delta} & e_{\sigma\gamma} \\ e_{\delta\sigma} & e_{\delta\delta} & e_{\delta\gamma} \\ e_{\gamma\sigma} & e_{\gamma\delta} & e_{\gamma\gamma} \end{pmatrix} \tag{4.6}$$

mit

$$e_{\sigma\sigma} = \begin{cases} I_{F_i}(2) + v \cdot (D + \Delta) & : \ f = 1 \\ I_{F_i}(2) & : \ f > 1 \end{cases} \tag{4.7}$$

$$e_{\sigma\delta} = I_{F_i}(f + 1) \tag{4.8}$$

$$e_{\delta\sigma} = v \cdot (D + \Delta) \tag{4.9}$$

$$e_{\delta\delta} = \begin{cases} D + \Delta + I_{F_i}(f + 1) & : \ v = 1 \\ D + \Delta & : \ v > 1. \end{cases} \tag{4.10}$$

$$\tag{4.11}$$

Die Parameter $e_{\sigma\gamma}$, $e_{\gamma\gamma}$ und $e_{\delta\gamma}$ sind nicht definiert, da es nur eine γ–Phase gibt und es keine γ–Phase nach einer δ– oder σ–Phase geben kann[4)]. Für die restlichen Koeffizienten der Matrix ergibt sich

$$e_{\gamma\sigma} = \begin{cases} \Delta & : \ v = 1 \\ \Delta + (v - 1) \cdot (D + \Delta) & : \ v > 1 \end{cases} \tag{4.12}$$

$$e_{\gamma\delta} = \begin{cases} \Delta & : \ v > 1 \\ \Delta + e_{\sigma\delta} & : \ v = 1. \end{cases} \tag{4.13}$$

$$\tag{4.14}$$

Definition 18
Die transformierte Ereignisfunktion $E_{F_{qos}i}$ wird durch ein vier–Tupel definiert

$$E_{F_{qos}i} = \{E_{F_\gamma i}, E_{F_\delta i}, E_{F_\sigma i}, EAM_i\}. \tag{4.15}$$

[4)] Diese Annahme gilt für die Worst Case Betrachtung.

Die Gesamtrechenzeitanforderungsfunktion für n Transaktionen $\Gamma_i \in \boldsymbol{Q_{qos}}$ lautet

$$\begin{aligned} D_{qos}(I) = \sum_{i=1}^{n} & (E_{F_\gamma i}(I - (\Delta)) \cdot C_i) \\ & + (E_{F_\delta i}(I - (D + \Delta) - e_{\sigma\delta} - e_{\gamma\delta}) \cdot C_i) \\ & + (E_{F_\sigma i}(I - D_i - e_{\gamma\sigma}) \cdot C_i). \end{aligned} \tag{4.16}$$

Analog dazu kann die transformierte Ereignisfunktion und die Gesamtrechenzeitanforderungsfunktion für n Transaktionen $\Gamma_i \in \boldsymbol{Q_{soft}}$ angegeben werden:

$$E_{F_{soft}i} = \{E_{F_\gamma i}, E_{F_\delta i}, EAM_i\} \tag{4.17}$$

und

$$D_{soft}(I) = \sum_{i=1}^{n} (E_{F_\delta i}(I - (D + \Delta)) \cdot C_i). \tag{4.18}$$

4.4.3 Realzeitnachweis RTEDF Scheduling

Der Eingriff in das Laufzeitsystem zur Laufzeit stellt eine Änderung des bis zu diesem Zeitpunkt angewandten Schedulings dar. Für diesen Eingriff ist nachzuweisen, dass unter allen Umständen mit dem RTEDF Scheduling eine Überlast bei gleichzeitiger Einhaltung der *QoS*–Pattern verhindert werden kann.

Der Nachweis beschränkt sich auf zwei ausgezeichnete Fälle, nämlich die Umschaltung in den Überlastmodus nach einer Intervallverzögerung I_x und die Wiedereinschaltung des Überlastmodus bei Transaktionen, die ihr Pattern noch nicht durch firme Instanzen vervollständigt haben.

Gleichung 4.16 besagt, dass bei einer stetigen Anwendung der λ– und σ–Pattern alle Echzeitbedingungen eingehalten werden. Das Laufzeitsystem soll die Pattern nur in einer Überlast Situation anwenden und nach einer Überlast wieder in den Normalbetrieb übergehen. Dieses *Mode Switching* wird vom RTEDF Scheduler nach einer fest vorgegebenen Strategie durchgeführt, für die ein Realzeitnachweis nötig ist. Es muss sichergestellt sein, dass durch das Zu– oder Abschalten der Pattern immer ein gültiger Schedule entsteht.

4.4.3.1 Aktivierung des Überlastmodus

Unter Aktivierung der λ– und ψ–Pattern wird ein Vorgang im RTEDF Scheduler verstanden, bei dem alle nachfolgenden Transaktionsinstanzen gemäß der vordefinierten Pattern in das Laufzeitsystem eingefügt werden.

Satz 4
Wenn $D_{qos}(I) \leq I \, \forall i$ gilt und ab dem Zeitpunkt der Überlastumschaltung t_L auf alle ankommenden Ereignisse und bereits im System befindlichen Transaktionen die spezifizierten λ–Pattern angewendet werden, dann existiert immer ein gültiger Schedule.

Beweis
Zum Umschaltzeitpunkt wird auf alle im System befindlichen Transaktionen der Delta Wert zur Deadline hinzuaddiert. Im Nachweis wird davon ausgegangen, dass alle Transaktionen gleichzeitig mit dem Deadlinewert Δ starten ($\sum_i E_{F_\gamma i}(I - \Delta)) \cdot C_i(I)$, Term aus Gleichung 4.16). Gibt es nun einen Zeitpunkt $t_{miss} > t_L$, an dem eine Transaktionsinstanz τ_i ihre Deadline verletzt, so bedeutet dies, dass trotz Anwendung der Delta Deadlines der Anteil der γ–Phase größer als $\sum_i E_{F_\gamma i}(I - (\Delta)) \cdot C_i(I)$ ist. Das wiederum würde bedeuten, dass mindestens eine der im System befindlichen Instanzen die primäre Deadline bereits überschritten hätte, was durch Anwendung der Rechenzeitanforderungsanalyse bei jeder neu eintreffenden Instanz und nach jedem Verzögerungsintervall I_x ein Widerspruch ist.

□

4.4.3.2 Umschalten in den Normalmodus

Das Laufzeitsystem befindet sich genau dann nicht mehr im Überlastmodus, wenn eine Idle Time detektiert worden ist. Davon unberührt bleiben die firmen Transaktionen in den Pattern noch bestehen. Es werden wieder alle Transaktionen zugelassen, solange nicht einer der folgenden Fälle auftritt:

(a) Neu eintreffende Transaktionen, die keine firmen Transaktionsinstanzen mehr erfüllen müssen, führen erneut zu einer Überlast.

(b) Neu eintreffende Transaktionen, die noch firme Transaktionsinstanzen erfüllen müssen, führen zu einem Umschalten in den Überlastmodus.

Diese Umschaltung in den Überlastmodus bei noch ausstehenden firmen Instanzen ist dadurch motiviert, dass sobald sich firme Instanzen im System befinden der Nachweis für das verzögerte Umschalten in Überlastmodus nicht mehr gilt.

4.4.4 Realzeitnachweis parallele RTEDF Einheit

Die in dieser Arbeit betrachtete Auslagerung des RTEDF Algorithmus auf eine externe Ausführungseinheit, erfordert ebenfalls eine Realzeitbetrachtung der parallelisierten Systemarchitektur. Die maximalen Rechenzeiten, die bei einer Analyse zu berücksichtigen sind, werden basierend auf dem in Abbildung 4.4 dargestellten Strukturbild in Tabelle 4.4 definiert. Die Konvention zur Namensgebung spiegelt die Zuordnung der Zeiten wider: $C_{a\_b_c}$ bezeichnet die maximale Rechenzeit, die für die Routine oder Funktion c im Modul $a = \{a : RTEDF \vee TEDF\}$ bei paralleler oder sequentieller Realisierung $b = \{b : par \vee seq\}$ benötigt wird.

4.4.4.1 Sequentielle Abarbeitung

Wir betrachten zunächst den Fall, dass der RTEDF Algorithmus auf derselben Recheneinheit wie das TEDF Laufzeitsystem ausgeführt wird. Das bedeutet insbesondere, dass dem Laufzeitsystem Rechenzeit für die Bearbeitung der Transaktionen entnommen wird. Die Entnahme ist

$C_{tedf_{event}}(n)$	Rechenzeit, die für die Übermittlung eines *QoS–*/*Soft–*Events zur RTEDF Einheit notwendig ist.
$C_{rtedf\_\{par/seq\}_{sort}}(n)$	Maximale Zeit, die für die sortierte Ausgabe des Heaps benötigt wird.
$C_{rtedf\_\{par/seq\}_{rtanalysis}}(n)$	Maximale Zeit, die für eine Rechenzeitanforderungsanalyse benötigt wird.
$C_{rtedf\_\{par/seq\}_{main}}(n)$	Rechenzeit, die im Worst Case für den RTEDF Algorithmus notwendig ist.
$C_{tedf_{insert}}(n)$	Maximale Zeit, die nach einem Interrupt des RTEDF Systems die Transaktionsverwaltung zum Einfügen einer neuen Instanz in die entsprechende Mailbox benötigt.
$C_{tedf_{extendall}}(n)$	Maximale Zeit, die die Transaktionsverwaltung benötigt, alle im Laufzeitsytem befindlichen *QoS–* und *Soft–* Transaktionsinstanzen gegebenenfalls zu Delta–Instanzen zu transformieren.

Tabelle 4.4: Maximale Rechenzeiten in der RTEDF Systemarchitektur

der Berücksichtigung von Interruptservicezeiten gleichzusetzen, da diese in der Kernelebene ablaufen und damit nicht zur TEDF Schedulingebene zählen.

Definition 19
Bezeichnet n die maximale Anzahl aller im Laufzeitsystem gleichzeitig vorkommenden Transaktionsinstanzen, so ergibt sich die Worst Case Ausführungszeit für das RTEDF Verfahren in Kombination mit dem TEDF System zu

$$\begin{aligned} C_{seq}(n) = C_{tedf_{event}}(n) &+ C_{rtedf\_seq_{main}}(n) \\ &+ C_{rtedf\_seq_{sort}}(n) + C_{rtedf\_seq_{rtanalysis}}(n) \\ &+ max(C_{tedf_{insert}}(n), C_{tedf_{extendall}}(n)). \end{aligned} \tag{4.19}$$

Beispiel 4.3
In Abbildung 4.13 ist ein Beispiel für den Realzeitnachweis mit zwei periodischen Transaktionen dargestellt. Die Gerade $D(I)$ wird um den zeitlichen Betrag verringert, der für neu eintreffende Rechenzeitanforderungen benötigt wird. Da es sich um zwei Transaktionen handelt, können im schlimmsten Fall jeder Zeit zwei Anforderungen innerhalb eines Intervalls von vier Zeiteinheiten auftreten. Dieses wird durch das zweifache Subtrahieren von $C_{seq}(2)$ berücksichtigt. Für die Realzeitanalyse bedeutet dies, dass sämtliche Rechenzeitanforderungen nun unterhalb der neuen Funktion $D'(I)$ liegen müssen.

4.4.4.2 Parallele Abarbeitung

Bei der parallelen Abarbeitung findet die Bearbeitung der im RTEDF System ablaufenden Routinen auf einer parallelen Einheit statt. Die maximal dafür benötigte Rechenzeit definiert sich

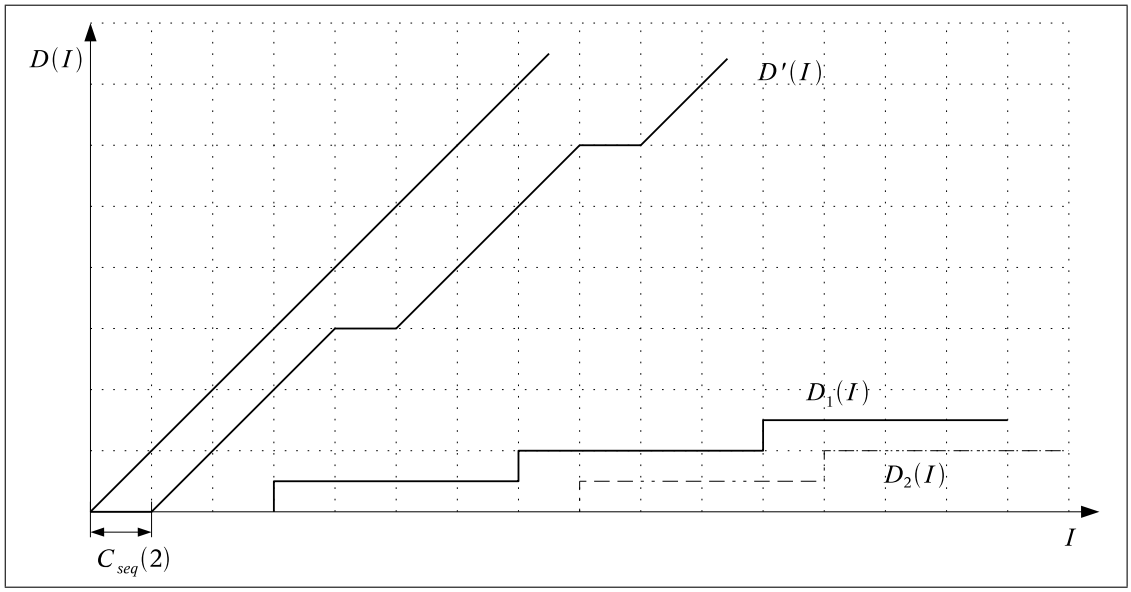


Abbildung 4.13: Berücksichtigung der RTEDF Laufzeiten bei sequentieller Realisierung

zu

$$C_{par_{rtedf}}(n) = C_{rtedf\_par_{main}}(n) + C_{rtedf\_par_{sort}}(n) + C_{rtedf\_par_{rtanalysis}}(n). \tag{4.20}$$

Die Rechenzeitanforderung der Routinen, die für die Transaktionsverwaltung benötigt werden, definiert sich zu

$$C_{par_{tedf}}(n) = C_{tedf_{event}}(n) + max(C_{tedf_{insert}}(n), C_{tedf_{extendall}}(n)). \tag{4.21}$$

Der RTEDF Algorithmus auf der parallelen Einheit wird, wenn er einmal initiiert wurde, nicht durch neu eintreffende Ereignisse unterbrochen. Das relevante Worst Case Szenario ist dadurch charakterisiert, dass infinitesimal kurz nachdem ein neues Ereignis eingetroffen ist und das zu einer Aktivierung der RTEDF Einheit geführt hat, weitere Ereignisse eintreffen. Die Zugangskontrolle genau dieser Ereignisse benötigt nun genau $2 \cdot C_{par_{rtedf}}$, da sie erst nach $t = C_{par_{rtedf}}$ eingelesen werden können. Anschließend wird noch die Zeit zum Eingreifen in das TEDF Laufzeitsystem $C_{par_{tedf}}$ berücksichtigt.

Beispiel 4.4
In Abbildung 4.14 ist die für den Realzeitnachweis resultierende Funktion $D''(I)$ dargestellt. Geht man davon aus, dass die parallele Einheit weit weniger Rechenleistung zur Verfügung stellt, so zeigt sich, dass der Großteil der Geradenabsenkung auf den Anteil auf der parallelen Einheit abfällt. Für den Realzeitnachweis darf jedoch die auf dem TEDF System entstehende Rechenzeitanforderung $C_{par_{tedf}}$ nicht vernachlässigt werden. Durch die Parallelisierung ergibt sich eine Unabhängigkeit von der Ereignisstromfunktion der Transaktionen. $D''(I)$ hängt sowohl von der maximalen als auch von der minimalen Rechenzeitanforderung des RTEDF Systems ab. Eine weitere Größe, die es beim Realzeitnachweis zu berücksichtigen gilt, ist das

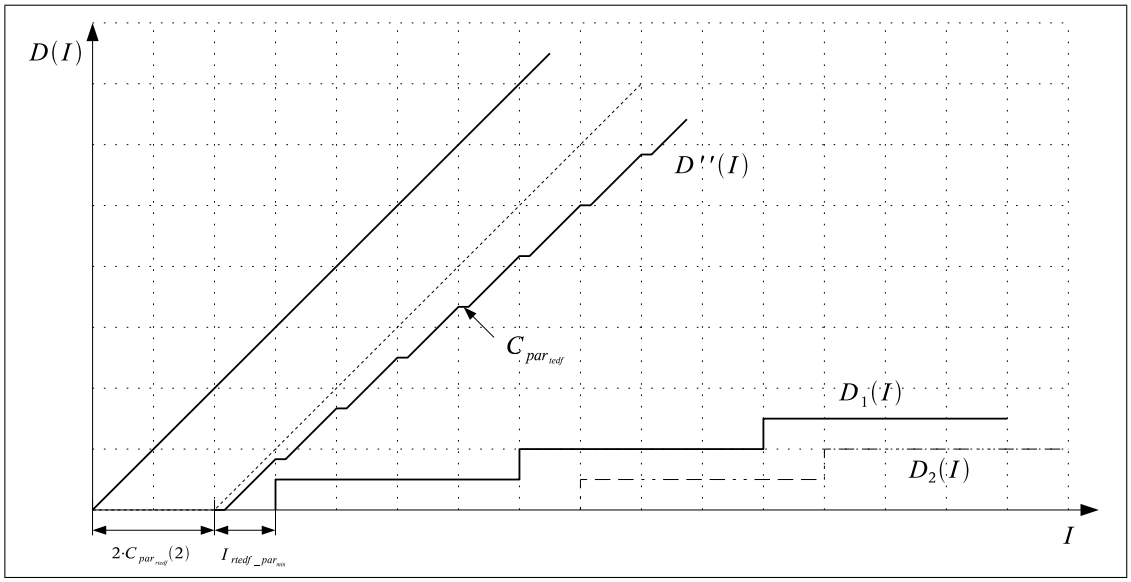


Abbildung 4.14: Berücksichtigung der RTEDF Laufzeiten bei parallelisierter Realisierung

Intervall $I_{rtedf\_par_{min}}$. Dieses Intervall muss derart gewählt werden, dass bei einer Summation aller Rechenzeitanforderungen, diese weiterhin unterhalb der Funktion $D''(I)$ bleiben.

5 Systemanalyse RTEDF

Für die Bewertung der vorgestellten Überlastbehandlung wurde ein System Simulator entwickelt, der es erlaubt, verschiedene Transaktionsszenarien zu simulieren. Die im Folgenden vorgestellten Simulationsergebnisse erlauben eine qualitative Einordnung des RTEDF Algorithmus in Abhängigkeit von den verschiedenen Transaktionsparametern. Die Simulation basiert auf einem idealisierten TEDF Scheduling und der idealisierten Überlastbehandlung RTEDF. Die Idealisierung bezieht sich auf die Vernachlässigung der Ausführungszeiten sämtlicher Betriebssystemfunktionen des TEDF Schedulings, der Kommunikationsstrukturen zwischen TEDF und RTEDF und der Laufzeiten des RTEDF Algorithmus selbst. Die zunächst vernachlässigten Zeiten werden später in der implementierungstechnischen Betrachtung in Kapitel 6 berücksichtigt.

5.1 Qualitätsmaß

Zur Klassifikation der Transaktionsinstanzen wird folgende Einordnung definiert:

Delta: Transaktionen, die entweder mit einer *Delta*–Deadline gestartet wurden oder während der Abarbeitung eine *Delta*–Deadline erhalten haben, werden als *Delta*–Transaktionen betrachtet. Die Menge aller *Delta*–Transaktionsinstanzen wird mit $\mathbf{N}_{delta_i}$ definiert.

Firm: Transaktionen, die in einer σ–Phase abgearbeitet werden, werden als *Firm*–Transaktionen bezeichnet. Die Menge aller *Firm*–Transaktionsinstanzen wird mit $\mathbf{N}_{firm_i}$ definiert.

QoS: Transaktionsinstanzen, die abgearbeitet werden, ohne dass eine Deadlineverlängerung stattfindet, zählen zu den *QoS*–Transaktionsinstanzen. Die Menge der *QoS*–Transaktionsinstanzen einer Transaktion pro Simulation wird mit $\mathbf{N}_{qos_i}$ definiert.

Skip: Transaktionsinstanzen, die während einer δ–Phase eintreffen, werden verworfen und werden daher als Skip–Transaktionsinstanzen definiert. Die Menge der Skip–Transaktionsinstanzen einer Transaktion pro Simulation wird mit $\mathbf{N}_{skip_i}$ definiert.

Die Menge aller Transaktionsinstanzen einer Transaktion Γ_i ergibt sich aus

$$\mathbf{N}_i = \mathbf{N}_{delta_i} \cup \mathbf{N}_{firm_i} \cup \mathbf{N}_{qos_i} \cup \mathbf{N}_{skip_i}$$

und für das Transaktionsset gilt dann

$$\mathbf{N} = \bigcup_{i=1}^{n} \mathbf{N}_i.$$

Für die Analyse der aus den System Simulationen gewonnen Daten, werden im Folgenden Metriken definiert, die sowohl eine Klassifikation einzelner Transaktionen als auch des gesamten Transaktionssets erlauben.

Beispiel 5.1
In Abbildung 5.1 und 5.2 sind für eine Transaktion die ermittelten Werte aus einer Simulation dargestellt. Die beiden Histogramme stellen die Anzahl der konsekutiven Firm–/QoS– *bzw.* Delta–*Transaktionsinstanzen dar, während Abbildung 5.2 zum einen den relativen Anteil der* Firm–/QoS– *Transaktionsinstanzen aufzeigt und zum anderen den Anteil der Transaktionsinstanzen darstellt, der unabhängig von seiner Deadline die primäre kürzere Deadline einhalten konnte. Für die Beispieltransaktion gilt ein λ–Pattern von*

$$\lambda = \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix}.$$

Der Wert $Q_{\text{kfirm}} = 5.086$ bezeichnet den Erwartungwert für konsekutive Firm– *und* QoS–*Transaktionsinstanzen. Der minimale Erwartungswert wird gemäß λ–Pattern bei $Q_{\text{kfirm}} = 1$ liegen. Ein Vergleich von unterschiedlichen Simulationsläufen für ganze Transaktionssets ist durch die normierte Größe $Q^{\sim}_{\text{kfirm}}$ in Gleichung 5.3 möglich.*

Der Wert $Q_{\text{kdelta}} = 1$ gibt an, wie viele konsekutive Transaktionsinstanzen als Delta–*Instanzen abgearbeitet wurden. In diesem Beispiel sind maximal zwei konsekutive* Delta–*Instanzen erlaubt. Auch dieser Wert wird für einen Vergleich in Gleichung 5.6 zu $Q^{\sim}_{\text{kdelta}}$ normiert.*

Der Wert $Q_{\text{rel}} = 0.78$ bedeutet, dass 78% aller Transaktionsinstanzen dieser Transaktion als Firm– *oder* QoS– *Transaktionsinstanzen abgearbeitet wurden. Um auch hier eine Vergleichbarkeit zu gewährleisten wird in Gleichung 5.8 ein Mittelwert über alle Transaktionen einer Simulation definiert.*

Der Wert Q_{meet} gibt an, wieviele Transaktionsinstanzen aller Instanzen trotz Deadlineverlängerung ihre primäre Deadline einhalten konnten. Für eine Gesamteinordnung der Ergebnisse wird mit $Q^{\sim}_{\text{meet}}$ in Gleichung 5.10 ein Mittelwert über alle Transaktionen einer Simulation definiert.

In den folgenden Abschnitten werden die oben genannten Größen formal definiert.

Für die qualitative Bewertung macht es keinen Unterschied, ob eine primäre Deadline in Form einer *Firm–* oder *QoS–*Instanz eingehalten wurde. In den folgenden Analysen werden daher beide Instanzarten zusammengefasst.

Definition 20 (Q_{kfirm})
Q_{kfirm} einer Transaktion i bezeichnet den gleichgewichteten Mittelwert für das Auftreten von konsekutiven normalen und Firm– *Transaktionsinstanzen. p_j bezeichnet dabei die Auftrittswahrscheinlichkeit für eine Folge von j konsekutiven normalen/*Firm–*Transaktionsinstanzen. Die untere Grenze ist durch den Parameter f_i des λ–Pattern vorgegeben, während die obere*

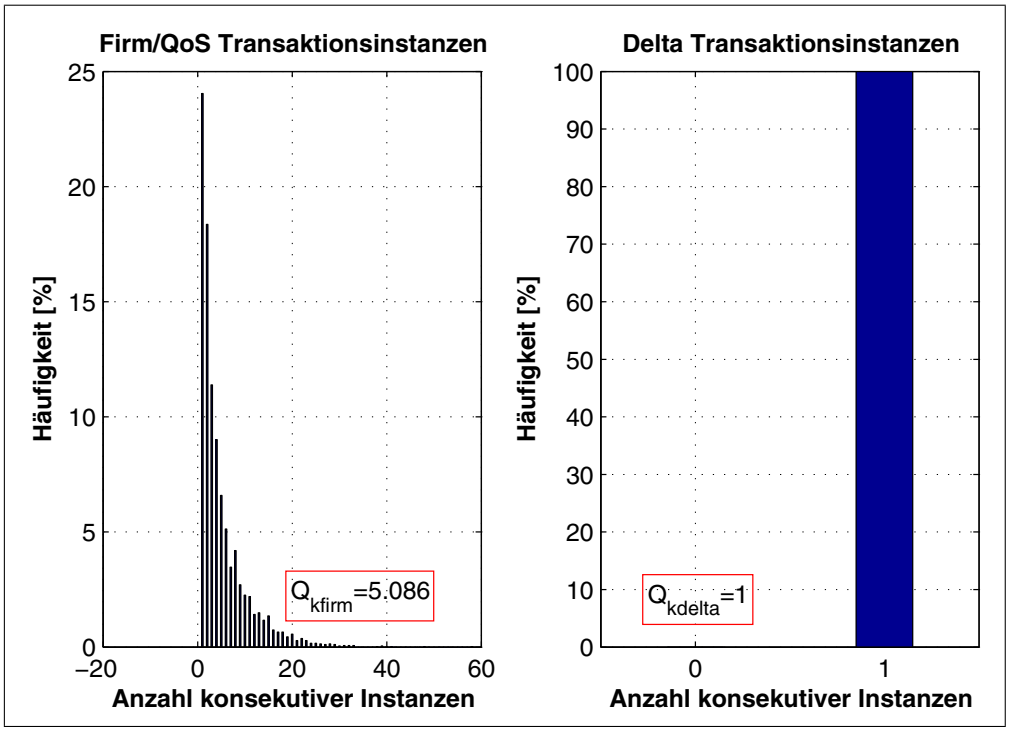


Abbildung 5.1: Simulationsergebnisse für Q_{kfirm} und Q_{kdelta} für eine Transaktion

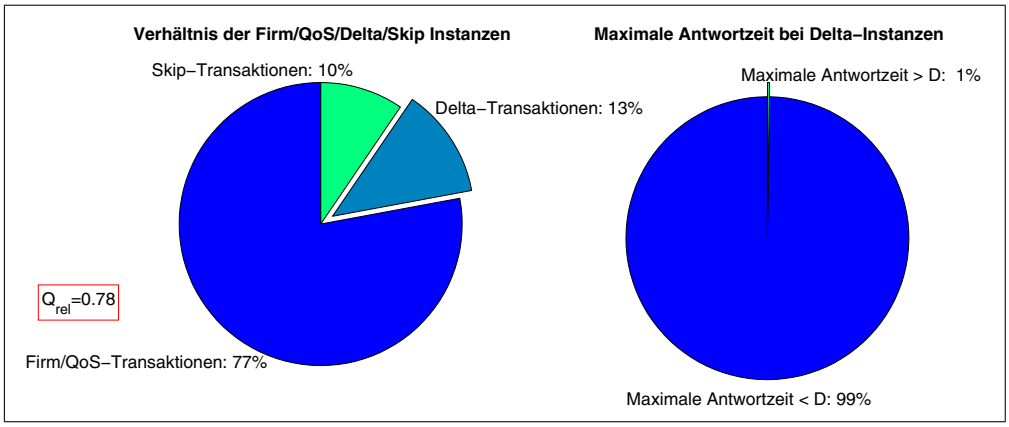


Abbildung 5.2: Simulationsergebnisse für Q_{krel} und Q_{meet} für eine Transaktion

*Grenze N durch die Folge mit der maximalen Anzahl an normalen/*Firm*−Transaktionsinstanzen definiert ist:*

$$Q_{kfirm_i} = \sum_{j=f_i}^{|\mathbf{N}_i|} p_j \cdot j \qquad \textit{mit} \qquad min(f_i) = 1. \tag{5.1}$$

Für einen Vergleich von Transaktionen mit unterschiedlichen λ−Pattern wird die Größe Q_{kfirm} mit dem Parameter f_i normiert:

$$Q^{\sim}_{kfirm_i} = \frac{1}{f_i} \cdot (Q_{kfirm_i} - f_i). \tag{5.2}$$

*Ein Wert von $Q^{\sim}_{kfirm} = 0$ gibt an, dass alle aufgetretenen konsekutiven normalen/*Firm*–Transaktionsinstanzfolgen immer genau f_i lang waren. Das ist genau dann der Fall, wenn ein System ständig in Überlast betrieben wird und sich σ– und δ–Phasen abwechseln. Je größer dieser Wert ist, umso häufiger treten Transaktionsinstanzfolgen mit mehr als f_i konsekutiven Instanzen auf.*

Ein Qualitätsmaß für ein Transaktionsset wird durch die Mittelung aller $Q^{\sim}_{kfirm_i}$ Werte erreicht:

$$Q^{\sim}_{kfirm} = \frac{1}{n} \cdot \sum_{i=1}^{n} Q^{\sim}_{kfirm_i}. \tag{5.3}$$

Definition 21 (Q_{kdelta})
Q_{kdelta} einer Transaktion i bezeichnet den gewichteten Mittelwert für das Auftreten von konsekutiven Delta*–Transaktionsinstanzen. p_j bezeichnet dabei die Auftrittswahrscheinlichkeit für eine Folge von j konsekutiven* Delta*–Transaktionsinstanzen. Die obere Grenze wird durch den Parameter v_i des λ–Pattern vorgegeben, der angibt, wie viele Transaktionsinstanzen konsekutiv zu* Delta*–Transaktionsinstanzen gemacht werden dürfen:*

$$Q_{kdelta_i} = \sum_{j=1}^{v_i} p_j \ \cdot \ j. \tag{5.4}$$

Für einen Vergleich von Transaktionen mit unterschiedlichen λ–Pattern wird die Größe Q_{kdelta} mit dem Parameter v_i normiert:

$$Q^{\sim}_{kdelta_i} = \frac{1}{v_i} \cdot Q_{kdelta_i}. \tag{5.5}$$

Ein Wert von $Q^{\sim}_{kdelta} = 1$ gibt an, dass alle aufgetretenen konsekutiven Delta*–Transaktionsinstanzfolgen immer genau v_i lang waren. Das ist genau dann der Fall, wenn ein System ständig in Überlast betrieben wird und sich σ– und δ–Phasen abwechseln. Je kleiner dieser Wert ist, umso häufiger treten Transaktionsinstanzfolgen mit weniger als v_i konsekutiven Instanzen auf.*

Ein Qualitätsmaß für ein Transaktionsset wird durch die Mittelung aller $Q^{\sim}_{kdelta_i}$ Werte erreicht:

$$Q^{\sim}_{kdelta} = \frac{1}{n} \cdot \sum_{i=1}^{n} Q^{\sim}_{kdelta_i}. \tag{5.6}$$

Definition 22 (Q_{rel})
Mit Q_{rel_i} wird der relative Anteil an nicht erweiterten Transaktionsinstanzen im Verhältnis zur gesamten Anzahl aller aufgetretenen Instanzen bezeichnet:

$$Q_{rel_i} = \frac{|\mathbf{N}_{firm_i}| + |\mathbf{N}_{qos_i}|}{|\mathbf{N}_i|}. \tag{5.7}$$

Ein Qualitätsmaß für ein Transaktionsset wird durch die gleichgewichtete Mittelung aller Q_{rel_i} Werte erreicht:

$$Q_{rel}^{\sim} = \frac{1}{n} \cdot \sum_{i=1}^{n} Q_{rel_i}. \tag{5.8}$$

Im Überlastmodus werden die Deadlines der *QoS*–Transaktionen dergestalt verlängert, dass für das Gesamtsystem keine Überlast mehr auftreten kann. Die Rechenzeitanforderung wird dadurch in jedem Intervall immer unter 100% liegen. Für die Analyse ist der Anteil an *Delta*–Instanzen von Interesse, die trotz verlängerter Deadline ihre firme primäre Deadline einhalten können.

Definition 23 (Q_{meet})
Q_{meet} bezeichnet den relativen Anteil an Transaktionsinstanzen, die unterhalb der firmen primären Deadline geblieben sind:

$$Q_{meet_i} = \frac{|\mathbf{N}_{meet_i}| + |\mathbf{N}_{firm_i}| + |\mathbf{N}_{qos_i}|}{|\mathbf{N_i}|} \tag{5.9}$$

mit

$$\mathbf{N}_{meet_i} = \{\tau_{i,j} : \tau_{i,j} \in \mathbf{N}_{delta_i} \ \wedge \ response\_time(\tau_{i,j}) < d_i - \Delta_i\}.$$

Für die Bewertung eines Laufzeitsystems werden die einzelnen Anteile gleichgewichtet gemittelt:

$$Q_{meet}^{\sim} = \frac{1}{n} \sum_{i=1}^{n} Q_{meet_i}. \tag{5.10}$$

Weitere Faktoren zur Beurteilung der Effizienz des Verfahrens sind die mittlere angeforderte Rechenzeit $\rho_{\text{demand}}^{\sim}$ und die tatsächlich zur Verfügung gestellte Rechenzeit $\rho_{\text{rtedf}}^{\sim}$.

Definition 24 ($\rho_{\text{demand}}^{\sim}$)
$\rho_{demand}^{\sim}$ definiert die mittlere angeforderte Rechenzeit pro Simulationszeitintervall I

$$\rho_{demand}^{\sim} = \frac{1}{I} \sum_{i=1}^{n} E'_{F_i} \cdot C'_i. \tag{5.11}$$

E'_{F_i} bezeichnet die je nach Simulationsszenario abgeänderte Ereignisfunktion, C'_i die reell zur Laufzeit angeforderte Rechenzeit.

Definition 25 ($\rho_{\text{rtedf}}^{\sim}$)
$\rho_{rtedf}^{\sim}$ definiert die mittlere zur Verfügung gestellte Rechenzeit pro Simulationszeitintervall I

$$\rho_{rtedf}^{\sim} = \frac{1}{I} \sum_{i=1}^{n} (|\mathbf{N}_{firm_i}| + |\mathbf{N}_{qos_i}| + |\mathbf{N}_{delta_i}|) \cdot C'_i. \tag{5.12}$$

Definition 26 ($\rho^{\sim}_{\text{diff}}$)
$\rho^{\sim}_{\text{rtedf}}$ ist definiert als die Differenz zwischen $\rho^{\sim}_{\text{demand}}$ und $\rho^{\sim}_{\text{rtedf}}$.

5.2 System Simulation

Für die generische Einordnung der Eigenschaften des RTEDF Schedulings wird ein synthetisches Transaktionsset bestehend aus drei *QoS*– Transaktionen verwendet. Die Daten der drei Transaktionen sind in Tabelle 5.1 dargestellt.

Transaktionsparameter	**Transaktion**		
	T1	T2	T3
Zeiteinheit[1)]	ms	ms	ms
Ereignisstromtupel	$E_{T_1} = \begin{pmatrix} 6 \\ 6 \end{pmatrix}$	$E_{T_2} = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$	$E_{T_3} = \begin{pmatrix} 8 \\ 8 \end{pmatrix}$
λ–Pattern	$\begin{pmatrix} 2 \\ 6 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 8 \\ 1 \end{pmatrix}$
Best Case Transaktionsausführungszeit[2)]	1.9	1.0	1.9
Worst Case Transaktionsausführungszeit	3.0	2.0	3.0
Relative Deadline	6	4	8

Tabelle 5.1: Transaktionsparameter für ein synthetisches Beispiel

Ein Realzeitnachweis dieses Transaktionssets zeigt, dass ohne die Anwendung der λ–Pattern eine maximale Rechenzeitanforderung von 137.5% auftreten könnte[3)] (Abb. 5.3 (a)). Der Realzeitnachweis für das RTEDF System mit Anwendung der λ–Pattern zeigt, dass das System im Überlastfall innerhalb der spezifizierten *QoS* Grenzen realzeitfähig bleiben wird (Abb. 5.3 (b)).

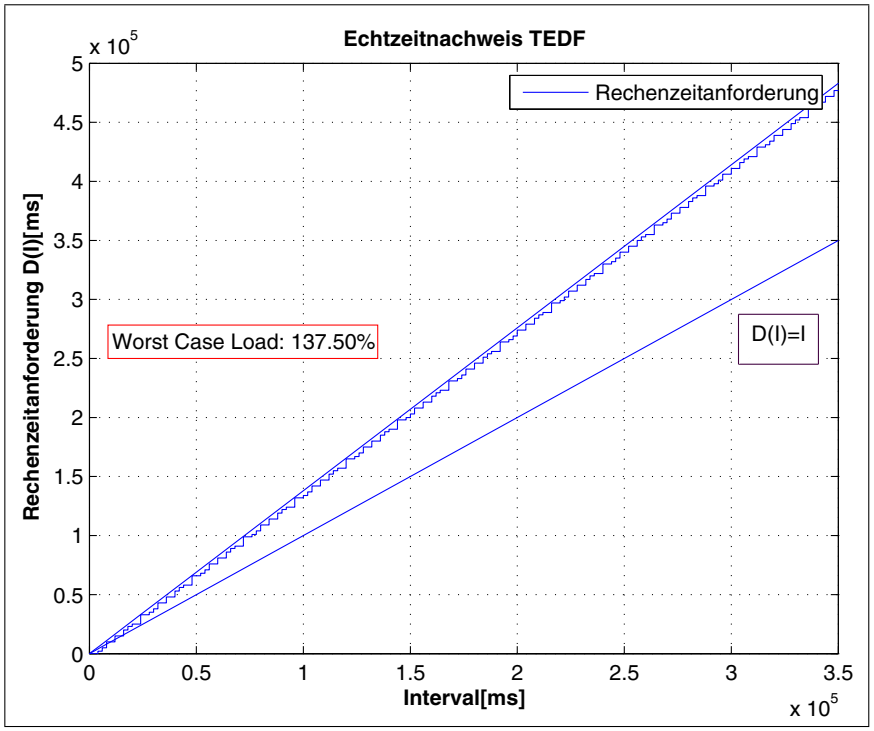
In den folgenden Abschnitten wird basierend auf dem Transaktionsset von Tabelle 5.1 anhand verschiedener Systemszenarien gezeigt, wie sich der variierende Rechenzeitbedarf auf die Erfüllung der *QoS*–Parameter einzelner Transaktionen auswirkt.

Die relevanten Größen, die betrachtet werden, sind $Q^{\sim}_{\text{kfirm}}$, $Q^{\sim}_{\text{rel}}$, $\rho^{\sim}_{\text{demand}}$, $\rho^{\sim}_{\text{rtedf}}$ und $Q^{\sim}_{\text{meet}}$.

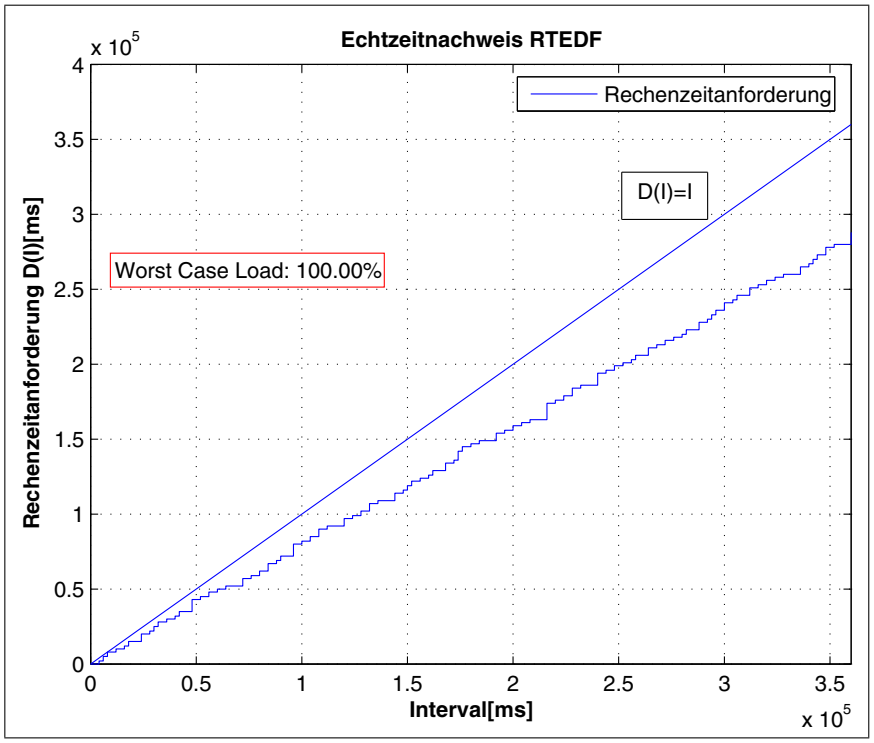
5.3 Worst Case Simulation

Beim RTEDF Scheduling wird durch die Überlastbehandlung zur Laufzeit in das Laufzeitsystem eingegriffen, um entsprechend der Spezifikation die Deadlines zu verlängern und damit

[1)] Die angegebene Zeiteinheit gilt sowohl für die Ereignisintervalle als auch für die Ausführungszeiten.
[2)] Die Zeiten sind in diesem Beispiel derart gewählt, dass damit sowohl das Verhalten des RTEDF Algorithmus in Überlast als auch in Unterlastsituationen analysiert werden kann.
[3)] Ein Rechnerkern müsste 37.5% mehr Rechenleistung liefern, um den geforderten Rechenzeitbedarf für den Worst Case zu decken.



(a) ohne Pattern



(b) mit Pattern

Abbildung 5.3: Realzeitnachweis für das Beispieltransaktionsset

Realzeitverletzungen zu verhindern. In dieser Worst Case Simulation werden die in der Spezifikation beschriebenen periodischen Transaktionen betrachtet, die im minimalen Ereignissabstand auftreten.

Diese Simulation liefert Basiswerte für einen praxisrelevanten Vergleich, da sie den Worst Case Fall für die periodische Aktivierung des Systems darstellt. Für die Transaktionen wird das einfachste Modell angenommen. Eine Transaktion besteht aus einem Startereignis und einer darauffolgenden Aktivierung eines Threads. Es gilt die WCTET für alle Transaktionen. In der folgenden Tabelle 5.2 sind die durch den Worst Case Nachweis ermittelten Qualitätswerte dargestellt.

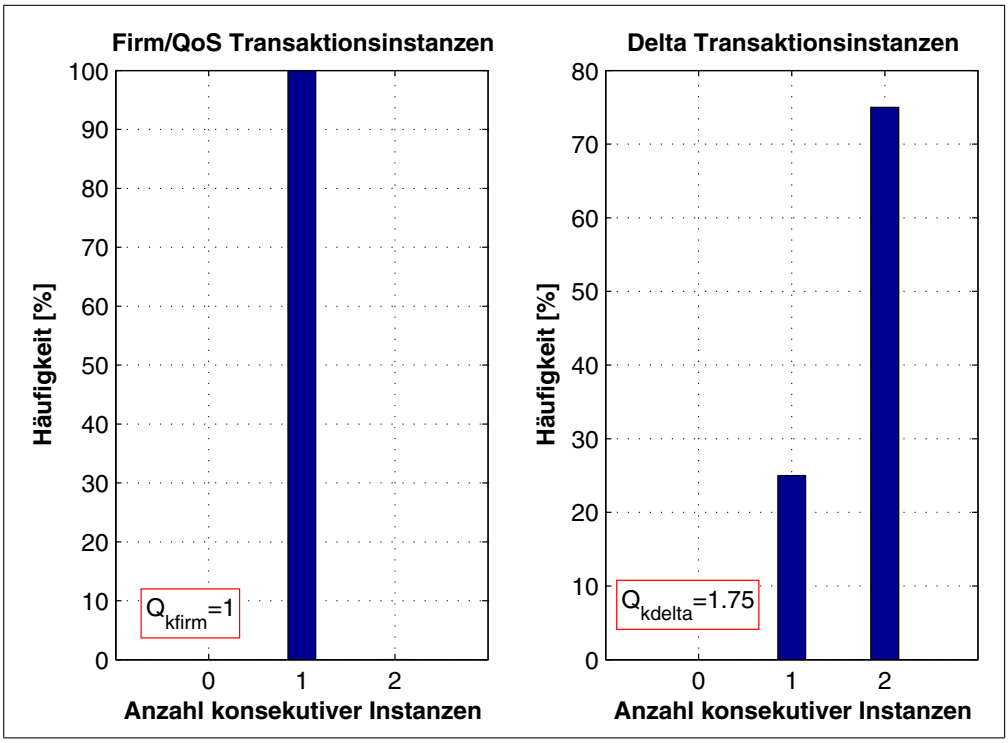


Abbildung 5.4: Gleichgewichteter normierter Mittelwert konsekutiver *Firm–/QoS–*Instanzen $Q^{\sim}_{\text{kfirm}}$

Simulationsszenario	$Q^{\sim}_{\text{kfirm}}$	$Q^{\sim}_{\text{rel}}$	$\rho^{\sim}_{\text{demand}}$	$\rho^{\sim}_{\text{rtedf}}$	$Q^{\sim}_{\text{meet}}$
Worst Case Szenario: Periodischer Worst Case Ereignisstrom mit Worst Case Transaktionsausführungszeiten	16.67%	29.63%	137.5%	88.89%	32.14%

Tabelle 5.2: Ergebnisse der Worst Case Simulation

Die Ergebnisse der Simulation (Tabelle 5.2) dienen als Referenz für die Vergleiche der folgenden Simulationsszenarien.

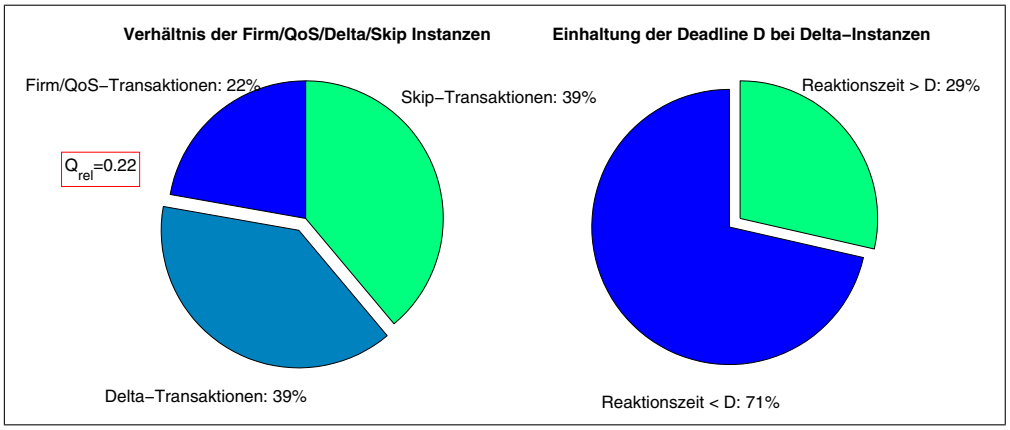


Abbildung 5.5: Simulationsergebnisse für Q_{krel} und Q_{meet} im Worst Case (für eine Transaktion)

5.4 Variation der Ausführungszeiten

Für Applikationscode kann in der Praxis fast nie eine konstante Rechenzeit bestimmt werden. Rechenzeitschwankungen treten unter anderem durch Cacheverdrängungsszenarien, TLB Umladevorgänge und natürlich aufgrund von diversitären Pfaden innerhalb der Software auf. Desweiteren sind bei einer Systemauslegung auch die Blockierungszeiten zu berücksichtigen, die zur Laufzeit nicht immer im vollen Umfang auftreten. Auch dieses „Nichteintreten“ von Rechenzeiten führt zur Laufzeit zu einem geringeren aber variierenden Rechenzeitbedarf. Vereinfacht kann für jede Threadaktivität angenommen werden, dass ein kleinster (BCTET) und ein größter Wert (WCTET) existiert und ermittelt werden kann.

In dieser Arbeit wird für die Modellierung der Ausführungszeitverteilung in der Simulation die Gumbel–Verteilung [86][87] verwendet. Die Gumbel–Verteilung wird zur Modellierung von Extremwertstatistiken verwendet. Sie erlaubt die Darstellung einer Zufallsgröße, dessen Werte um einen ausgezeichneten Arbeitspunkt verteilt sind (Extremwert). Gleichzeitig können durch Abflachen der Dichteverteilung weiter entfernte Werte stärker berücksichtigt werden[4)]. Die Wahrscheinlichkeitsverteilungsfunktion $F(x)$ ist in Gleichung 5.13 angegeben:

Definition 27 (Gumbel Verteilungsfunktion)

$$F(x) = exp\left\{-exp\left[-\left(\frac{x-\alpha}{\beta}\right)\right]\right\}. \tag{5.13}$$

Beispiel 5.2
In Abbildung 5.6 sind mehrere Wahrscheinlichkeitsdichteverteilungen dargestellt. Auf der Abszisse ist die Ausführungszeit von BCTET(800) bis WCTET(900) aufgetragen. Der Parameter $\alpha = 810$ stellt einen Arbeitspunkt der Transaktion dar. Mit dem Parameter β wird der Arbeitspunkt etwas abgeflacht und berücksichtigt daher die Extremwerte stärker. Die Zufallsvariable ist jedoch immer auf ein vorgegebenes Intervall begrenzt.

[4)] Für eine detaillierte Modellierung der Rechenzeitverteilungen müsste für jeden Einflussfaktor eine eigene Verteilungsfunktion berücksichtigt werden.

Wie die entsprechenden Parameter[5)] für eine realistische Modellierung der Ausführungszeiten gewählt werden müssen, hängt stark vom Verhalten der Software ab. Für die Bewertung der RTEDF Behandlung wurden daher mehrere Simulationen mit unterschiedlichen Parametersätzen durchgeführt.

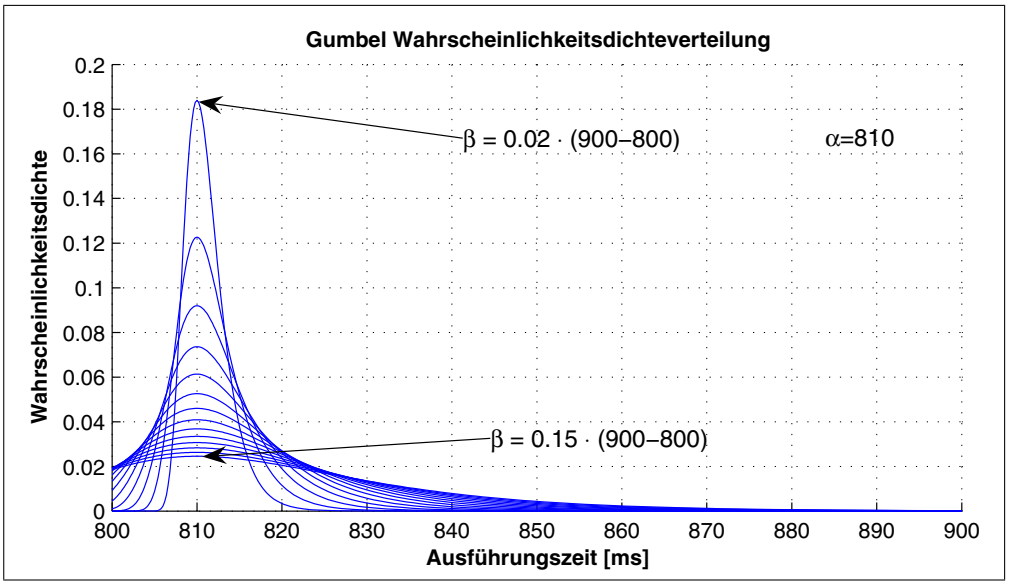


Abbildung 5.6: Gumbel Wahrscheinlichkeitsdichteverteilung

Für die Parametrierung der Gumbel Verteilung werden zwei Größen verwendet.

Gumbel Parameter a Der Parameter **a** gibt an, in welchem Bereich zwischen WCTET und BCTET der Arbeitspunkt liegen soll. Der Parameter α bestimmt sich dann zu

$$\alpha = BCTET + a \cdot (WCTET - BCTET). \tag{5.14}$$

Der verwendete Wertebereich für a liegt zwischen $[0.1 \ldots 0.9]$.

Gumbel Parameter b Der Parameter **b** gibt an, wie stark die Verteilung gestreckt werden soll. Der Parameter β bestimmt sich dann zu

$$\beta = (0.013 \cdot b + 0.02) \cdot (WCTET - BCTET). \tag{5.15}$$

Der verwendete Wertebereich[6)] für b liegt zwischen $[0 \ldots 10]$.

5.4.1 Simulationsergebnisse

Für die Variation der Ausführungszeiten wurde eine Simulation für das in Tabelle 5.1 definierte Transaktionset mit einem festen Gumbel Parameter ($b = 9$) erstellt.

[5)] Die wahrscheinlichkeitstheoretische Betrachtung der Ausführungszeit wird im Forschungsgebiet der WCET Analyse von Realzeitsystemen behandelt und ist nicht Bestandteil dieser Arbeit.

[6)] Die Parameter in Gleichung 5.15 wurden heuristisch ermittelt.

In Abbildung 5.7 ist der zum Gumbel Paramter a resultierende Simulationsverlauf der konsekutiven *Firm–/QoS–* Transaktionsinstanzen $Q^{\sim}_{\text{kfirm}}$ dargestellt. Die Simulation zeigt, dass erst bei einem primär angeforderten Rechenzeitbedarf von 90% die ersten Umschaltungen in den Überlastmodus notwendig werden. Bis zu diesem durchschnittlichen Rechenzeitbedarf zeigt das System ein optimales TEDF Verhalten.

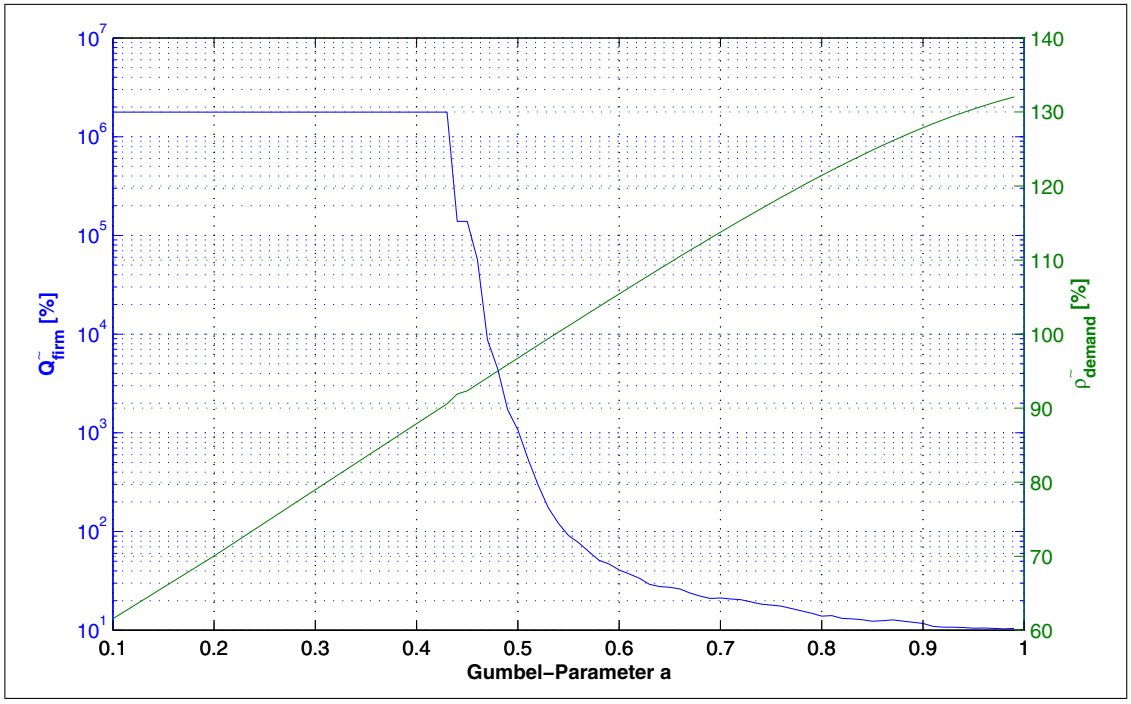


Abbildung 5.7: Gleichgewichteter normierter Mittelwert konsekutiver *Firm–/QoS–*Instanzen $Q^{\sim}_{\text{kfirm}}$ bei Variation der Ausführungszeiten

Das Verhältnis $Q^{\sim}_{\text{rel}}$ von *Firm–/QoS–*Instanzen zu Skip und *Delta–*Instanzen ist in Abbildung 5.8 dargestellt. Auch in diesen Simulationsergebnissen zeigt sich, dass eine Umschaltung erst bei einem durchschnittlich angeforderten Rechenzeitbedarf von über 90% stattfindet.

Diese Aussage ist auch in den anderen Metriken (Abb. 5.9 und 5.10) sichtbar.

Wird eine mögliche Überlast vom RTEDF Algorithmus detektiert, so wird in den Überlastmodus geschaltet. Das führt auch dazu, dass neue Ereignisse verworfen werden müssen. Dadurch kann es auch vorkommen, dass aufgrund der aktuellen Lastsituation bei einer Transaktionsinstanz mit einer verlängerten Deadline die primäre Deadline trotzdem noch eingehalten werden kann. Dieses Vorgehen zeigt sich in der Metrik für den Anteil der *Delta–*Transaktionen $Q^{\sim}_{\text{meet}}$ in Abbildung 5.11.

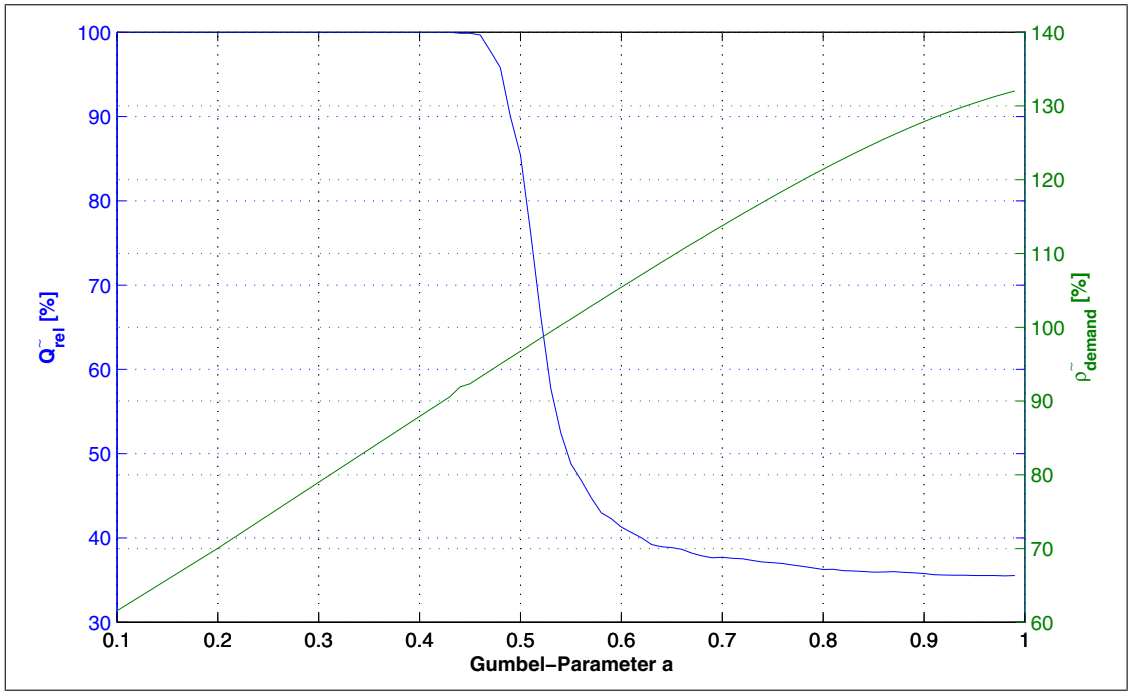


Abbildung 5.8: Gleichgewichteter relativer Anteil an *Firm–/QoS–*Instanzen $Q^{\sim}_{\text{rel}}$ bei Variation der Ausführungszeiten

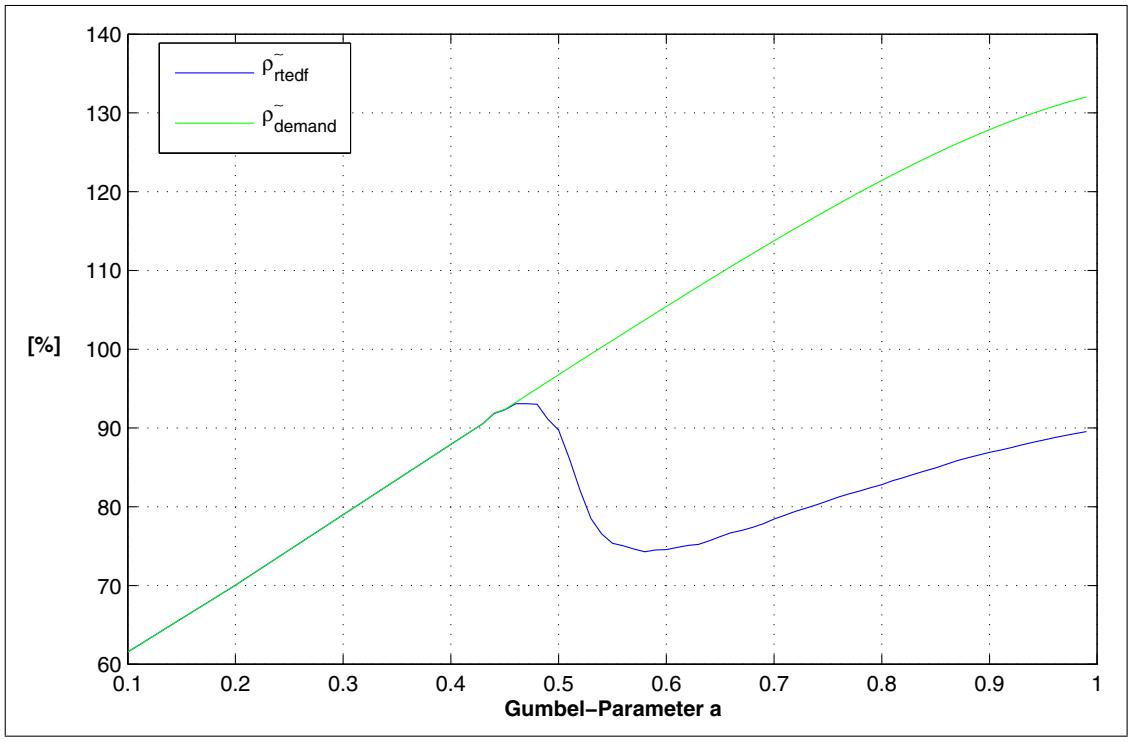


Abbildung 5.9: $\rho^{\sim}_{\text{rtedf}}$ bei Variation der Ausführungszeiten

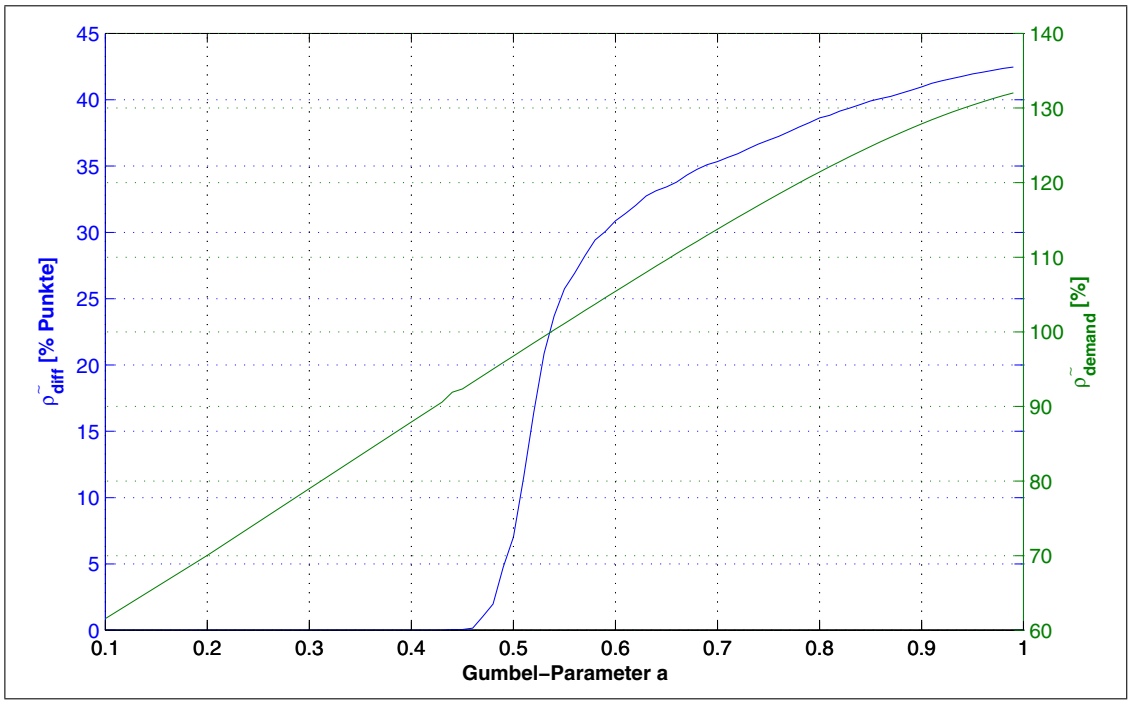


Abbildung 5.10: $\rho^{\sim}_{\text{diff}}$ bei Variation der Ausführungszeiten

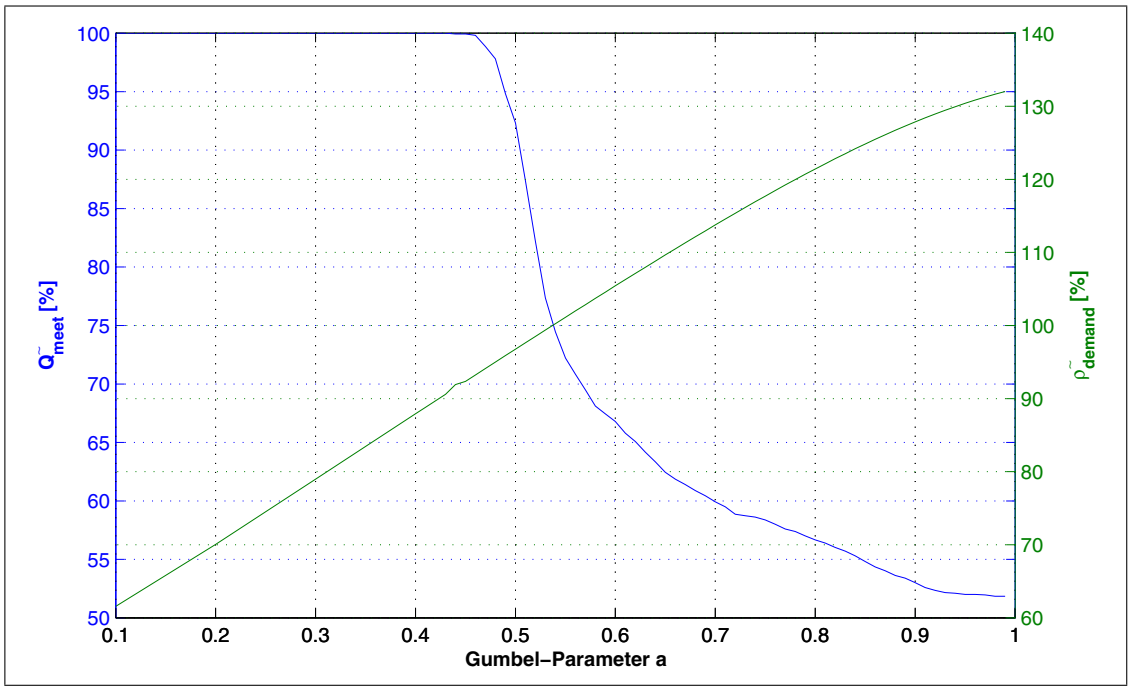


Abbildung 5.11: $Q^{\sim}_{\text{meet}}$ bei Variation der Ausführungszeiten

5.5 Variation der Ereignisströme

Variierende Ereigniszeitpunkte stellen eine Quelle für frei werdende Rechenzeitanforderungen dar. Im Beispieltransaktionsset sind alle Transaktionen der Einfachheit halber mit periodischen Aktivierungszeitpunkten angenommen worden. Für die Betrachtung variierender Ereigniszeitpunkte gehen wir für die Ereignisse von einer Poisson Eigenschaft aus. Die Zeiten zwischen zwei Poisson Ereignissen sind exponential verteilt. Daher wird die Variation der Periodendauern mit einer Exponentialverteilung modelliert. Die Verteilungsfunktion lautet:

Definition 28 (Exponentialverteilungsfunktion)

$$F(x) = \begin{cases} 0 & \textit{für} \quad x < 0 \\ 1 - e^{-\lambda x} & \textit{für} \quad x \geq 0. \end{cases} \tag{5.16}$$

Die Grenzen für die Ereignisabstände sind für die Simulation heuristisch zwischen dem minimalen Ereignisabstand t_{min} und $2 \cdot t_{min}$ definiert. Der Erwartungswert μ der Exponentialverteilung wird als prozentualer Anteil des minimalen Ereignisabstandes definiert:

$$\mu = k \cdot t_{min} \qquad \text{mit} \qquad k =]0.0 \ldots 0.9].$$

Der Ereignisabstand ergibt sich dann aus der Summe des aktuellen Ereignisabstandes und dem Wert der Zufallsvariable.

Die Ergebnisse aus den Simulationen für die definierten Metriken sind in den Abbildungen 5.12, 5.13, 5.14, 5.15 und 5.16 dargestellt.

5.5.1 Simulationsergebnisse

Die Simulationsergebnisse zeigen für dieses Beispiel ein Umschalten in den Überlastmodus bereits bei einem durchschnittlich angeforderten Rechenzeitbedarf von 70%. Das liegt daran, dass die Anzahl der Transaktionen sehr gering ist. Der Rechenzeitbedarf von 137.5% teilt sich dabei auf nur drei Transaktionen auf, so dass bei einem gleichzeitigen Auftreten von zwei Transaktionen bereits eine Überlastbehandlung notwendig wird.

Wird eine mögliche Überlast vom RTEDF Algorithmus detektiert, so wird in den Überlastmodus geschaltet. Das führt auch dazu, dass neue Ereignisse verworfen werden müssen. Dadurch kann es auch vorkommen, dass aufgrund der aktuellen Lastsituation bei einer Transaktionsinstanz mit einer verlängerten Deadline die primäre Deadline trotzdem noch eingehalten werden kann. Dieses Vorgehen zeigt sich in der Metrik für den relativen Anteil der Transaktionen $Q^{\sim}_{\text{meet}}$ (Abb. 5.16), die ihre primäre Deadline einhalten konnten.

In Simulationen mit einer größeren Anzahl an Transaktionen ergeben sich für eine reine Variation der Ereignisströme ähnliche Ergebnisse wie aus Abschnitt 5.4.

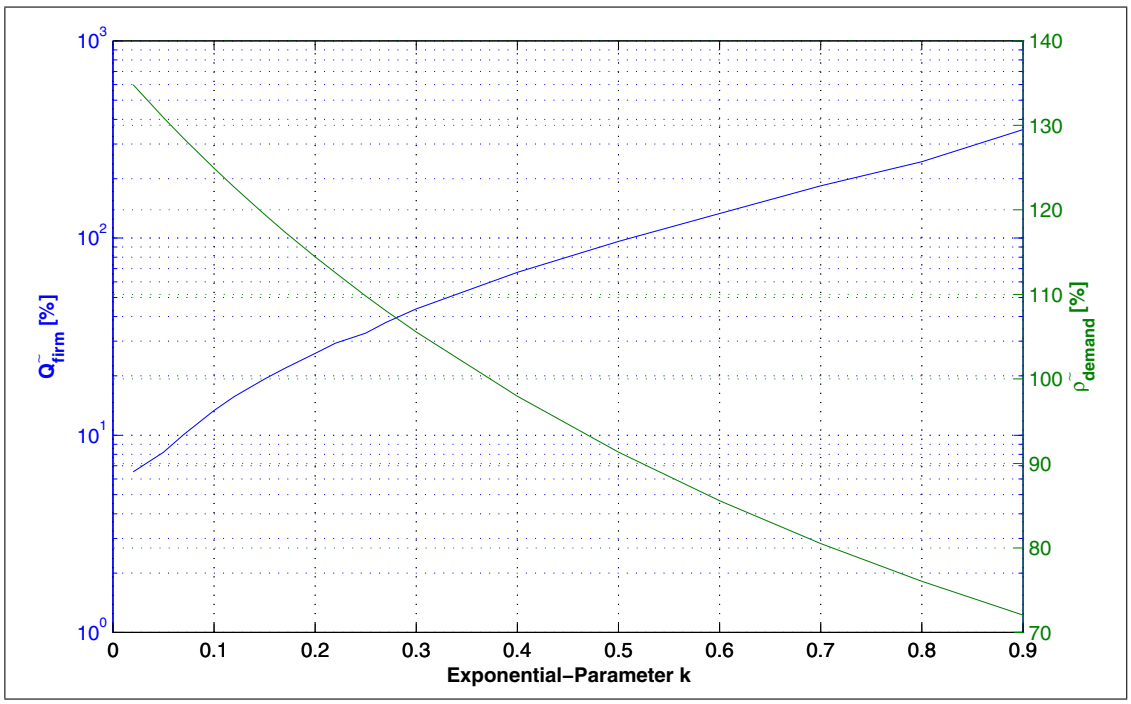


Abbildung 5.12: $Q^{\sim}_{firm}$ bei Variation der Ereignisströme

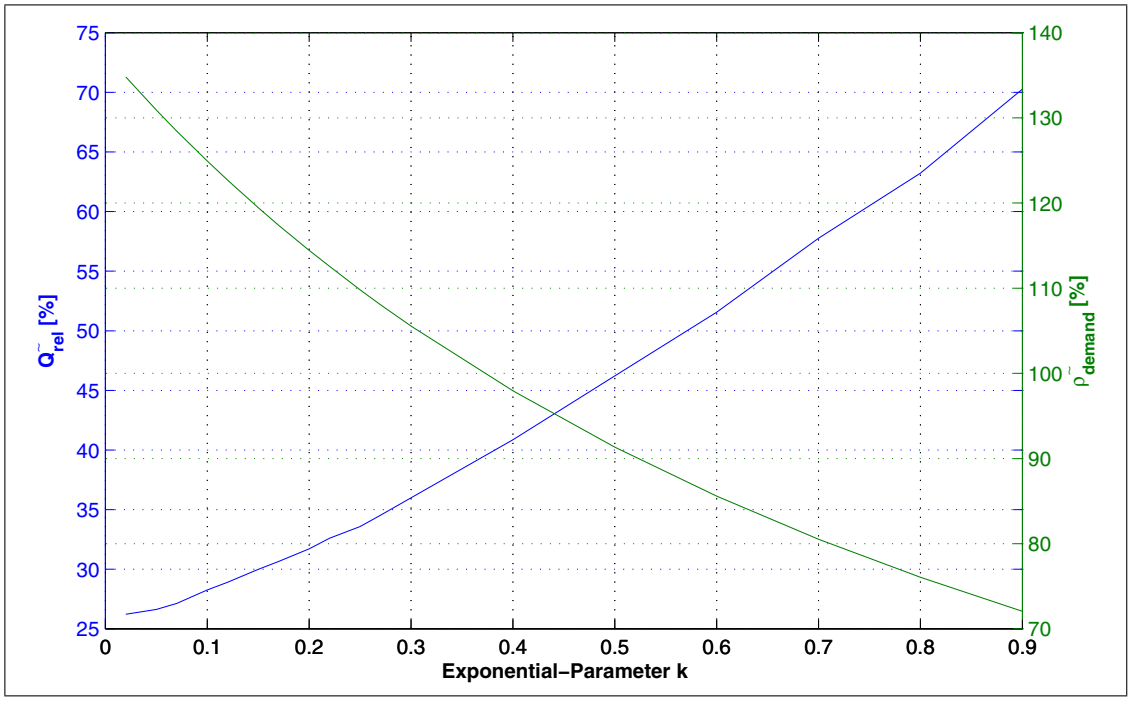


Abbildung 5.13: $Q^{\sim}_{rel}$ bei Variation der Ereignisströme

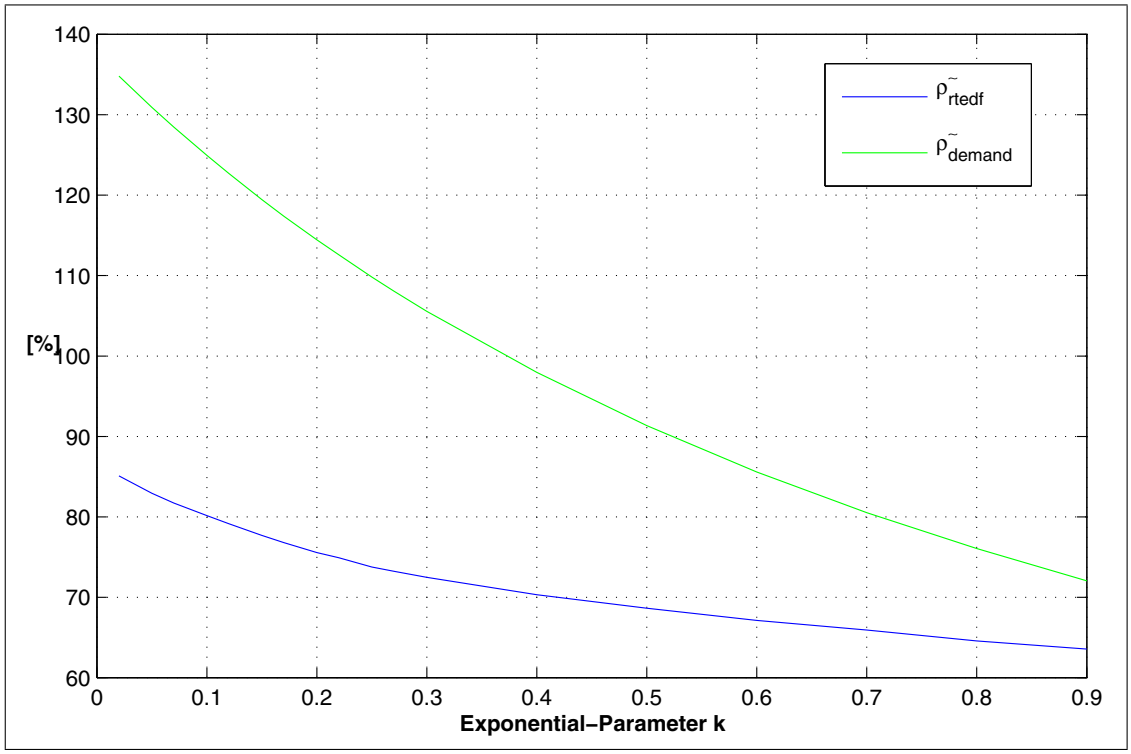


Abbildung 5.14: $\rho^{\sim}_{rtedf}$ bei Variation der Ereignisströme

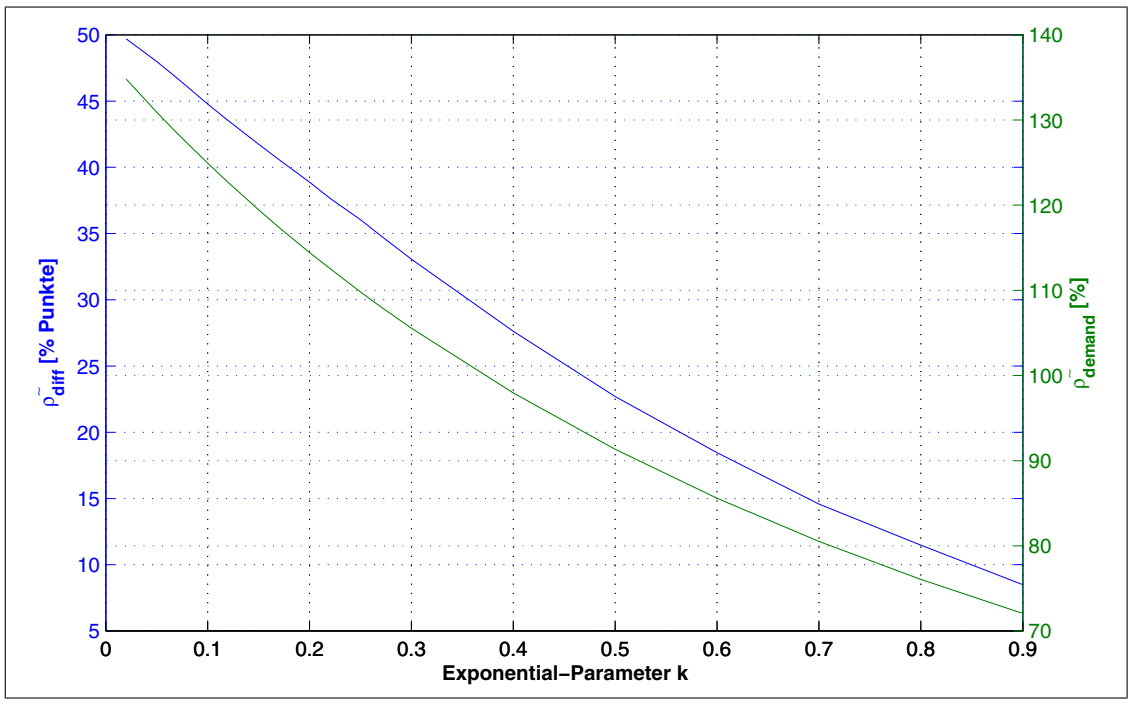


Abbildung 5.15: $\rho^{\sim}_{\text{diff}}$ bei Variation der Ereignisströme

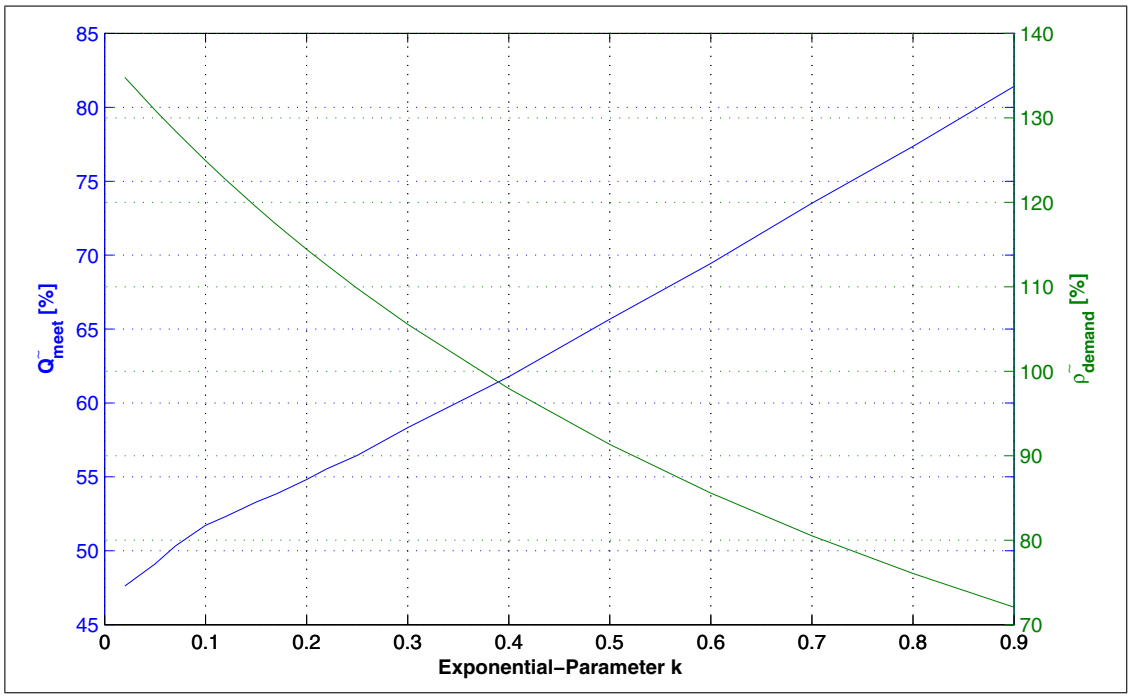


Abbildung 5.16: Relativer Anteil an primären Deadlines $Q^{\sim}_{meet}$, die bei Variation der Ereignisströme eingehalten werden

5.6 Variation der Ausführungzeiten und Ereignisströme

Werden sowohl variierende Rechenzeiten als auch variierende Ereignisströme betrachtet, so überlagern sich die Effekte beider Faktoren.

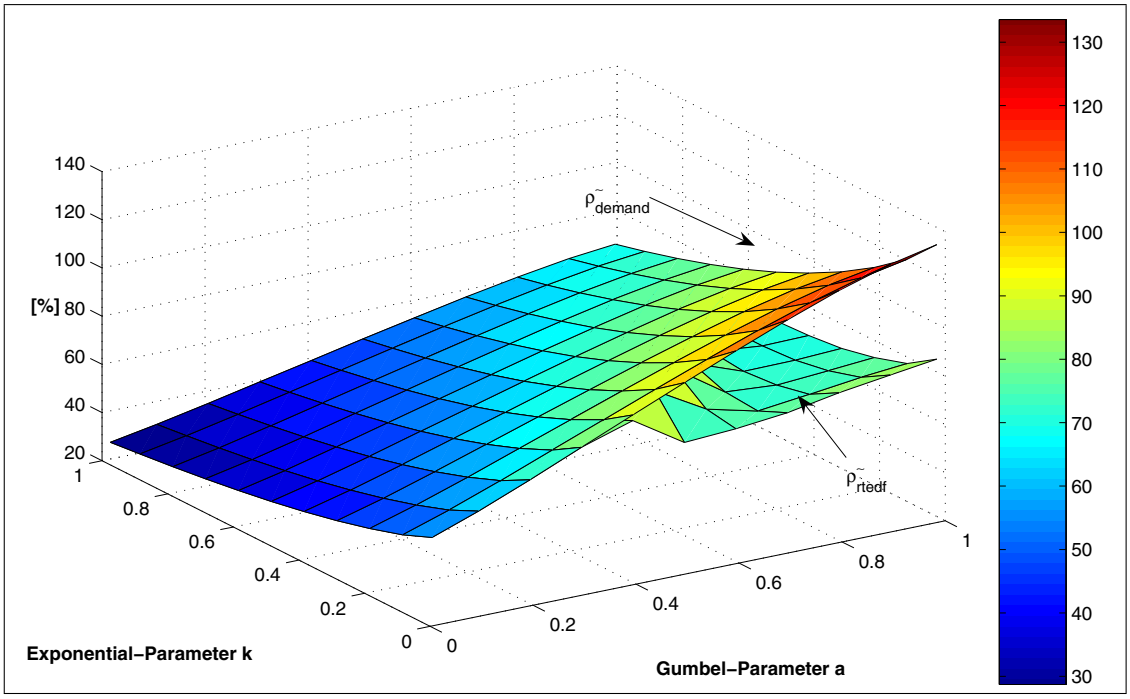


Abbildung 5.17: Primär angeforderter und effektiver Rechenzeitbedarf bei Variation der Ausführungszeiten und Ereignisströme $\rho^{\sim}_{\text{demand}}$ und $\rho^{\sim}_{\text{rtedf}}$

Alle diskutierten Ergebnisse resultieren aus der Verwendung eines Gumbel Parameter $b = 9$.

5.6.1 Simulationsergebnisse

Abbildung 5.17 zeigt sowohl den primär angeforderten Rechenzeitbedarf $\rho^{\sim}_{\text{demand}}$ als auch den effektiv realisierten Rechenzeitbedarf $\rho^{\sim}_{\text{rtedf}}$. Es ist zu erkennen, dass für den Bereich, in dem der durchschnittlich angeforderte Rechenzeitbedarf unter 90% liegt, der RTEDF Algorithmus ein reines EDF Verhalten zeigt.

Dieses Verhalten ist auch bei der Differenz der beiden Rechenzeitanforderungen in Abbildung 5.18 zu erkennen.

Wird eine mögliche Überlast vom RTEDF Algorithmus detektiert, so wird in den Überlastmodus geschaltet. Das führt auch dazu, dass neue Ereignisse verworfen werden müssen. Dadurch kann es auch vorkommen, dass aufgrund der aktuellen Lastsituation bei einer Transaktionsinstanz mit einer verlängerten Deadline die primäre Deadline trotzdem noch eingehalten werden kann. Dieses Vorgehen zeigt sich in der Metrik für den relativen Anteil der Transaktionen $Q^{\sim}_{\text{meet}}$ (Abb. 5.21), die ihre primäre Deadline einhalten konnten. Dabei ist zu erkennen, dass selbst

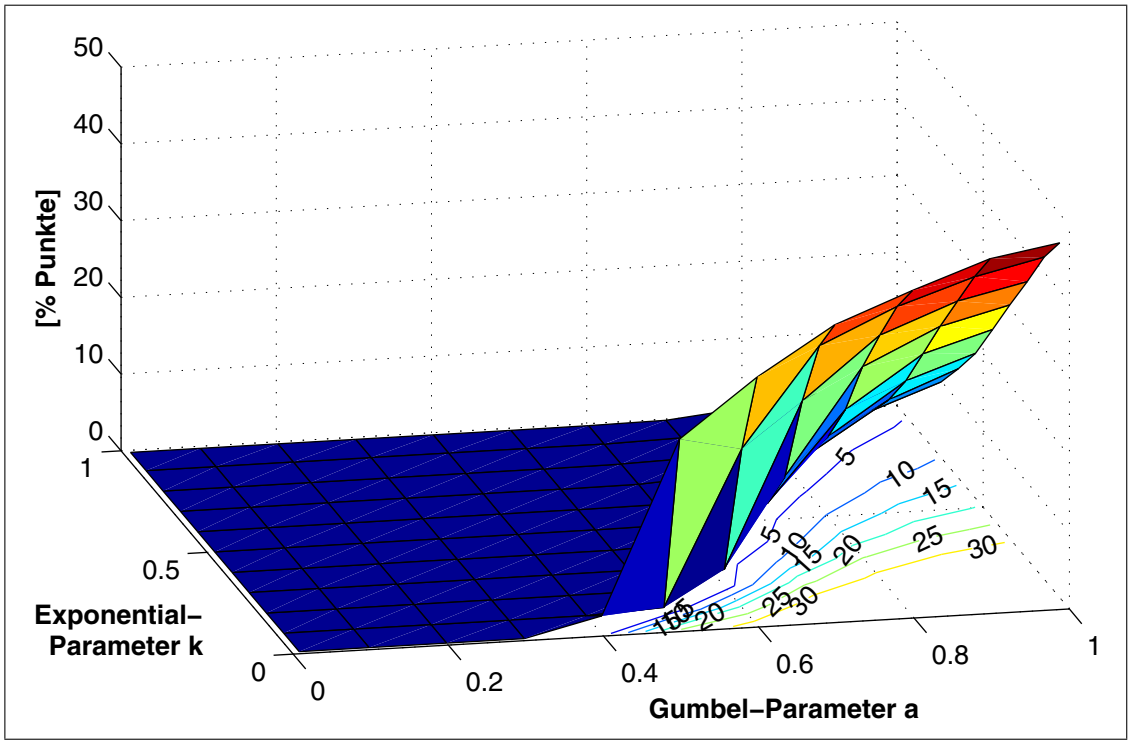


Abbildung 5.18: Differenz zwischen dem primär angeforderten und dem effektiv realisierten Rechenzeitbedarf $\rho^{\sim}_{\text{demand}} - \rho^{\sim}_{\text{rtedf}}$ bei Variation der Ausführungszeiten und Ereignisströme

bei den Simulationen im Überlastbereich ein gemittelter durchschnittlicher Anteil von mehr als 60% aller Instanzen ihre primäre Deadline einhalten konnten.

5.7 Abschließende Bewertung der Simulationsergebnisse

Die Simulationsergebnisse für das Beispieltransaktionsset zeigen, dass in allen drei Szenarien (reine Variation der Ausführungszeiten, reine Variation der Ereignisströme und die Kombination beider Variationen) bis hin zu einem Rechenzeitbedarf von 90% keine Umschaltung in den Überlastmodus notwendig wird.

Das frühe Umschalten in den Überlastmodus bei der Variation der Ereignisströme ist durch die ungünstige Verteilung des Rechenzeitbedarfes auf nur drei Transaktionen bedingt. Das Verhalten ist dem Verhalten bei der Variation der Ausführungszeiten ähnlich, wenn die Anzahl der Transaktionen steigt.

Dieser nachteilige Effekt kommt bei der kombinierten Simulation nicht so stark zur Geltung. Dieses kombinierte Szenario spiegelt dabei den realen Betrieb wider, in dem sowohl die Ausführungszeiten als auch die Ereigniszeitpunkte Variationen unterworfen sind.

Das Eingreifen durch die Umschaltung in den Überlastmodus verschafft dem Laufzeitsystem

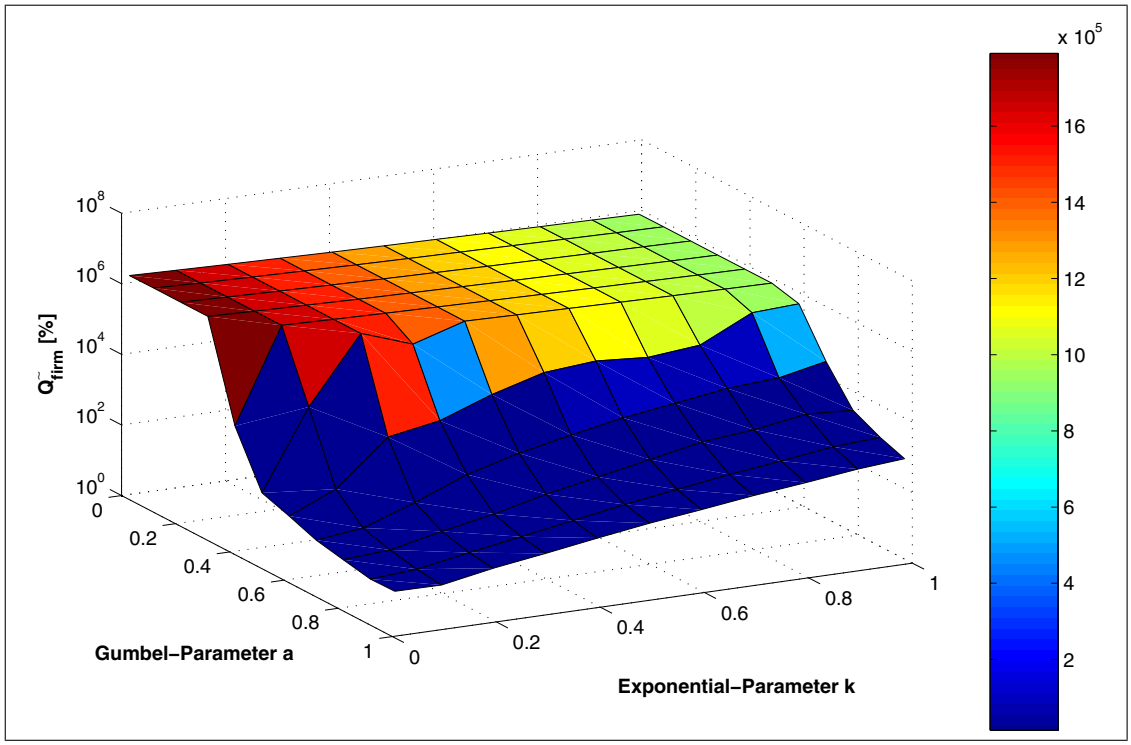


Abbildung 5.19: Gleichgewichteter normierter Mittelwert konsekutiver *Firm–*/*QoS*–Instanzen $Q^{\sim}_{firm}$ bei Variation der Ausführungszeiten und Ereignisströme

einen erhöhten *Slack*, so dass in allen drei Szenarien, trotz verlängerter Deadlines ein großer Teil der *Delta*–Transaktionsinstanzen ihre primäre Deadline einhalten.

Effektiv zeigt sich im Überlastmodus ein Verhalten, dass mit einer Reduktion der Bearbeitungsrate gleichzusetzen ist.

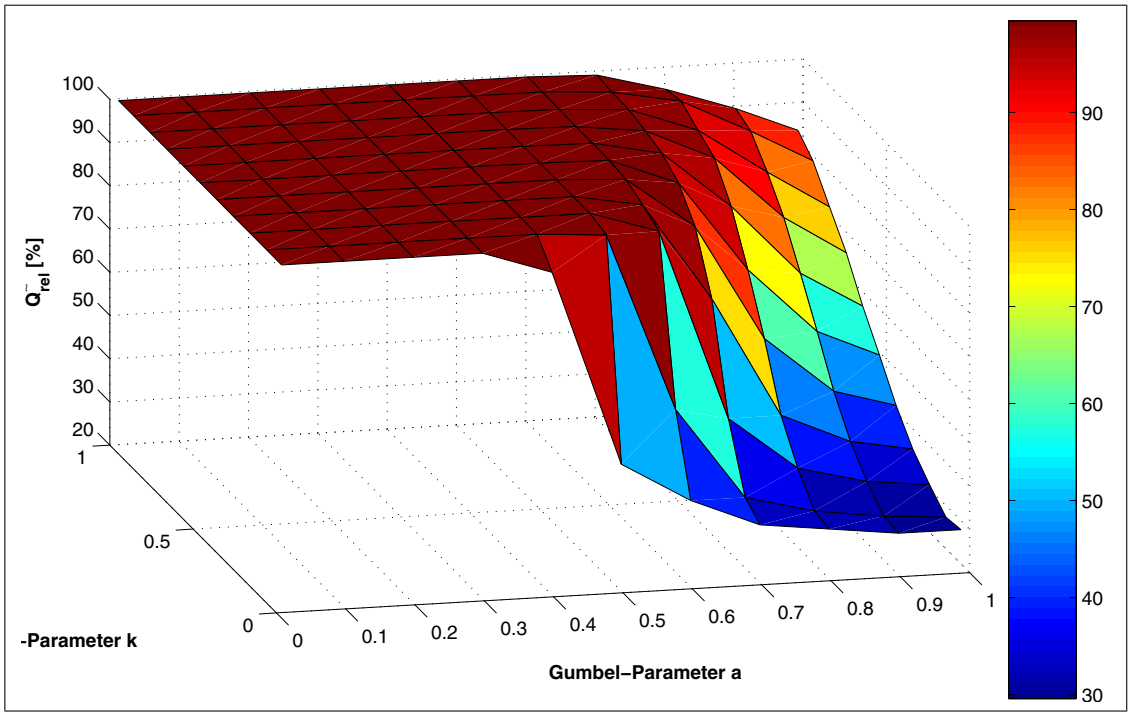


Abbildung 5.20: Gleichgewichteter relativer Anteil an *Firm–/QoS–*Instanzen $Q^{\sim}_{rel}$ bei Variation der Ausführungszeiten und der Ereignisströme

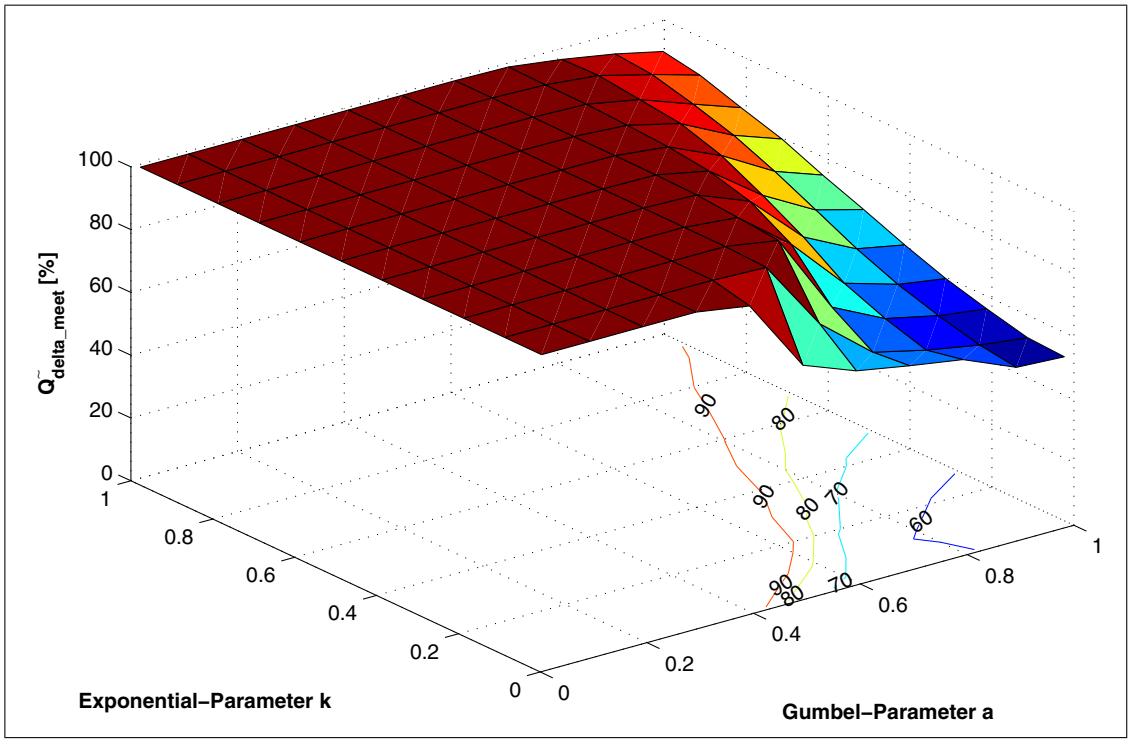


Abbildung 5.21: Relativer Anteil an primären Deadlines $Q^{\sim}_{meet}$, die bei Variation der Ausführungszeiten und Ereignisströme eingehalten werden

6 Realisierung und Integration

Das in dieser Arbeit vorgestellte TEDF Scheduling und das RTEDF Überlastbehandlungsverfahren wurden zur Verifikation der Realisierbarkeit der Konzepte prototypisch implementiert. Die Implementierung der verwendeten Algorithmen und der Datenstrukturen des TEDF Schedulings in ein Realzeitbetriebssystem wird in Abschnitt 6.2 behandelt. Es werden sowohl die internen Strukturen und Routinen als auch die für den Anwender sich ändernde semantische Schnittstelle (API) des Betriebssystems beschrieben.

Die notwendigen Erweiterungen des Betriebssystems zur Kommunikation mit dem Coprozessor sind in Abschnitt 6.3 beschrieben. Die verwendete Coprozessorarchitektur und die auf dieser Architektur implementierten Algorithmen und Datenstrukturen werden im Hinblick auf Speicherverbrauch und Rechenzeitbedarf in Abschnitt 6.4 analysiert.

Eine beispielhafte Anbindung des TEDF Laufzeitsystems an ein *State–of–the–Art* CASE Werkzeug sowie einige Vorschläge zum *Coding Style* mit TEDF werden in den Abschnitten 6.5 und 6.6 beschrieben.

Zu Beginn dieses Kapitels wird zunächst auf die Einbettung des RTEDF Laufzeitsystems in den Entwurfsprozess eingegangen.

6.1 Entwurfsprozess

Im Rahmen der vorliegenden Arbeit wurde mit dem Konzept auf einen minimalen Eingriff in die etablierte Entwurfsmethodik verfolgt. In Abbildung 6.1 ist die Einbettung in den Entwurfsprozess dargestellt. Ausgehend von einem mit einem CASE–Werkzeug erstellten Softwaredesign werden bei funktionalen Simulationen und Testszenarien Sequenzdiagramme erzeugt. Die zu spezifizierenden Metadaten für einzelne Transaktionen wie die λ– und ψ–Pattern und die Eventfunktionen können als Annotationen im Sequenzdiagramm deklariert werden. Diese Daten werden ebenso wie alle möglichen Pfade aus den Sequenzdiagrammen durch Parsen extrahiert und zur Generierung des Transaktionsgraphen verwendet. Sowohl die Annotationen als auch die Sequenzdiagramme liegen zur Dokumentation und Testzwecken bei heutigen Entwurfsprozessen bereits vor. Der Entwickler hat somit keinen zusätzlichen Modellierungsaufwand zu bewältigen.

Für einen Realzeitnachweis und *Quality of Service* Simulationen einzelner Szenarien ist das Wissen über die Ausführungszeiten von entscheidender Bedeutung. Auf diesem Gebiet gibt es zahlreiche Ansätze, mit denen sich die Ausführungszeiten entweder mit Hilfe von *Hardware in the Loop*– oder über zyklengenaue Instruktionssatzsimulatoren ermitteln lassen. Durch eine entsprechende Instrumentierung des Target Codes können beispielsweise bei einer Cosimulation mit dem realen Target Ausführungszeiten der Threadaktionen gemessen werden.

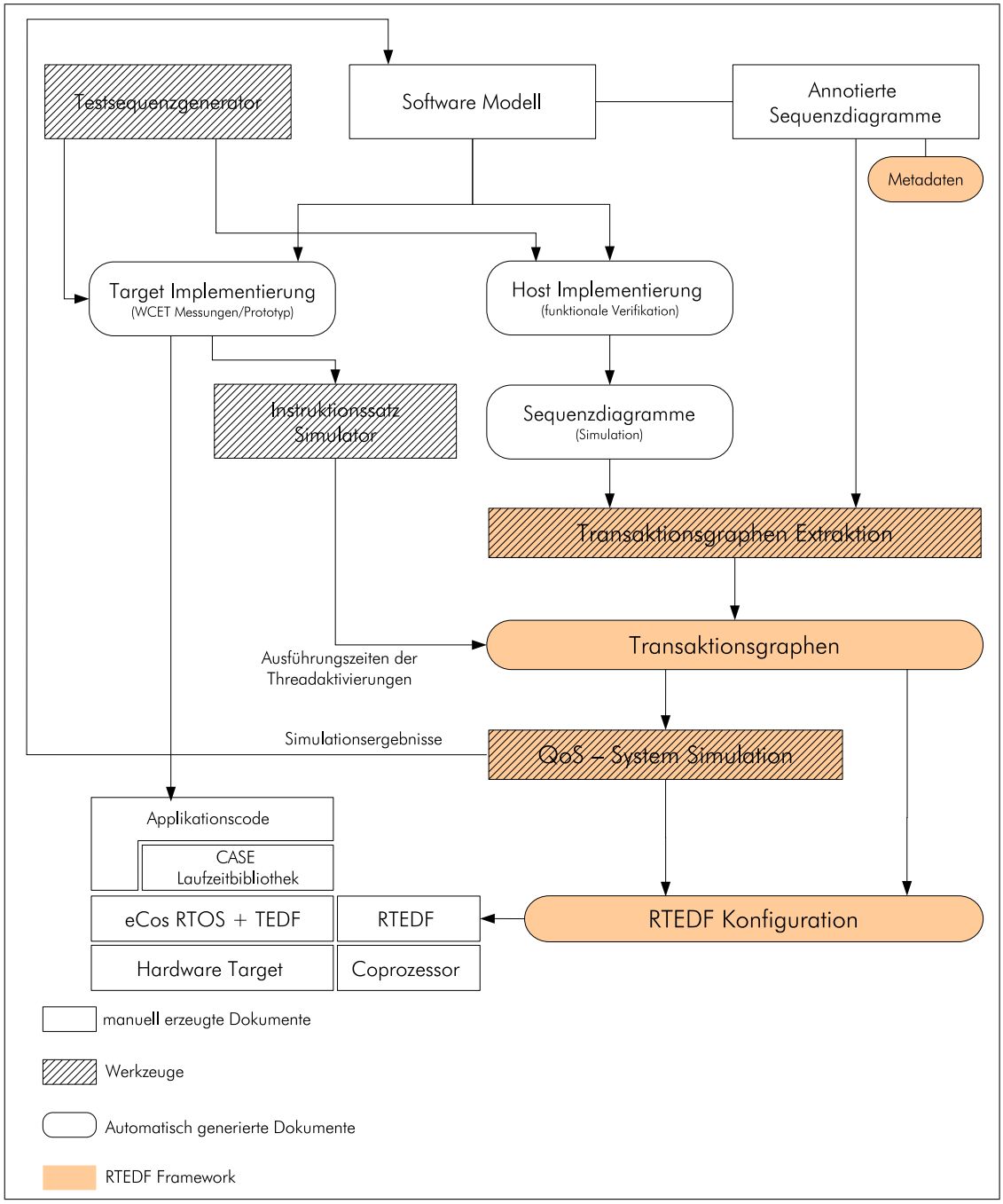


Abbildung 6.1: Anbindung des RTEDF Laufzeitsystems an den Entwurfsprozess

Die anschließende *QoS*–System Simulation liefert zum einen den Realzeitnachweis für das Transaktionsset und zum anderen die Größen $Q^{\sim}_{\text{firm}}$, $Q^{\sim}_{\text{delta}}$, $Q^{\sim}_{\text{rel}}$, $\rho^{\sim}_{\text{demand}}$, $\rho^{\sim}_{\text{rtedf}}$ und $Q^{\sim}_{\text{meet}}$.

Diese Daten können im Softwareentwurfswerkzeug genutzt werden, um die Einhaltung der Anforderungen durch funktionale Simulationen zu verifizieren.

Gleichzeitig ist es dem RTEDF Framework möglich, eine Konfiguration des Coprozessors zur Überlastüberwachung durchzuführen, so dass der bestehende Entwurfsfluss keine zusätzlichen manuellen Eingriffe erfordert.

6.2 TEDF Laufzeitsystem

Das Laufzeitsystem ist so aufgebaut, dass das TEDF Scheduling ohne eine Überlastüberwachung genutzt werden kann. Der Fokus liegt jedoch im Zusammenspiel zwischen TEDF und RTEDF und daher wird im Folgenden auf die TEDF + RTEDF Kombination direkt eingegangen.

Für die Implementierung des TEDF Laufzeitsystems wurde das Open Source Realzeitbetriebssystem eCos [26] verwendet, dass von Red Hat übernommen wurde und unter der GPL erhältlich ist. eCos unterstützt eine Vielzahl von Architekturen sowie Peripheriebausteinen (Ethernet, USB, PCMCIA) bei einem vergleichsweise schlanken und effizienten Footprint (ca. 80 kB).

Die Modifikationen, die an eCos vorgenommen wurden, beziehen sich zum einen auf neue Kernelfunktionalität und zum anderen auf ein abgeändertes eCos *native API* für die Applikationsentwicklung. Die Strukturerweiterungen sind in Abbildung 6.2 dargestellt.

RTEDF Modul Das RTEDF Modul kapselt die Kommunikation zwischen dem TEDF Laufzeitsystem und dem RTEDF Coprozessor. Neu eintreffende Ereignisse aus dem einbettenden System werden dem Coprozessor gemeldet. Darüber hinaus werden Anforderungen vom Coprozessor empfangen und im Laufzeitsystem durchgeführt.

Cyg_Message Die Klasse Cyg_Message kapselt die Scheduling Metadaten, die für den TEDF Scheduler notwendig sind. Zudem beinhaltet die Klasse Daten, die mit einer Nachricht transportierten werden.

Cyg_Mbox_TEDF Cyg_Mbox_TEDF stellt eine Nachrichtenwarteschlange dar, in der die Nachrichten nach ihrer absoluten Deadline einsortiert werden. Jedem Thread wird ein Objekt der Klasse Cyg_Mbox_TEDF zugewiesen.

Cyg_Thread Die Thread Datenstruktur stellt die notwendigen Variablen zur Speicherung der zur Laufzeit übermittelten absoluten Deadline zur Verfügung.

Cyg_TEDF_Scheduler Der Scheduler stellt die eigentliche EDF Scheduling Funktionalität dar. Aktive Threads werden nach ihrer absoluten Deadline sortiert und in einer Warteschlange verwaltet.

[0)] Es wurde das eCos *native API* verwendet, um keine syntaktischen oder semantischen Änderungen an bestehenden Standards vorzunehmen.

Cyg_DeadlineClock Die Zeitauflösung der Deadlines stellt in der Realisierung eine besondere Anforderung an die Systemarchitektur dar. Über die Cyg_DeadlineClock Struktur wird ein Timerbaustein mit entsprechender Auflösung angesprochen.

Die *Hardware Abstraction Layer (HAL)* wurde um Methoden zum Ansprechen des Deadline Timer Bausteines und der Schnittstelle zum Coprozessor erweitert.

Im Folgenden soll zunächst auf die für das TEDF Scheduling relevanten Kernelstrukturen eingegangen werden. Anschließend werden die zur Kommunikation mit dem RTEDF Coprozessor notwendigen Funktionen beschrieben. Die Beschreibung des Kernel API und seiner Methoden beschränkt sich dabei im Folgenden nur auf die relevanten Funktionen.

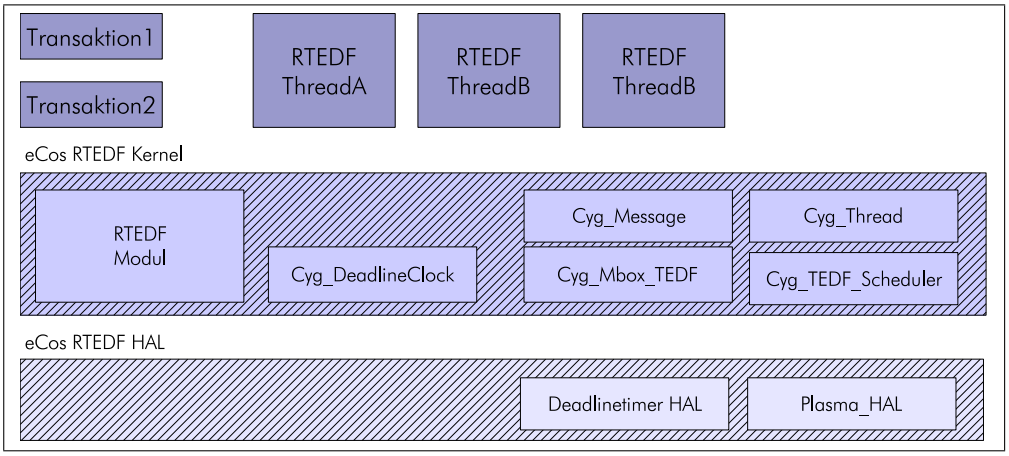


Abbildung 6.2: Erweiterungen der eCos Softwarearchitektur

6.2.1 Kommunikationsstruktur

Die transaktionsbasierte Laufzeitsemantik baut im Kern auf eine nachrichtenbasierte Kommunikation auf. Eine Transaktion besteht aus einer Reihe von Nachrichten, die sequentiell verschiedene Threads aktivieren. Daher kommt dem Betriebssystemprimitiv der Mailbox und dem Nachrichtenformat besondere Bedeutung zu.

Eine Transaktion startet mit einem initialen Event. So kann beispielsweise ein Interruptsignal eines Peripheriebausteines, ein abgelaufener Systemtimer oder ein bestimmter Threadzustand zur Generierung des Startevents angesehen werden. Ein abstraktes übergreifendes Konzept, das alle möglichen Anwendungsfälle abdeckt, stellt daher die Kommunikation mit Nachrichten[1)] dar.

Interrupt

Die interruptgesteuerte Aktivierung von Ereignissen stellt eine effiziente Synchronisationstechnik dar, die im Umfeld von konfigurierbaren Peripheriebausteinen eingesetzt wird. Dabei wer-

[1)] Eine optionale Realisierung auf Basis von Mutexes wäre ebenfalls zur Realisieurng der TEDF Laufzeitsemantik möglich.

den Interrupts genau dann erzeugt, wenn bestimmte externe Ereignisse die Interaktion der Software mit der CPU benötigen. Diese Ereignisse stellen für die Semantik der transaktionsbasierten Laufzeitsemantik initiale Ereignisse dar.

Alarm

Jedes Realzeitbetriebssystem besitzt eine Zeitverwaltung, die unter anderem von Threads für zeitgesteuerte Aufgaben verwendet werden kann. Für diese Funktionalität stehen in eCos Alarm Handler zur Verfügung, die beim Überlauf eines Zählerobjektes eine vorab definierte Funktion aufrufen. Das Zählerobjekt ist in der Regel an den Real–Time Zähler gebunden, der durch einen Hardwarebaustein inkrementiert wird. Der Alarm Handler kann bei TEDF derart ausgelegt werden, dass eine initiale Nachricht einer Transaktion initiiert wird.

Zustandsbasierte Aktivierung

Oft existieren sogenannte Serverthreads, die aufgrund von aktuellen Zustandsdaten bestimmte Dienste erbringen. Bei ereignisgesteuerten Systemen ist das beispielsweise genau dann der Fall, wenn bestimmte Messwerte überschritten werden. In diesem Fall kann eine initiale Nachricht auch durch einen aktiven Thread generiert werden.

Für alle Arten der initialen Transaktionsinitiierung können sowohl die minimalen Ereigniszeitpunkte als auch die Ereignisströme angegeben werden.

6.2.2 Nachrichten

Bei TEDF repräsentiert die Nachricht die eigentliche Aufgabe. Es wurde daher eine Datenstruktur bereitgestellt, die die zugehörigen Metadaten kapselt. Die Elemente dieser Datenstruktur sind in Tabelle 6.1 dargestellt.

Die Konfiguration und Erstellung von Nachrichten wird von der Applikation durchgeführt. Das zur Verfügung stehende Kernel API und die Methoden der Klasse *Cyg_Message* sind in Tabelle 6.2 aufgelistet.

6.2.3 Mailboxen

Die Klasse *Cyg_Mbox_TEDF* stellt die Kernfunktionalität des transaktionsbasierten Ansatzes zur Verfügung. Auf der Applikationsseite stehen Methoden zum Einfügen sowie zum Empfangen von Nachrichten zur Verfügung. Auf der Kernelseite werden initiale Ereignisse von Transaktionen dem RTEDF Modul mitgeteilt, um die notwendige Information für die RTEDF Funktionalität bereitzustellen. Jede Mailbox verwaltet als interne Datenstruktur eine verkettete Liste von *Cyg_Message* Objekten. In der bestehenden Implementierung wird jedem Thread nur eine Mailbox zugeordnet. In Tabelle 6.3 sind die Erweiterungen des Kernel API dargestellt.

Das Weiterreichen der absoluten Deadline sowie die Vererbung der Deadline bei transaktionsbasierter Prioritätsinversion wird mit privaten Methoden der Klasse *Cyg_Mbox_TEDF* realisiert.

Attributnamen	**Beschreibung**
Name	Eindeutige Namensbezeichnung, die zur Identifikation der Nachricht mit einem CASE Werkzeug dient.
ID	Eindeutiger Identifier, mit dem die Transaktion im TEDF und RTEDF System identifiziert wird.
Relative Deadline	Relative Deadline einer Transaktion.
Delta Time	*Delta* Zeitintervall, dass bei einer δ–Phase zur bestehenden Deadline hinzuaddiert wird.
Absolute Deadline	Bei der initialen Aktivierung einer Transaktion wird die absolute Deadline ermittelt und eingetragen.
Daten	Zeiger auf die mit der Nachricht transportierte Datenstruktur
MBox	Zeiger auf die Mailbox, in der die Nachricht gerade eingereiht ist.
Busy	Ein Flag, dass anzeigt, ob diese Nachricht noch Teil einer aktiven Transaktionsinstanz ist.
Extended	Ein Flag, dass anzeigt, ob es sich bei dieser Nachricht um eine *Delta* Transaktionsinstanz handelt.
Firm	Ein Flag, dass anzeigt, ob es sich bei dieser Nachricht um eine *Firm*–Transaktionsinstanz handelt.
Original Absolute Deadline	Zur Transaktionsinitiierungszeit ermittelte absolute Deadline. Sie wird bei Deadlinevererbungsprotokollen verwendet, um die ursprünglich festgelegte absolute Deadline zu sichern.

Tabelle 6.1: Attribute der Klasse *Cyg_Message*

Syntax:	`void` **cyg_message_create** (*cyg_handle_t *handle,* *cyg_message *message,* `char` **name,* *cyg_priority_t rel_dline,* *cyg_priority_t delta_dline,* *cyg_uint32 id*);
Beschreibung:	Nachrichten werden initial vor dem eigentlichen Start der Applikation initialisiert. Alle für TEDF relevanten Metadaten werden bei der Nachrichtenerstellung initialisiert. Über den Zeiger **handle* kann auf das Nachrichtenobjekt zugegriffen werden.
Syntax:	`void` **cyg_message_free** (*cyg_handle_t handle*);
Beschreibung:	Die Methode kennzeichnet das Ende einer Transaktionsinstanz.
Syntax:	*cyg_bool_t* **cyg_message_check_if_busy** (*cyg_handle_t handle*);
Beschreibung:	Liefert den Wert *Wahr* zurück, wenn die Nachricht noch Teil einer aktiven Transaktionsinstanz ist.
Syntax:	`void` **cyg_message_data_set** (*cyg_handle_t handle*, `void` **data*);
Beschreibung:	Setzt den Zeiger in der Nachrichtenstruktur auf die zu versendenden Daten.
Syntax:	`void*` **cyg_message_data_get** (*cyg_handle_t handle*);
Beschreibung:	Liefert einen Zeiger auf die in der Nachricht enthaltene Datenstruktur.

Tabelle 6.2: API der Klasse *Cyg_Message*

Die privaten API Methoden für die Kernelebene sind in Tabelle 6.4 dargestellt.

Jede Mailbox besitzt einen Zeiger *get_threadq*, der auf eine Threadqueue zeigt, in der alle Threads, die auf Nachrichten warten eingereiht sind. Bei TEDF kann dies immer nur ein und derselbe Thread sein, da diese Mailbox als Kommunikationsport zu diesem Thread angesehen werden kann. Die Methoden *wakeup_winner* und *update_message* werden bei einer Aktivierung eines Threads durch eine Nachricht, beziehungsweise beim Versenden einer Nachricht aufgerufen. Sie sorgen für die korrekte Weiterleitung der absoluten Transaktionsdeadline beim Übergang von Nachricht zu Thread und Thread zu Nachricht. Weiterhin existieren in der Klasse zwei Methoden (*exchange_messages*, *dip_boost*), die für die Deadlinevererbung bei transaktionsbasierter Blockierung verwendet werden.

Transaktionsbasierte Blockierung

Die Behandlung von transaktionsbasierter Blockierung wurde in Abschnitt 3.4.2 bereits definiert. Das entsprechende *Transaction Deadline Inheritance* Protokoll wird mit der Methode *dip_boost* realisiert. Wird bei der Einreihung einer neuen Nachricht in die Mailbox festgestellt, dass dessen Deadline kürzer ist als die Deadline der zu diesem Zeitpunkt aktiven Nachricht, so vererbt die Methode *dip_boost* diese Deadline an den aktiven oder lauffähigen Thread.

Syntax:	`void` **cyg_mbox_create** (*cyg_handle_t *handle cyg_mbox *mbox*);
Beschreibung:	Liefert einen Zeiger auf die Mailboxstruktur, mit dem auf die Mailbox zugegriffen werden kann.
Syntax:	`void*` **cyg_mbox_get** (*cyg_handle_t mbox*);
Beschreibung:	Liefert einen Zeiger auf das Datenobjekt in der Nachricht. Der Aufruf ist blockierend, so dass bei einer leeren Mailbox der Thread in den Zustand *Wartend* versetzt wird.
Syntax:	*cyg_bool_t* **cyg_mbox_put** (*cyg_handle_t mbox*, `void` **item*);
Beschreibung:	Startet eine Transaktion durch Ablegen einer initialen Nachricht in die Mailbox mit dem Handler *mbox*. Der Aufruf ist asynchron und nicht blockierend.
Syntax:	*cyg_bool_t* **cyg_mbox_isr_put** (*cyg_handle_t mbox*, `void` **item*);
Beschreibung:	Versendet die initiale Nachricht, die als initiale Transaktion deine Nachricht an die Mailbox mit dem Handler *mbox*. Der Aufruf ist asynchron und nicht blockierend.
Syntax:	`void` **cyg_mbox_set_owner** (*cyg_handle_t mbox*, *cyg_thread *owner*);
Beschreibung:	Jede Mailbox darf genau zu einem Thread assoziiert sein.

Tabelle 6.3: API der Klasse *Cyg_Mbox_TEDF*

Ein Spezialfall des transaktionsbasierten Blockierens ergibt sich, wenn ein niedrigpriores Ereignis[2)] einen bereits wartenden Thread in den Zustand *Lauffähig* versetzt. Dieser Thread wird aufgrund der längeren Deadline in die *Run Queue* eingereiht, startet die Abarbeitung der Nachricht noch nicht. Falls nun ein weiteres Ereignis mit einer kürzeren Deadline kurz darauf eintreffen sollte, so wäre es effizient, wenn diese Nachrichten noch getauscht werden könnten, da die erste Nachricht noch keine Bearbeitung erfahren und somit auch keine Systemzustandsänderung herbeigeführt hat. Für diese Funktionalität steht die Methode *exchange_messages* zur Verfügung.

6.2.4 Threads

Die Datenstruktur der Threads ist durch minimale Änderungen gegenüber der prioritätsbasierten Implementierung gekennzeichnet. Threadattribute zur Speicherung der absoluten Deadline, der *Delta*–Deadline und dem Metazustand des Threads (*QoS, Firm, Delta*) entsprechen den Attributen der Nachrichtenklasse. Die Methoden zur Änderung des Threadzustandes bleiben hingegen bestehen.

[2)] Das entspricht einer längeren absoluten Deadline als die des gerade aktiven Threads.

Syntax:	*Cyg_ThreadQueue* **get_threadq**();
Beschreibung:	Bei einem blockierenden *cyg_mbox_get* Aufruf im Applikationscode wird der Thread in die *get_threadq* Struktur aufgenommen. Bei TEDF kann es immer nur einen einzigen Thread pro Mailbox geben.
Syntax:	*Cyg_Message** **msg_dequeue**();
Beschreibung:	Entfernt eine Nachricht aus der Mailbox.
Syntax:	`void` **msg_enqueue**(*Cyg_Message *msg*);
Beschreibung:	Fügt Nachricht in die Mailbox ein.
Syntax:	`void` **wakeup_winner**(`void`);
Beschreibung:	Weckt den wartenden Thread in *get_threadq* auf.
Syntax:	`void` **update_thread**(*Cyg_Thread *thread*, *Cyg_Message *msg*);
Beschreibung:	Der Thread erhält die Schedulingdaten der gerade konsumierten Nachricht.
Syntax:	`void` **update_message** (*Cyg_Thread *thread*, *Cyg_Message *msg*);
Beschreibung:	Die Nachricht erhält die Schedulingdaten des Threads, der die Nachricht weitersendet.
Syntax:	`void` **exchange_messages**();
Beschreibung:	Tauscht eine bereits aus der Mailbox genommene Nachricht aus.
Syntax:	`void` **dip_boost**(*cyg_priority new_dl*);
Beschreibung:	Für den Fall der Deadline Vererbung wird die Deadline des aktiven Threads herabgesetzt.
Syntax:	`void` **extend_all_qos**();
Beschreibung:	Alle normalen Nachrichten werden zu *Delta*–Transaktionen erweitert.

Tabelle 6.4: Private Methoden der Klasse *Cyg_Mbox_TEDF*

6.2.5 Scheduler

Das zentrale Konzept der Trennung der Funktionalitäten spiegelt sich auch in der Schedulerimplementierung wider. Der Schedulingalgorithmus basiert auf dem Earliest Deadline First Scheduling und verwaltet eine einzige Queue (*Run Queue*), in der alle lauffähigen Threads nach ihrer absoluten Deadline sortiert eingefügt werden. Der Unterschied zu einem prioritätsbasierten Scheduler liegt darin, dass die einzelnen Threads nach der absoluten Deadline sortiert werden und daher zusätzlicher Rechenaufwand berücksichtigt werden muss. Zudem meldet der Scheduler dem RTEDF Modul, wenn nur noch der *Idle Thread* in der *RunQueue* ist.

6.2.6 Deadline Clock

Die Priorität einer Transaktion bestimmt sich aus der beim initialen Ereignis absolut gemessenen Zeit zuzüglich der einzuhaltenden Deadline. Deshalb ist eine Mindestgranularität der zeitlichen Auflösung gefordert.

Die Zeitverwaltung von Realzeitbetriebssystemen, die ausschließlich prioritätsbasiertes Scheduling anbieten, nutzen dazu unterschiedliche Strategien. In dieser Arbeit wurde das Konzept des freilaufendenen Timers verwendet, der nach Überschreiten eines bestimmten Wertes die Korrektur der verwendeten Deadlines notwendig macht. Dieses implementierungsspezifische Problem zeigt sich beispielsweise bei 32–Bit Bausteinen nach ca. 11 h bei einer Auflösung von 10 μs. Dieser zur Laufzeit stattfindende *Wrap–Around* kostet zusätzliche Zeit, die in der Realzeitanalyse zu berücksichtigen ist.

Allerdings zeigt sich bei vielen leistungsfähigen eingebetteten Systemen, dass 64–Bit Zähler selbst bei 32-Bit Architekturen bereits zur Verfügung stehen. Eingebettete Systeme mit 64–Bit Architektur stellen zur Zeit noch die Ausnahme dar. Ein Einsatz wird in Zukunft jedoch nicht unrealistisch sein. Mit 64–Bit Zählerbausteinen wären dann bei identischer Genauigkeit Intervalle von $5,8 \cdot 10^6$ Jahren möglich.

6.2.7 Kollaboration der TEDF Kernelobjekte

Zur Verdeutlichung des Kontrollflusses der TEDF Funktionalität wird ein einfaches Beispiel herangezogen, das in Abbildung 6.3 dargestellt ist.

Beispiel 6.1
Eine einfache Applikation besteht aus einem Thread, einer Mailbox und einer Interrupt Service Routine. Zu Beginn ruft der Applikationscode des Threads die get*–Methode der ihr zugeordneten Mailbox auf. Da noch keine Nachricht in der Mailbox vorhanden ist, wird der Thread aus der Run Queue entfernt, in die ThreadQueue*[3)] *der Mailbox eingetragen und schließlich ein* Context Switch *mit dem Befehl reschedule angestoßen.*

Der Thread verbleibt im Ruhezustand solange keine Nachricht in die Mailbox eingereiht wird.

Findet ein externes Ereignis statt, so dass über die Funktion cyg_mbox_isr_put eine initiale Nachricht in die Mailbox eingereiht wird, so wird bei wartendem Thread die Funktion wakeup_winner() aufgerufen. Diese Methode entfernt sowohl den Thread als auch die Nachricht aus der Warteschlange und ruft die Methode update_thread auf. Diese Methode ist für die Vererbung der absoluten Deadline an den Thread verantwortlich. Es werden die absolute Deadline, die mögliche Delta*–Deadline, der Name der Nachricht und die Information, ob es sich um eine* Delta*–Transaktion handelt, übertragen. Anschließend wird der Thread aufgeweckt, indem er in die Ready Queue des Schedulers eingefügt wird. Analog zum Empfang von Nachrichten, wird das Weitersenden der Transaktion (Nachricht) durch die Methode cyg_mbox_put bewerkstelligt.*

[3)] Die TEDF Semantik erlaubt nur eine 1:1 Relation zwischen Thread und Mailbox. Daher wurde die Queue Implementierung durch eine Pointervariable ersetzt.

In diesem Fall werden die Schedulingdaten mit der Methode update_message an die versendete Nachricht vererbt.

Abbildung 6.3 zeigt den Kontrollfluss[4)] mit den beteiligten Funktionen für das Beispiel 6.1. Die bisher beschriebenen Kernelerweiterungen beziehen sich auf die Realisierung des TEDF Laufzeitverhaltens. Die Kernelerweiterungen zur Überlastbehandlung RTEDF werden im Folgenden beschrieben.

6.3 TEDF Kernelerweiterungen

6.3.1 RTEDF Modul

Das RTEDF Modul im Kernel stellt die Schnittstelle zwischen den transaktionsverwaltenden Klassen (*Cyg_Message*, *Cyg_Mbox_TEDF*) und der HAL zum RTEDF Coprozessor dar.

Zu den Aufgaben gehört die Verwaltung von Start– und Endeereignissen der Transaktionen, die Deadlineverlängerung im Überlastfall und die Kommunikation mit dem Coprozessor. Die Einbettung des RTEDF Moduls in der Betriebssystemstruktur ist in Abbildung 6.4 dargestellt. Die Schnittstellen des RTEDF Moduls zur RTEDF Kommunikation sind in Tabelle 6.5 definiert.

Syntax:	`void` **RTEDF_Modul::isr**();
Beschreibung:	Interrupt Service Routine, die bei einem Interrupt vom Coprozessor aktiviert wird. Es werden die vom Coprozessor vorgegebenen Instruktionen durchgeführt.
Syntax:	`void` **RTEDF_Modul::idle_time** () ;
Beschreibung:	Hiermit wird dem Coprozessor mitgeteilt, dass eine Idle Time stattgefunden hat. Diese Methode wird vom TEDF Scheduler aufgerufen.
Syntax:	`void` **RTEDF_Modul::end_time** (*cyg_uint32 ID*);
Beschreibung:	Diese Methode teilt dem Coprozessor das Ende einer Transaktionsinstanz mit.
Syntax:	`void` **RTEDF_Modul::init_qos** (*Cyg_Message *msg*, *cyg_handle_t mbox*);
Beschreibung:	Bei neuen Ereignissen, die *QoS–*/*Soft–* Transaktionsinstanzen initiieren, wird mit dieser Methode die Analyse durch den Coprozessor angestoßen.

Tabelle 6.5: Kernel API des RTEDF Moduls

[4)] Es sei angemerkt, dass es sich bei der Darstellung um ein Pseudo C++ Code handelt und dass wesentliche Teile des Codes der Übersichtlichkeit halber nicht dargestellt sind.

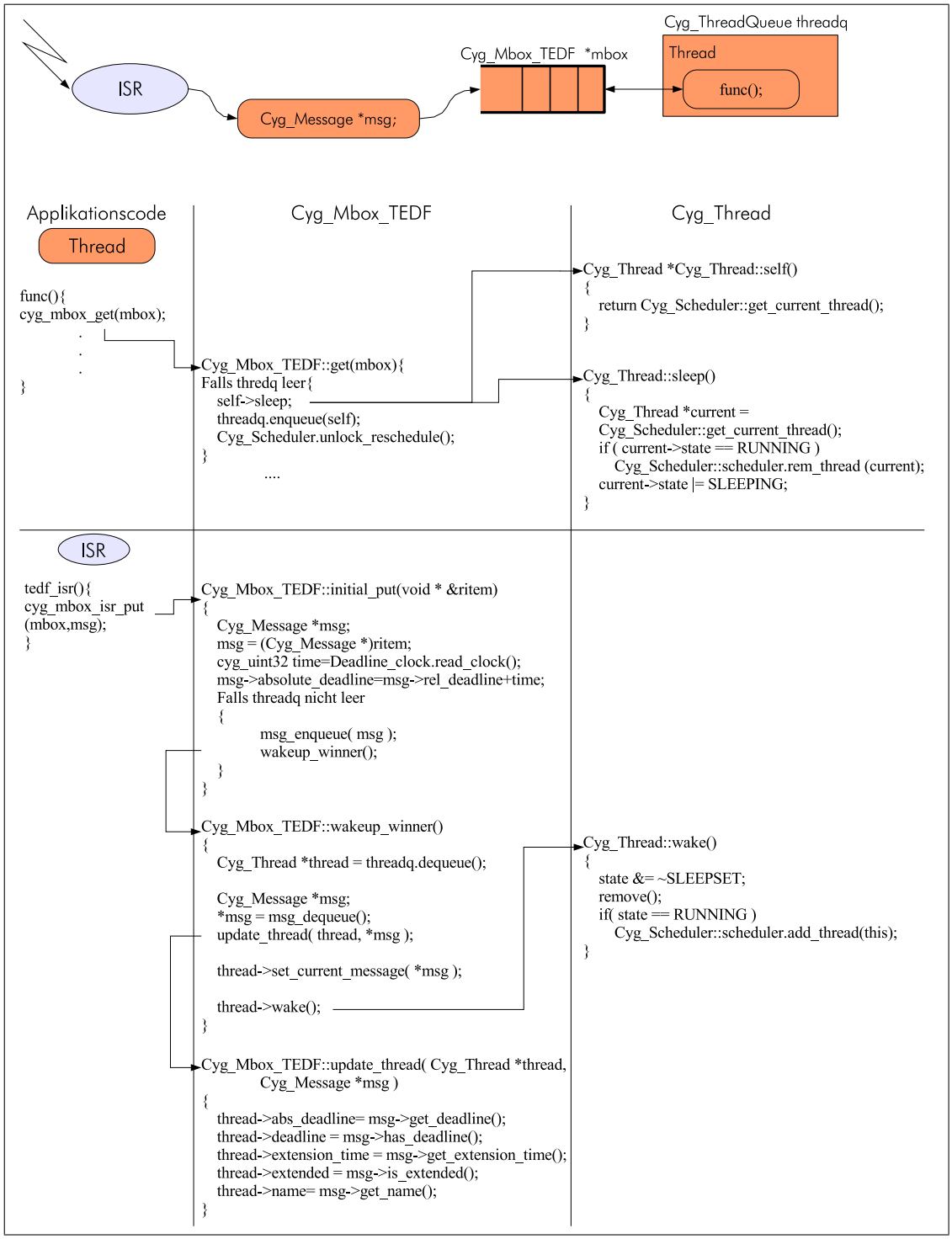


Abbildung 6.3: Kontrollfluss für ein *cyg_mbox_get* und ein *cyg_mbox_isr_put* Aufruf

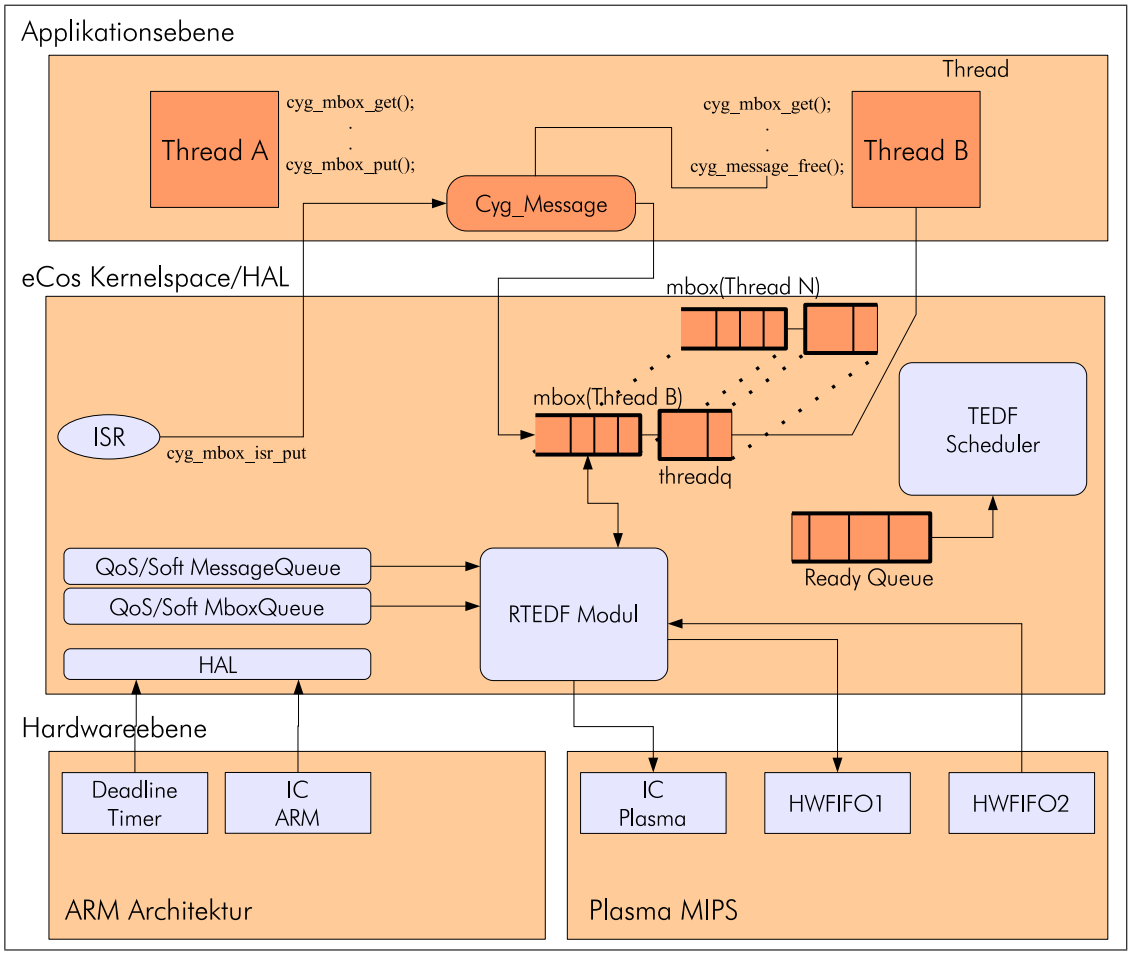


Abbildung 6.4: Strukturbild der Transaktionsverwaltung im TEDF Laufzeitsystem

6.3.1.1 TEDF zu RTEDF Interface

Für die effiziente Kommunikation zwischen dem RTEDF Modul und dem RTEDF Coprozessor ist ein einfaches Protokoll notwendig. Im Folgenden werden die Funktionen betrachtet, die von Ereignissen im TEDF System ausgehen und zu einer Notifikation des RTEDF Systems führen.

Initiale Events Alle Startevents von Transaktionen müssen auf ein Überlastszenario hin überprüft werden. Dabei wird zwischen *Soft–/QoS–* und harten Transaktionen unterschieden. Abbildung 6.3 zeigt den Kontrollfluss für die Methode *cyg_mbox_isr_put*, die auf die Methode *initial_put* abgebildet wird. Der Unterschied zur reinen TEDF Version besteht darin, dass unterschieden werden muss, ob es eine harte oder eine *QoS–/Soft–*Transaktion ist. Für harte Transaktionen wird die Nachricht in die entsprechende Mailbox eingefügt.

Handelt es sich um eine *QoS–/Soft–*Transaktion, so wird die gesamte Nachricht dem RTEDF Modul als Parameter übergeben. Die Nachricht wird in einem temporären Nachrichtenspeicher abgelegt, bis vom Coprozessor eine Anweisung erhalten wird. Anschließend wird ein Interrupt

mit der Meldung einer neuen *QoS*–Transaktion generiert.

Das RTEDF Modul trägt sowohl die ID als auch die absolute Deadline in den *memory mapped* HWFIFO ein und löst einen Interrupt beim Coprozessor aus. Der Mehraufwand beschränkt sich auf drei zusätzliche Schreibzugriffe. In Abbildung 6.5 ist die verwendete Funktion dargestellt.

```
   Data Input : Cyg_Message msg;
   Data : Cyg_Thread *owner, Cyg_Threadqueue threadqueue;
   Variable(lokal) : cyg_uint32 current_time,cyg_uint32 difference;
 1 current_time=DeadlineClock.read();
 2 msg–>absolute_Deadline=msg–>rel_deadline + current_time;
 3 msg–>busy=true;
   /* Nachricht ist ab jetzt in Benutzung                          */
 4 if (msg ∈ H) then
 5     msg_enqueue(msg);
       /* Sortiertes Einfügen in die Mailbox                       */
 6     if (!(threadqueue.empty())) then
 7         wakeup_winner();
 8         difference=msg–>absolute_deadline – owner–>absolute_deadline;
 9         if (difference < 0) then
10             if (owner–>get_wait_info() !=0) then
11                 exchange_messages();
12             else
13                 dip_boost(difference);
14 else
15     RTEDF_Modul.init_qos(msg, this);
```

Abbildung 6.5 : Cyg_Mbox_TEDF::initial_put(Cyg_Message *msg)

End Events Das Ende einer Transaktion wird dem Coprozessor ebenfalls über diese Schnittstelle mitgeteilt. In diesem Fall hängt das Ende der Transaktion von dem Applikationsverhalten ab. Wird eine zuletzt konsumierte Nachricht nicht mehr weitergesendet, so endet die Transaktion an dieser Stelle. Dies wird dem Laufzeitsystem durch die API Methode *cyg_message_free* angezeigt. Diese Methode ruft wiederum die Methode *RTEDF_Modul::end_time* auf.

```
   Data Input : cyg_uint32 ID;
 1 HWFIFO1.write(ID);
 2 HWFIFO1.write(-1);
 3 IC_Plasma.raise_qos();
   /* Auslösen des Interrupts beim Coprozessor                     */
```

Abbildung 6.6 : RTEDF_Modul::end_time(*cyg_uint32 ID*)

Idle Time Für die Detektion des Umschaltzeitpunktes ist es notwendig, über die Idle Phasen informiert zu werden. Nach einer Idle Phase kann das System wieder in den Normalmodus ohne Überlastbehandlung umgeschaltet werden. Die Tatsache, dass während der Überlastanalyse des Coprozessor eine Idle Zeit auftritt, kann durch die vom RTEDF Modul gesendeten Daten nur sehr zeitaufwändig rekonstruiert werden, da für jedes Datum ein eigener Zeitstempel benötigt würde. Detektiert der Scheduler nur noch den Idle Thread in der RunQueue, so wird das RTEDF Modul darüber informiert.

```
   Data Input : Cyg_Message *msg, cyg_handle_t mbox;
1  HWFIFO1.write(msg–>id);
2  HWFIFO1.write(msg–>absolute_deadline);
3  QSMessageQueue[msg–>id]=msg;
4  QSMboxQueue[msg–>id]=mbox;
5  IC_Plasma.raise_qos(); /* Auslösen des Interrupts beim
       Coprozessor                                          */
```

Abbildung 6.7 : RTEDF_Modul::init_qos(Cyg_Message *msg, cyg_handle_t mbox)

6.3.1.2 RTEDF Ereignisse

Eine weitere Aufgabe des RTEDF Moduls besteht darin, Anforderungen des RTEDF Systems im TEDF Laufzeitsystem durchzuführen. Die Abarbeitung erfolgt durch einen Interrupt des Coprozessors und anschließendem Auslesen des HWFIFOs durch das RTEDF Modul.

6.3.2 Laufzeitmessungen

Jede Erweiterung bedeutet zur Laufzeit einen zusätzlichen Overhead, der sowohl für die Realisierbarkeit eines Konzeptes als auch für die Realzeitanalyse berücksichtigt werden muss. Die Bewertung erfolgt getrennt nach Datenstrukturen und Rechenzeit, und beinhaltet die für eine Bewertung relevanten Strukturen.

6.3.2.1 Messaufbau

Sämtliche Messungen wurden auf der in Abschnitt 6.4 beschriebenen Plattform durchgeführt. Um Messungen durchzuführen, wurden im Quellcode Assembly Schreibzugriffe über den Systembus in FPGA Register durchgeführt. Alle Schreibzugriffe auf die Messregister erfolgten ohne Einfluss des Datencaches und der Schreibpuffer. Der Inhalt dieser Register wurde extern mit einem Logic Analyzer aufgezeichnet und ausgewertet. Der Overhead der Messmethodik beläuft sich auf zwei Schreibzugriffe pro Messintervall.

Die Algorithmen sind teilweise in Assembly als auch in ANSI C erstellt worden. Als Compiler kommen auf eCos/ARM Seite der GCC Cross Compiler in Version 3.2.1 und auf MIPS/Plasma Seite in Version 2.96 jeweils ohne Optimierungsflags zum Einsatz.

```
   Data Input : Cyg_Message msg;
   Data : Cyg_Message *msg, Cyg_Threadqueue threadqueue, cyg_handle_t mbox;
   Variable(lokal) : struct data{char* command, cyg_uint32 id};
1  while (!(HWFIFO2.empty())) do
2      data=HWFIFO2.read();
3      switch (data.command) do
4          case INSERT_NORMAL
5              msg=QSMessageQueue[data.id];
6              mbox=QSMboxQueue[data.id];
7              cyg_mbox_put(mbox, msg);
8          case INSERT_EXTENDED
9              msg=QSMessageQueue[data.id];
10             mbox=QSMboxQueue[data.id];
11             msg–>extended=true;
12             cyg_mbox_put(mbox, msg);
13         case INSERT_FIRM
14             msg=QSMessageQueue[data.id];
15             mbox=QSMboxQueue[data.id];
16             msg–>firm=true;
17             cyg_mbox_put(mbox, msg);
18         case EXTEND_ALL
19             forall the (mbox ∈ Mboxlist) do mbox–>extend_all_qos();
20             scheduler–>extend_all_qos();
               /* Alle QoS--Threads werden zu Delta--Threads
                  erweitert                                    */
21         case SKIP
               /* Nachricht wird nicht weitergeleitet          */
22             msg–>busy=false;
```

Abbildung 6.8 : RTEDF_Modul::isr()

6.3.3 Datenstrukturen

Die Verwaltung der Nachrichtenmetadaten wird in den Nachrichten selber durchgeführt. Für die Datenstruktur *Cyg_Message* aus Tabelle 6.1 muss daher pro Nachricht ein zusätzlicher Bedarf von 60 Byte berücksichtigt werden.

6.3.4 Laufzeiten

In Kapitel 4, Tabelle 4.4 wurden die für eine Realzeitanalyse relevanten Ausführungszeiten bereits definiert. In diesem Abschnitt werden die Ausführungszeiten $C_{tedf_{event}}$, $C_{tedf_{insert}}$ und $C_{tedf_{extendall}}$ für diese Implementierungsarchitektur diskutiert.

Initiierung von *QoS–/Soft*–Transaktionen Die Funktionalität, die zur Ausführungszeit $C_{tedf_{event}}$ führt, ist in Abbildung 6.7, Zeile 1-5 dargestellt. Die maximale Ausführungszeit dieser Funktion beträgt

$$C_{tedf_{event}}(n) = n \cdot 1.12\mu s. \tag{6.1}$$

Verlängern der Deadlines aller *QoS–/Soft*–Transaktionen Die Umschaltung in den Überlastmodus erfordert zur Laufzeit die Aktivierung der *Delta*–Deadlines aller *QoS*– und *Soft*–Transaktionen. Dieser Eingriff erfordert die meiste Rechenzeit im TEDF Laufzeitsystem.

Für die Rechenzeitanforderungsanalyse wird von folgenden Voraussetzungen ausgegangen:

- Der Durchlauf aller Transaktionen erfolgt linear und besitzt somit eine Komplexität von $O(n)$. Die Zeit, die benötigt wird, um zu bestimmen, ob eine Transaktion verlängerbar ist und der darauf folgenden Verlängerung der Deadline, beträgt

$$C_{extend}(n) = n \cdot 0.36\mu s. \tag{6.2}$$

- Die maximale Anzahl der Nachrichten M_i, die sich gleichzeitig in einer Mailbox eines Threads i befinden können, sei bekannt.
- Jede Mailbox wird nach der Deadlineerweiterung als unsortiert angesehen.
- Die Sortierung der Mailboxen erfolgt mit dem *Insertion Sort* Algorithmus und besitzt eine Komplexität von $O(n^2)$. Die Funktion zur Berechnung der maximalen Ausführungszeit wird mit der Funktion

$$C_{insertion\_sort}(n) = (n-1) \cdot \frac{n}{2} \cdot C_{compare} \qquad \text{für} \quad n > 1 \tag{6.3}$$

in Abhängigkeit von der Anzahl der zu sortierenden Elemente n beschrieben. Die Größe $C_{compare}$ bezeichnet dabei die maximale Zeit, die für den Vergleich von zwei Einträgen in einer Liste benötigt wird.

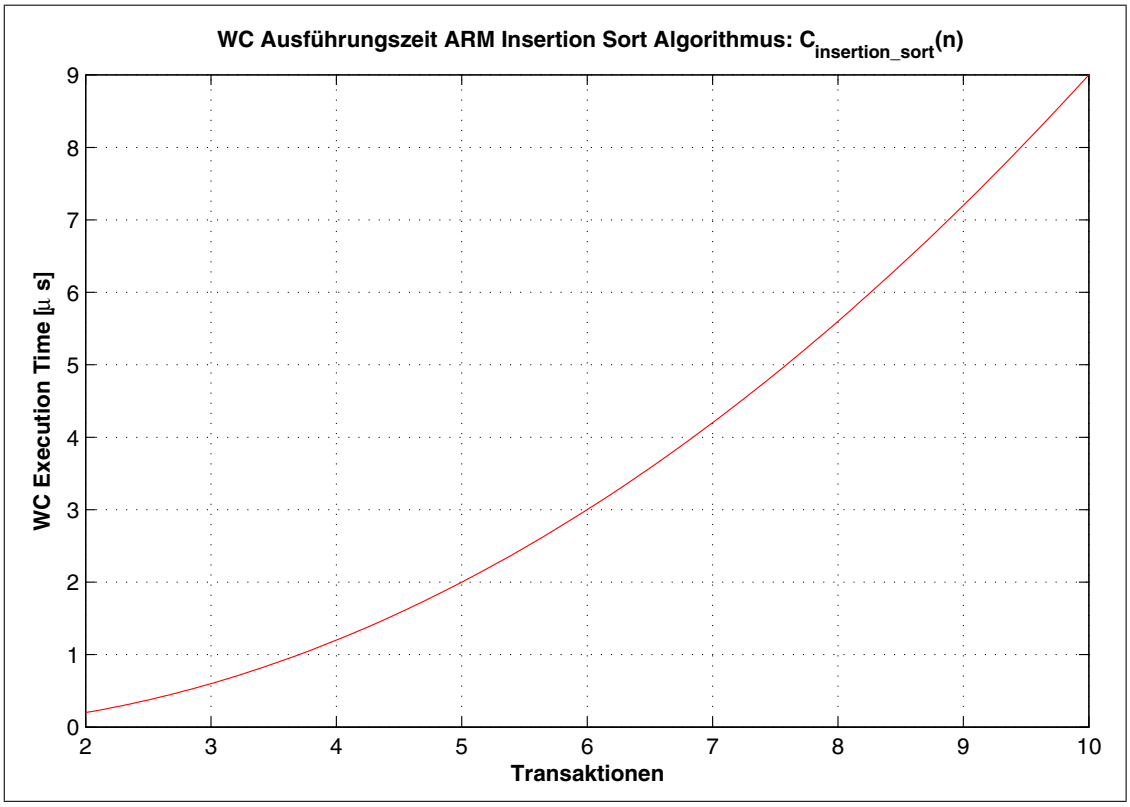


Abbildung 6.9: Quadratische Laufzeitkomplexität der Insertion Sort Routine aus Gleichung 6.3 auf einem ARM922T™ (166 MHz)

Der Wert für $C_{compare}$ beträgt

$$C_{compare} = 0.2\mu s. \tag{6.4}$$

Die Berechnung der maximalen Rechenzeit erfolgt nach folgendem Algorithmus:

1. Wähle die Mailbox i mit der maximalen Anzahl an enthaltenen Nachrichten $max(M_i \in \mathbb{M})$ aus.
2. Berechne die Rechenzeit $C_{tedf\_sort} = C_{tedf\_sort} + C_{insertion\_sort}(M_i)$.
3. Korrigiere alle M_j mit $j \neq i$, da eine Nachricht in M_i nicht gleichzeitig in einer anderen Mailbox sein kann.
4. Fahre mit Schritt 1 solange fort, bis alle Nachrichten berücksichtigt sind.

Somit ergibt sich für die maximale Ausführungszeit:

$$C_{tedf\_extendall}(n) = C_{tedf\_sort} + C_{extend}(n). \tag{6.5}$$

Aufgrund der quadratischen Komplexität ist die Berücksichtigung der maximalen Anzahl an Nachrichten pro Mailbox bei der Auslegung der Software Architektur von Bedeutung. Die zu berücksichtigende Laufzeit für dieses System ist in Abbildung 6.9 dargestellt.

Einfügen von Nachrichten in eine Mailbox Das Einsortieren der Nachrichten in die Mailbox erfolgt nach dem *Insertion Sort* Algorithmus. Trotz einer Komplexität von $O(n^2)$ ist die benötigte Rechenzeit für kleine n tragbar. Hinzu kommt, dass eine neue Nachricht immer in eine bereits sortierte Mailbox eingefügt wird und somit die dafür benötigte Komplexität linear ist. Die Zeit für das Einfügen von Nachrichten hängt von der maximalen Anzahl an Nachrichten in der Mailbox und der Zeit für den Vergleich von zwei Deadlines ab:

$$C_{tedf\_insert}(n) = n \cdot C_{compare}. \tag{6.6}$$

6.4 Hardwaregestützte Überlastbehandlung

In dieser Arbeit wurde die Überlastüberprüfung auf einen externen Coprozessor ausgelagert. In diesem Abschnitt soll daher kurz auf die Architektur, den Ressourcenbedarf und die Performance der realisierten Architektur eingegangen werden. Wie in Kapitel 2 in Abschnitt 2.4.1 bereits dargelegt wurde, bietet sich eine Lösung zur Implementierung der Überlastüberwachung mit einem *loosely coupled* Coprozessor an. Dieser Ansatz liegt darin begründet, dass der späteste Zeitpunkt, an dem eine Überlast im Laufzeitsystem noch verhindert werden kann nicht mit dem Zeitpunkt des Eintreffens neuer Transaktionsinstanzen zusammenfallen muss. Die Aufgabe besteht zum einen in der Zugangskontrolle von neuen Transaktionsinstanzen und zum anderen in der Verzögerung der Überlastüberprüfung bis zum primären Deadlineende.

Für die prototypische Realisierung der Arbeit wurde eine SOPC (*System on a Programmable Chip*) Architektur der Firma ALTERA® verwendet. Der Baustein beinhaltet einen Hard–Core ARM922T™ Prozessor und einen programmierbaren logischen Bereich (FPGA). Für die Realisierung des *loosely coupled* Coprozessors wurde ein *Open Source* Soft–Core Prozessor[5)] im FPGA realisiert.

Die Soft–Core Lösung ist ein MIPS R2000 kompatibler RISC Kern, der lizenzfrei als offener VHDL Quellcode verfügbar ist. Es wurden an dem Kern einige spezifische Anpassungen vorgenommen. Ein Vorteil dieses standardisierten Maschineninstruktionssatzes ist die freie Verfügbarkeit von Compilern.

Die einfache Corearchitektur zeichnet sich durch exakt vorhersagbare Instruktionsszyklen ohne Cache– und MMU–Einwirkungen aus. Dies ist ein wesentlicher Vorteil, da die RTEDF Überlastüberwachung selbst einem Realzeitnachweis unterzogen werden muss und dafür eine analysierbare Architektur notwendig ist. Im Anhang A sind die wesentlichen Kennzahlen zur Coprozessorimplementierung sowie ein Vergleich zu bestehenden kommerziell erhältlichen Soft–Core Prozessoren dargestellt.

Im Folgenden soll auf diejenigen Teile des RTEDF Algorithmus eingegangen werden, die den größten Teil der Rechenzeit beanspruchen.

[5)] frei unter http://www.Opencores.org verfügbar

6.4.1 Sortierung

Die Sortierung der Transaktionsdaten ist eine der zentralen Funktionen des RTEDF Algorithmus. Für eine Realzeitanalyse müssen die Transaktionsdaten nach der Deadline sortiert vorliegen. Folgende Anforderungen sind zu erfüllen:

- Zu jeder Deadline muss eine Ausführungszeit und der Typ der Transaktion abgelegt werden. Die Sortierung muss dabei indexbasiert erfolgen.
- Es soll ein von der Komplexität optimaler Algorithmus mit einer analysierbaren maximalen Rechenzeit verwendet werden.

Für die Sortierung wurde ein modifizierter Heapsortalgorithmus verwendet. Die verwendete Datenstruktur ist in Abbildung 6.10 dargestellt. Das Array *Index* enthält die Zeiger auf die einzelnen Transaktionsdaten und stellt die zentrale Heapstruktur dar. Bei einer sortierten Ausgabe der Indizes, die man für die Rechenzeitanforderungsfunktion benötigt, würde die Heapstruktur zerstört werden und müsste für jeden zusätzlichen Eintrag erneut aufgebaut werden.

Um diese rechenaufwendige Operation zu eliminieren, wird nur bei Bedarf eine Kopie des Heaps erstellt und dieser sortiert ausgegeben (*SortedCopied Index*). Ereignisse, wie *IDLE EVENT* und *END EVENT* führen nicht zu einer erneuten Realzeitanalyse. In der Realisierung ist eine Änderung der Daten von ihrer Neusortierung entkoppelt.

Die Ergebnisse der durchgeführten Optimierung sind in Abbildung 6.11 dargestellt. Für die Laufzeit wurden anhand von Messungen folgende Werte ermittelt:

Sortiertes Einfügen von Daten:	$C_{heap\_insert} = 4.6\mu s \cdot log_2(n)$
Kopieren des Heaps:	$C_{heap\_copy} = 0.7\mu s \cdot n$
Ausgeben der sortierten Indizes:	$C_{heap\_out} = 2.5\mu s \cdot n \cdot log_2(n).$

6.4.2 Laufzeiten und Datenstrukturen des RTEDF Algorithmus

Im Folgenden wird auf die Codestruktur des in Kapitel 4, Abschnitt 4.3.5, vorgestellten RTEDF Algorithmus eingegangen.

6.4.2.1 Datenstrukturen

Für die Datenstrukturen TMD und PZD werden insgesamt 20 Byte pro Transaktion benötigt. Die Rechenzeitanforderungsfunktion (TZD) und die Heapsortstrukturen benötigen weitere 16 Byte pro Transaktion.

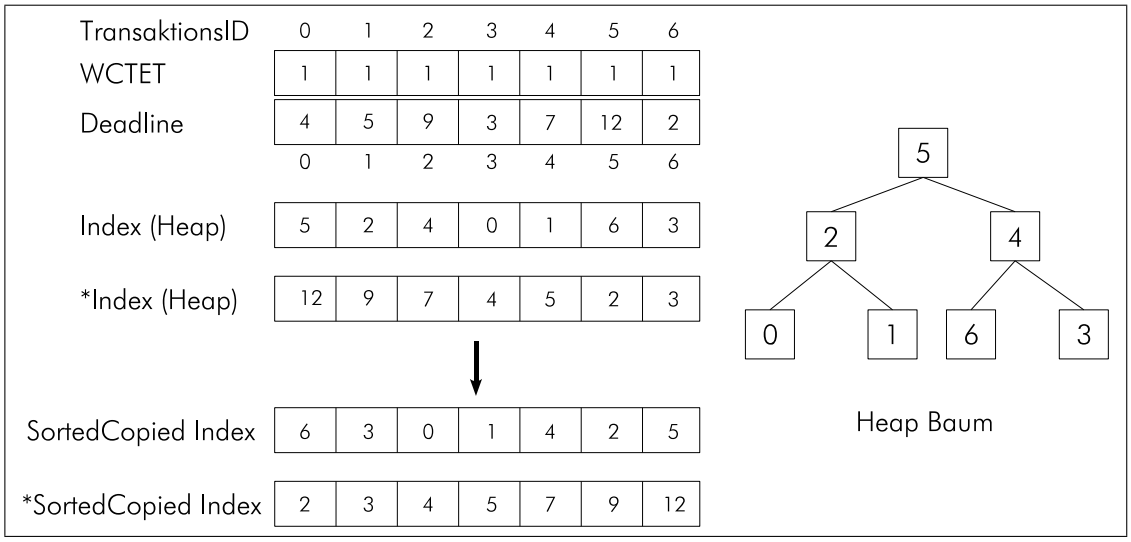


Abbildung 6.10: Beispiel zu Datenstruktur und Heapbaum des modifizierten Heapsort Algorithmus

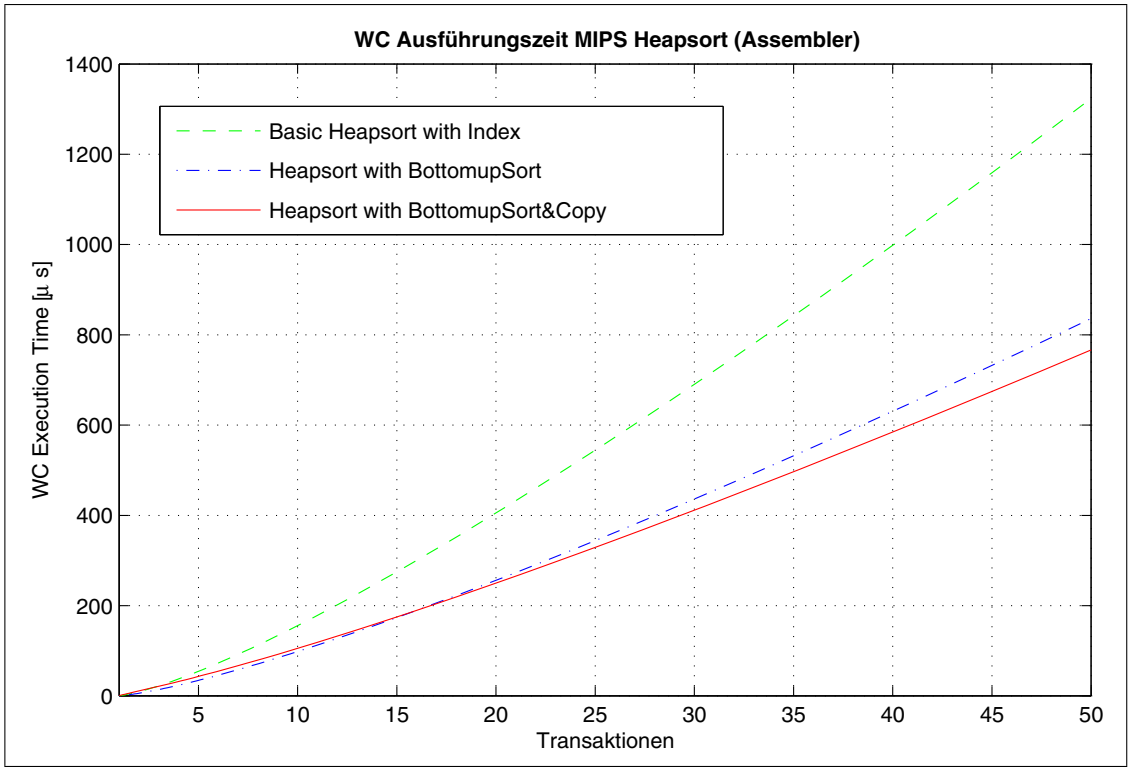


Abbildung 6.11: Ausführungszeiten der verschiedenen Heapsort Varianten auf dem Soft–Core MIPS Prozessor (16 MHz)

6.4.2.2 Laufzeiten

Zu den in Kapitel 4, Tabelle 4.4, definierten Größen $C_{rtedf\_par_{sort}}$, $C_{rtedf\_par_{rtanalysis}}$ und $C_{rtedf\_par_{main}}$ sind im Folgenden die maximalen Ausführungszeiten dargestellt.

Sortierung Die Sortierung betrifft das sortierte Einfügen der Elemente, das Kopieren des Heaps und das anschließende Ausgeben der sortierten Elemente für eine Realzeitanalyse. Die zu berücksichtigenden Zeiten ergeben sich aus

$$\begin{aligned} C_{rtedf\_par_{sort}}(n) &= C_{heap\_insert}(n) + C_{heap\_copy}(n) + C_{heap\_out}(n) \\ &= 4.6\mu s \cdot log_2(n) + 0.7\mu s \cdot n + 2.5\mu s \cdot n \cdot log_2(n). \end{aligned}$$

Realzeitanalyse Die Ausführungszeit der Realzeitanalyse ist linear und kann mit

$$C_{rtedf\_par_{rtanalysis}}(n) = 1.5\mu s \cdot n \tag{6.7}$$

angegeben werden.

RTEDF Main Kontrollfluss Die Ermittlung der maximalen Ausführungszeiten zum RTEDF Algorithmus aus Abbildung 4.5 basiert auf dem Worst Case Pfad im Algorithmus. Die

Aktion	Zeile
Einlesen aller TEDF Ereignisse aus dem HWFifo	1-4
Feststellen, welcher RTEDF Zustand vorliegt	5-6
Überlastsystemzustand	7-8
Aktivieren des RTEDF Moduls	25

Tabelle 6.6: WC Kontrollfluss im RTEDF Algorithmus

maximale Ausführungszeit $C_{rtedf\_par_{main}}$ ist:

$$C_{rtedf_{main}}(n) = n \cdot 5.8\mu s. \tag{6.8}$$

Somit kann für die Ausführungszeit des RTEDF Algorithmus auf dem Coprozessor folgende Funktion angenommen werden:

$$C_{rtedf}(n) = C_{rtedf_{main}}(n) + C_{rtedf_{sort}}(n) + C_{rtedf_{rtanalysis}}(n). \tag{6.9}$$

6.5 Anbindung an High–Level Entwurfswerkzeuge

Die Vorgehensweise bei der automatischen Abbildung von Softwaremodellen auf ein Laufzeitsystem ist bei vielen Werkzeugen ähnlich. Das Werkzeug bringt eine Reihe von Toolboxen und Service Bibliotheken mit, die dem Entwickler als Funktionsumfang zur Verfügung stehen. Für die Abbildung selber wird meistens ein minimalistischer Ansatz gewählt. Die Laufzeitsemantik wird von den Werkzeugherstellern durch eigene Funktionalität realisiert, um möglichst unabhängig vom Realzeitbetriebssytem zu bleiben und möglichst viele Architektur– und Betriebssystemkombinationen zu unterstützen. Von den Realzeitbetriebssystemdiensten werden daher nur ein periodischer Timerinterrupt und Semaphore zur Synchronisation verwendet. Eine stärkere Nutzung von Betriebssystemprimitiven würde jedoch zu einer effizienteren Abbildung führen.

In dieser Arbeit wurde eine prototypische Abbildung des UML–RT Laufzeitsystems aus dem Werkzeug IBM Rational Rose–RT auf die TEDF Laufzeitsemantik durchgeführt [6][8].

Die wesentlichen Entwurfsschritte erfolgen im CASE–Werkzeug in den Entwurfsphasen *Logical View* und *Component View*. Im *Logical View* steht die Kapselung der Struktur und des Verhaltens in *RTCapsules* im Vordergrund, während in dem *Component View* das Mapping der *RTCapsules* auf die physikalischen Threads durchgeführt wird.

Dabei kommunizieren Hauptstrukturen (*RTCapsule*), die auf einen Thread gemapped werden, über die TEDF Primitive (Mailbox) (Abbildung 6.12). Alle weiteren hierarchisch strukturierten *RTCapsules* kommunizieren weiterhin über die internen Strukturen.

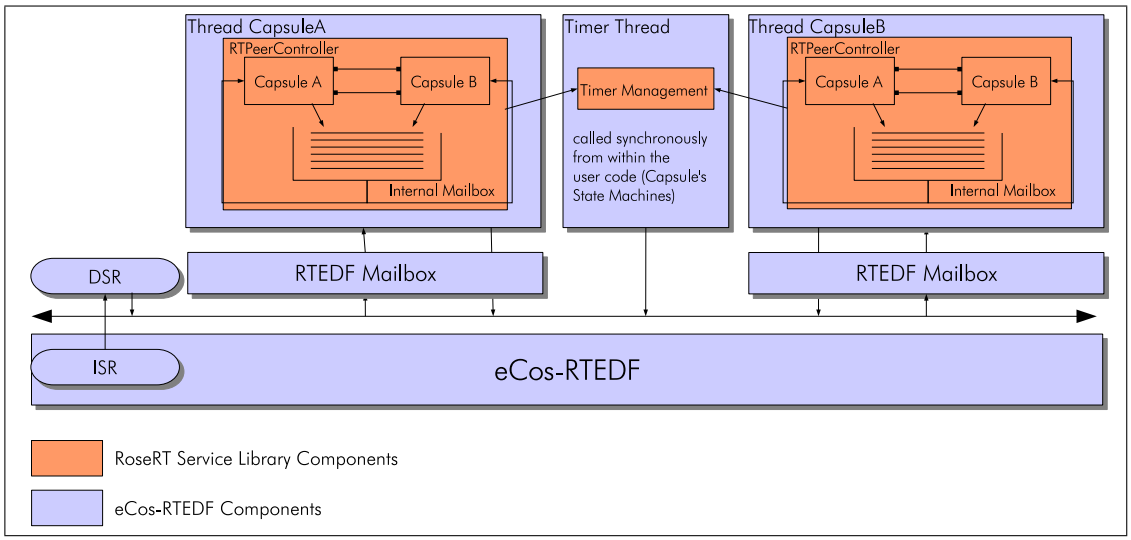


Abbildung 6.12: Abbildung des UML–RT Laufzeitsystems auf die TEDF Laufzeitsemantik

Durch die Abbildung entsteht auch eine minimale Änderung des UML–RT APIs. Asynchronen Nachrichten kann mit EDF keine Priorität mehr zugewiesen werden.

Das *Run–to–Completion* Paradigma bleibt durch die TEDF Abbildung erhalten.

6.6 RTEDF Entwurfsrichtlinien

Bei jeder Spezifikationssprache und -methodik gibt es Richtlinien für den Entwurf eines „*guten*“ Designs. Ohne diese Vorgaben und Leitfäden sind Entwurfswerkzeuge allein kein Garant dafür, dass ein erweiterbares und sinnvolles Design entsteht. In diesem Abschnitt soll daher auf einige Punkte eingegangen werden, die für eine effiziente Anwendung des RTEDF Schedulings notwendig sind.

6.6.1 Objektkomposition

In Abschnitt 3.4.1 zur Blockierungszeitenanalyse wurde gezeigt, dass semantisch unterschiedlicher Funktionscode, der in einem Thread gekapselt ist und von Transaktionen mit unterschiedlicher Wichtigkeit genutzt wird, zu einer pessimistischen Realzeitanalyse führen kann. Mitunter wird man einen sehr hohen *Delta* Wert festlegen müssen, um unterhalb der Rechenzeitanforderungsfunktionsgeraden $D(I)$ bleiben zu können. An dieser Stelle wäre es Aufgabe des Transaktionsparsers (Abbildung 6.1), dieses Problem zu detektieren und dem Anwender die kritischen Threads zur Umstrukturierung vorzuschlagen.

6.6.2 Ereignissensitivität

Selbst wenn die Kombination von mehreren Funktionen in einem Thread für den Entwurf Vorteile bietet, so ist dies in Betrachtung mit den entstehenden Laufzeiten in den Mailboxen abzuwägen. Aufgrund der quadratischen Komplexität ist der Rechenzeitbedarf für die Verwaltung von mehreren Nachrichten durch eine Mailbox zu berücksichtigen.

6.7 Gesamtbetrachtung der Systemdimensionierung

In den vorangegangenen Abschnitten wurden sowohl die benötigten Datenstrukturgrößen als auch die auftretenden Ausführungszeiten dargestellt. In diesem Abschnitt sollen die Werte zu den für die parallele Ausführung aufgestellten Gleichungen 4.20 und 4.21 zu einem Gesamtbild zusammengefügt werden, so dass generell gültige Aussagen zur Systemdimensionierung abgeleitet werden können.

In der prototypisch implementierten Realisierung wurde ein Hauptprozessor ARM922T™ mit 166 MHz Taktfrequenz für das TEDF Laufzeitsystem verwendet. Der Soft–Core Coprozessor wurde mit 16 MHz auf ein zehntel der Prozessorleistung des Hauptprozessor dimensioniert. Die maximale Ausführungszeit, die bei der Ausführung des RTEDF Algorithmus auf dem Coprozessor benötigt wird, ist in Abbildung 6.13 dargestellt.

Für die maximalen Laufzeiten im TEDF Laufzeitsystem müssen Annahmen bezüglich der Nachrichtenverteilung in den Mailboxen gemacht werden. Geht man davon aus, dass die Verteilung aller Nachrichten gleichverteilt ist, so ergibt sich für die realisierte Architektur die

Ausführungszeitenkurve aus Abbildung 6.14. Der Unterschied zwischen beiden Funktionen besteht in der maximalen Anzahl an angenommenen Nachrichten pro Mailbox.

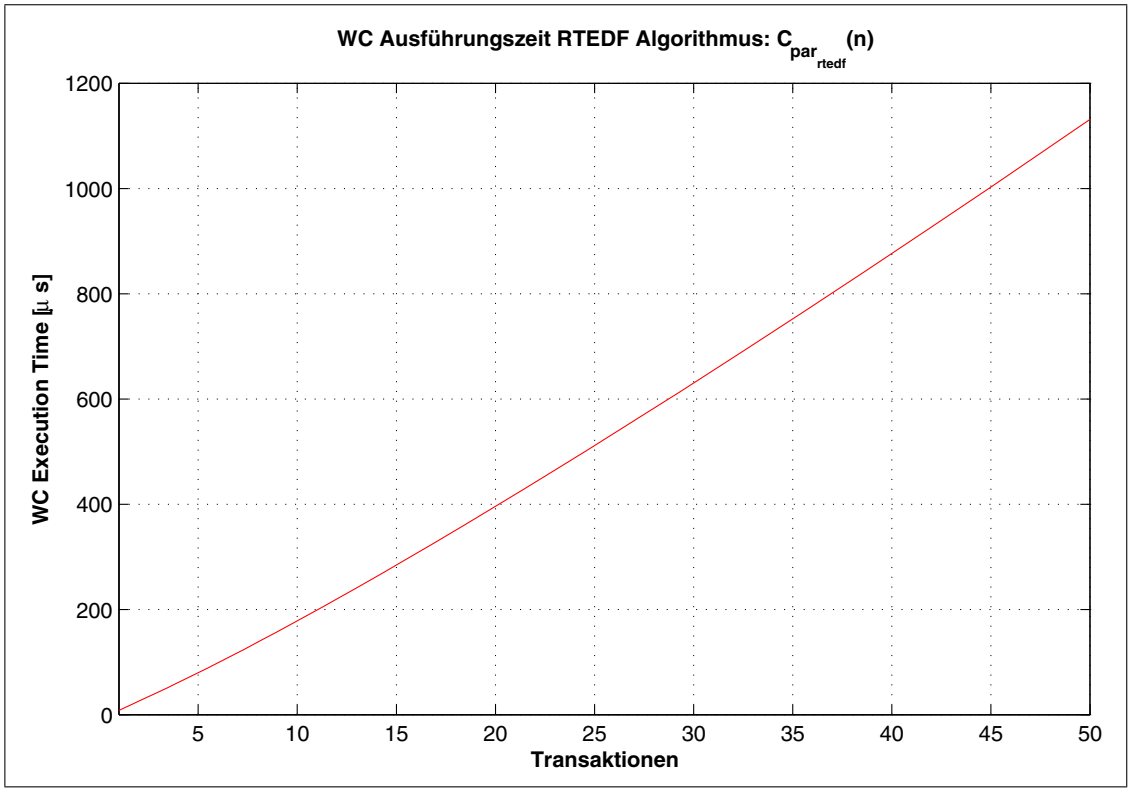


Abbildung 6.13: Maximale Ausführungszeit einer RTEDF Algorithmus Berechnung auf dem Soft–Core Coprozessor

Der zusätzliche Rechenzeitbedarf für $C_{par_{tedf}}$ tritt nur dann auf dem Hauptprozessor in Erscheinung, wenn aufgrund einer nicht transienten Überlast eingegriffen werden muss. Im Unterlastfall sind diese Ausführungszeiten nicht vorhanden. Die Überlastüberwachung findet auf dem Coprozessor statt, so dass dafür keine Rechenzeit auf dem Hauptprozessor zu berücksichtigen ist. Für einen Realzeitnachweis müssen diese Zeiten wie in Abbildung 4.14 dargestellt, berücksichtigt werden.

Der Einfluss des logarithmischen Anteils des Heapsort Algorithmus ist erkennbar, jedoch durch gewichtigere lineare Anteile überlagert.

So ergibt sich bei einer verzehnfachten CPU Leistung eine Reduktion von 20% gegenüber einem identisch konditionierten linearen Algorithmus.

Geht man also bei zukünftigen leistungsfähigen eingebetteten Systemen von der zehnfachen Leistung aus, so wären bei einer Einhaltung einer Mindestrate von $1ms$ für *QoS–*/*Soft*–Transaktionen, mit dem selben Leistungsverhältnis zwischen Hauptprozessor und Coprozessor, bis zu 200 Transaktionen verarbeitbar.

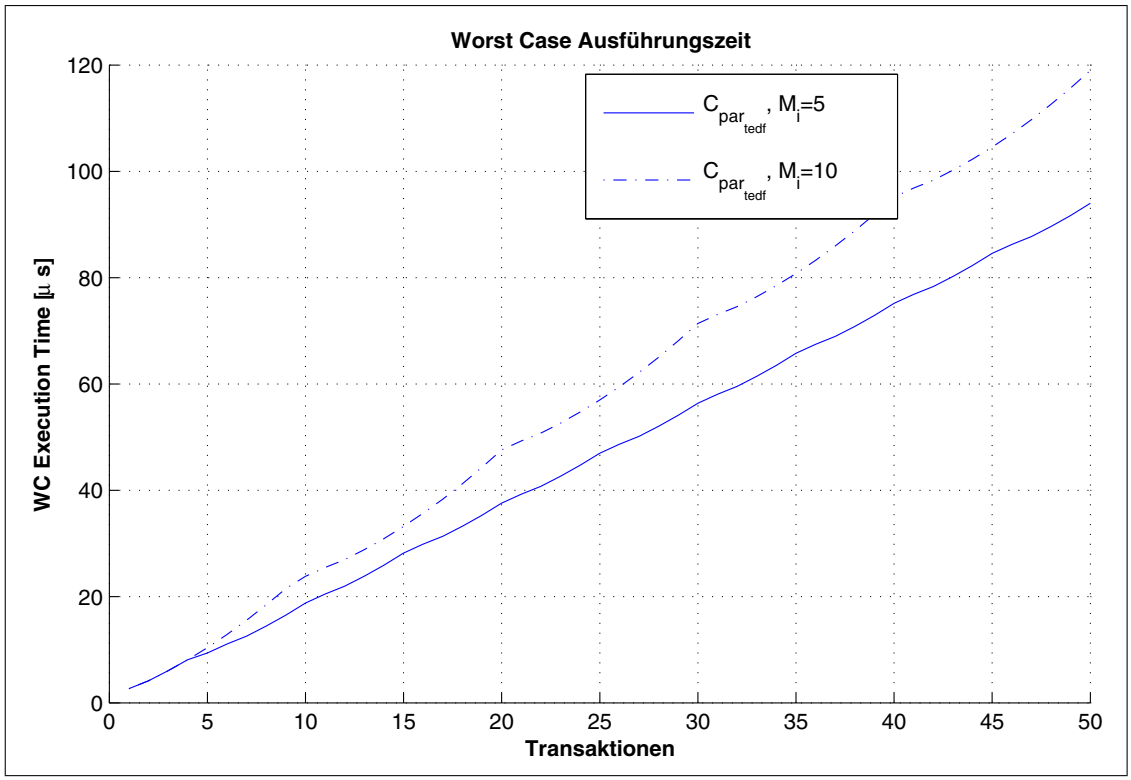


Abbildung 6.14: Maximale Ausführungszeit der Funktion $C_{par_{tedf}}(n)$ für das TEDF Laufzeitsystem auf einem ARM922T™ Prozessor (166 MHz)

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein neues Laufzeitsystem vorgestellt, das eine robuste und transaktionsbasierte Erweiterung (RTEDF) zum EDF Schedulingverfahren darstellt. Die Kernpunkte dieser Erweiterung können wie folgt zusammengefasst werden.

- Es wird eine effiziente Berücksichtigung von Präzedenzrelationen in Softwaremodellen sowohl für das Nachweisverfahren als auch zur Laufzeit (TEDF) ermöglicht.
- Die RTEDF Erweiterung zum TEDF Laufzeitsystem realisiert eine Integration und zeitliche Koexistenz von harten, *QoS*– und *Soft*–Transaktionen. Dabei wird zur Laufzeit ein robustes und a priori spezifiziertes Verhalten im Überlastfall realisiert.
- Die Integration des Laufzeitsystems in ein bestehendes Entwurfsframework wird durch einen parallelen Meta–Entwurfsfluss zur Extraktion und Implementierung parafunktionaler Anforderungen (*QoS*) realisiert.
- Die Auslagerung der Überlastüberwachung auf eine parallele Ausführungseinheit bietet eine effiziente und zum Hauptsystem orthogonale Architektur zur Implementierung des RTEDF Algorithmus.

Die effiziente Berücksichtigung von semantischen Präzedenzrelationen im Laufzeitsystem besteht in der dynamischen Weiterleitung der absoluten Deadline während der Abarbeitung einer Transaktion. Das vorgestellte TEDF Verfahren sorgt dafür, dass abhängige Tasks in einer Transaktion immer die optimale Priorität (Deadline) erhalten. Die transaktionsbasierte Laufzeitsemantik sorgt beim Nachweisverfahren für einen optimistischen Realzeitnachweis und eine zur Laufzeit spezifikationsgerechte Berücksichtigung der dynamischen Priorität.

Die Unterstützung von zeitlichen Anforderungen für harte, *QoS*– und *Soft*– Transaktionen gleichermaßen, wird durch den RTEDF Algorithmus bereitgestellt. Im Unterlastfall werden die zur Laufzeit entstehenden Slackzeiten (durch Variation von Ausführungszeiten, nicht auftretenden Blockierungszeiten und Variation von Ereignisabständen) zur Erzielung einer hohen *Qualität* von *QoS*–/*Soft*–Transaktionen genutzt. Im Überlastfall hingegen wird für diese Taskklassen eine vorab definierte Reduktion der Qualität mit *QoS*–Pattern angewendet, so dass ein robustes Verhalten gewährleistet ist. Die Eignung dieses Verfahrens konnte mit den Simulationsergebnissen für ein Beispieltransaktionsset aus Kapitel 5 gezeigt werden.

Der RTEDF Ansatz kombiniert die online Analyse aus den *Best–Effort* Ansätzen mit einem offline Nachweis für *QoS*–Pattern. Es werden definierte Pattern spezifiziert, die mit einem Nachweisverfahren zur Laufzeit garantiert werden können.

Im Unterlastfall wird durch eine verzögerte Umschaltung in den Überlastmodus die Nutzung von Slackzeiten realisiert. Das führt zu einem EDF Verhalten und somit einem optimalen Scheduling in der Unterlastphase. Der Rechenaufwand für die online Analyse wird auf die Sortierung

und die Verwaltung der Metadaten reduziert, da die offline Analyse bereits einen hinreichenden Nachweis zur Verfügung stellt.

Der Entwurf von RTEDF–basierten Systemen teilt sich in zwei Bereiche auf.

Der Hauptteil des Entwurfsprozesses entfällt auf die Extraktion der während der Modellierung und Spezifikation entstehenden Metadaten (Transaktionspfade, Deadlines, Ressourcennutzung). Dieser Teil kann größtenteils werkzeuggestützt realisiert werden, so dass dem Entwickler kein zusätzlicher Entwurfsaufwand entsteht. Der wesentlich kleinere Teil entfällt auf die leicht modifizierte Modellierungstechnik, die jedoch ohne weiteres mit dem Einsatz von Entwurfswerkzeugen durch Werkzeugbibliotheken abstrahiert werden kann.

Bei heutigen Entwurfswerkzeugen ist die Bereitstellung eines parallelen Meta–Entwurfsflusses durch offene Schnittstellen und APIs gegeben. Die Extraktion der Metadaten sowie die beispielhafte Abbildung von *Rational Rose–RT* Modellen auf eine TEDF Laufzeitsemantik konnte prototypisch validiert werden.

Des Weiteren wurde die Auslagerung der Überlastüberwachung (RTEDF) auf einen *loosely coupled* Coprozessor untersucht. Mit der hardwaretechnischen Trennung wird eine Entkopplung des Meta–Entwurfsflusses vom funktionalen Entwurf ermöglicht.

Das vorgestellte Verfahren zeigt sowohl bei Variation der Ausführungszeiten als auch bei Variation der Ereignissströme ein robustes Verhalten. Es eignet sich daher sowohl für ereignisgesteuerte Systeme mit Ereignisintervallen, die nicht streng periodisch verlaufen, als auch für Softwaremodelle, die starke Variationen der Ausführungszeiten aufweisen.

Das vorgestellte Laufzeitsystem kann unabhängig vom Aktivierungsmodell des einbettenden Systems eingesetzt werden. Eine Anbindung an zeitgesteuerte Peripheriebausteine ist ebenso möglich wie eine Ansteuerung durch eventbasierte Prozesse. Dieses erfordert keine Änderung der TEDF–basierten Softwarearchitektur.

Der vorgestellte Ansatz zeigt, dass durch eine optimierte Aufteilung von online und offline Analyse, realistische Laufzeiten für den Einsatz erweiterter dynamischer Schedulingverfahren bei Realzeitsystemen möglich sind.

Ausblick

Basierend auf den vorgestellten Ergebnissen sind weitere Einsatzmöglichkeiten und Optimierungen denkbar:

- In der Überlastphase wird streng nach spezifizierten *QoS*–Pattern verfahren. Die Eigenschaft, dass bei der Anwesenheit von *Firm*–Transaktionen wieder in den Überlastmodus geschaltet wird, könnte mit zusätzlichem Rechenaufwand minimiert werden. Dazu müßte bei jeder Rechenzeitanforderungsanalyse die aktuell verwendete Rechenzeit ermittelt und als weiteres Maß für eine Überlastumschaltung verwendet werden. Auch eine Kombination mit server–basierten Verfahren wäre in dieser Phase denkbar.

- Steht kein zusätzlicher Coprozessor zur Verfügung, so ist eine Realisierung des RTEDF Algorithmus auf dem Applikationsprozessor denkbar, wenn eine deterministische Ausführungszeitanalyse für den Algorithmus selbst garantiert werden kann. Zusätzlich muss für eine korrekte Konfiguration des RTEDF Algorithmus innerhalb des Hauptbetriebssystems gesorgt werden. Aufgrund der geringen Codegröße von ca. 5.5 kByte des RTEDF Algorithmus (coprozessorseitig) ist eine effiziente Realisierung durch *Cache Locking* Mechanismen möglich.
- Auch die Kombination des RTEDF Ansatzes mit Verfahren, die einen Eingriff in das Softwaremodell erfordern, wie z. B. das *Imprecise Computation Model*, sind denkbar, wenn eine zusätzliche Rechenzeitbedarfssteuerung notwendig wird.

Durch den domänenübergreifenden Einsatz eingebetteter Systeme, ergeben sich in Zukunft komplexe und heterogene Anforderungen an eingebettete Laufzeitsysteme.

Ein Beispiel für diese Entwicklung stellt der Einsatz von homogenen und heterogenen Multiprozessorarchitekturen im eingebetteten Systembereich dar. Diese Architekturen, die auch um anwendungsspezifische Coprozessoren erweitert sind, benötigen effiziente, optimale, parametrierbare, vorhersagbare und analysierbare Laufzeitsysteme, die unterschiedliche Optimierungsstrategien unterstützen.

Im Hinblick auf diese Entwicklung, soll mit dieser Arbeit ein Beitrag zu den evolvierenden Anforderungen an eingebettete Laufzeitsysteme geleistet werden.

A Systemarchitektur

A.1 Struktur

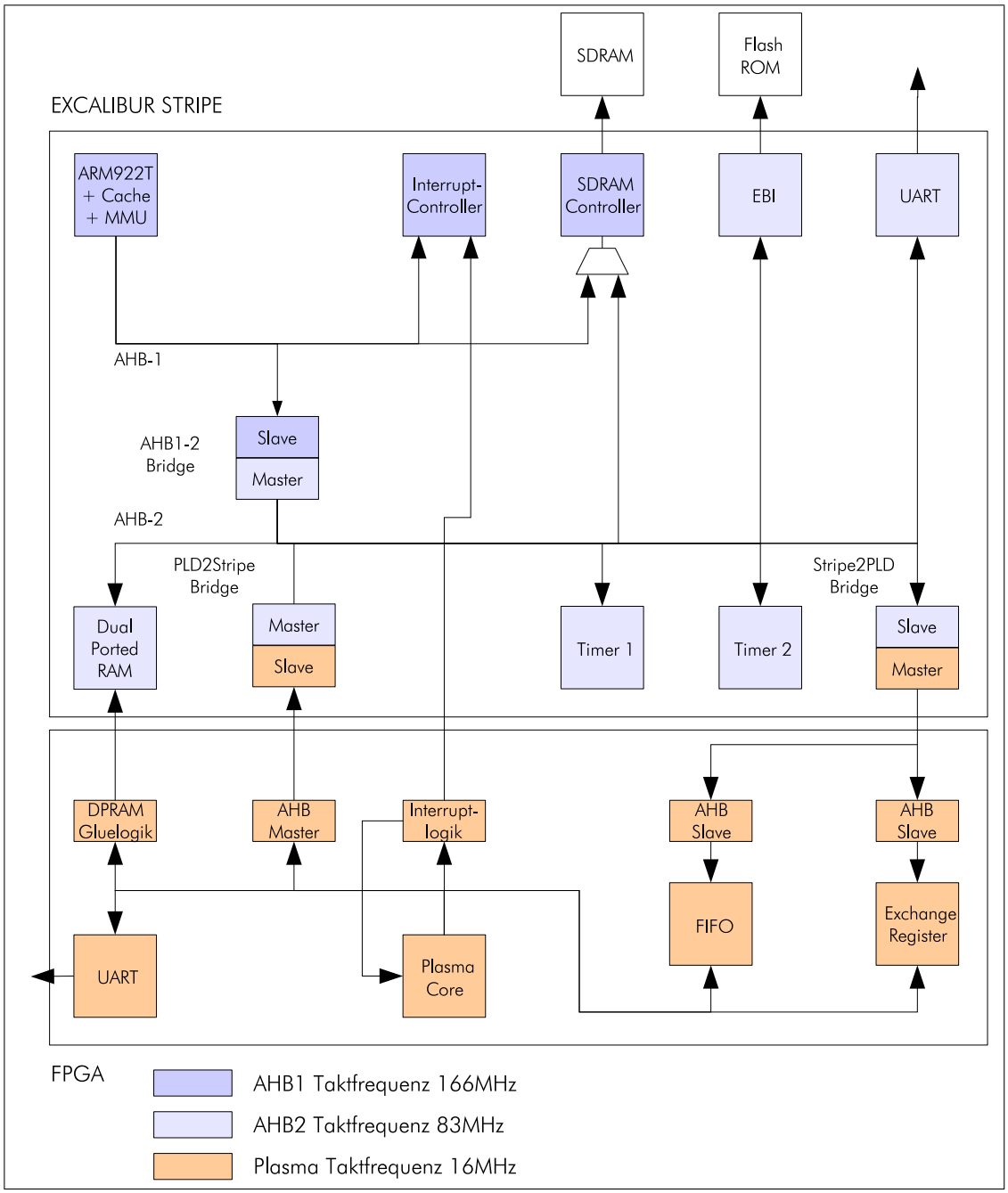


Abbildung A.1: Struktur der Coprozessorrealisierung im FPGA (EPXA10)

A.2 Synthesedaten

Gesamtes Design

	Plasma Soft–Core mit Peripherie
	MIPS–Core Excalibur–Stripe DP–RAM Schnittstelle Exchange Register AHB–Master/Slave FIFO
Logikzellen	4310
Memory Bits	4096
PLLs	1
FPGA	EPXA10

Tabelle A.1: Ressourcenverbrauch des gesamten Designs

Einzelkomponenten

	Plasma Soft–Core	**UART**	**AHB Slave**	**AHB Master**	**Exchange Register**	**FIFO**
Logikzellen	3176	467	53	41	982	27
Memory Bits	2048	576	0	0	0	2048
FPGA	APEX20KE	APEX20KE	APEX20KE	APEX20KE	APEX20KE	APEX20KE

Tabelle A.2: Ressourcenverbrauch einzelner Komponenten

Vergleich von Soft–Core Prozessoren

	Plasma Soft–Core	**NIOS**[1)]	**LEON**[1)]
Logikzellen	3176	3789	9345
Memory Bits	2048	142336	89088
FPGA	APEX20KE	Cyclone	Cyclone

Tabelle A.3: Vergleich des Ressourcenbedarfs von *State–of–the–Practice* Soft–Core Prozessoren

[1)] Quelle http://www.mdforster.pwp.blueyonder.co.uk/LeonCyclone.html

FPGA Bausteinkennzahlen

	EPXA10 (EPXA10F1020C2)	**EPXA1** (EPXA1F672C2)	**APEX20KE** (EP20K100EFC324-1)	**Cyclone** (EP1C20F400C7)
Logikzellen	38400	4160	4160	20060
Memory Bits	327680	53248	53248	294912
PLLs	4	2	2	2
Pins	715	327	327	301

Tabelle A.4: Kennzahlen der vewendeten und referenzierten FPGA Bausteine

Literaturverzeichnis

Allgemeine Literatur

[1] BROY, PROF. DR. MANFRED, DR. MICHAEL VON DER BEECK und INGOLF KRÜGER: *SOFTBED: Problemanalyse für ein Großverbundprojekt Systemtechnik Automobil – Software für eingebettete Systeme*. Technischer Bericht, Technische Universität München, Insitut für Informatik, http://www.bmbf.de/pub/softbed.pdf, 1998.

[2] GAJSKI, DANIEL D., NIKIL D. DUTT, ALLEN C-H WU und STEVE Y-L LIN: *High–Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[3] SEILER, PETRA, DIRK HOLTMANNSPÖTTER und ULRICH ALBERTSHAUSER: *Internationale Technologieprognosen im Vergleich*. Technischer Bericht ISSN–1436-5928, VDI Technologiezentrum GmbH im Auftrag des BMBF, 2004.

Projektliteratur

[4] BARBARINO, STEFAN: *Automatische Laufzeitanalyse am Beispiel eines Motorsteuergerätes*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, BMW AG München, November 2003.

[5] CASTRO BONFIM, CLEMENTE DE: *Flexible Multisensor Fusion*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, BMW AG München, November 2004.

[6] FENG, BO: *Anbindung von Rational Rose Real–Time zur target–spezifischen Codegenerierung an ein ARM Target mit Realzeitbetriebssystem*. Bachelorarbeit, Lehrstuhl für Realzeit–Computersysteme, Januar 2003.

[7] FENG, BO: *Erstellung einer Host–Target Verbindung für CTC*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Knorr–Bremse Systeme für Schienenfahrzeuge GmbH, Februar 2004.

[8] JIANG, HONGKUN: *Automatische Generierung von Transaktionsgraphen aus Rational Rose–RT Modellen*. Interdisziplinäres Projekt, Lehrstuhl für Realzeit–Computersysteme, März 2004.

[9] KESSLER, STEFAN: *Automatische Konfigurations- und Initialisierungscodegenerierung für den Mikrocontroller MSP430 von Texas Instruments*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, European Southern Observatory, Juli 2002.

[10] LICCARDI, MAIER-KOMOR, OSWALD, ELKOTOB und FÄRBER: *A Meta-Modeling Concept for Embedded RT-Systems Design*. 14th Euromicro Conference on Real-Time Systems – Work in Progress Session, 2002.

[11] LICCARDI, BENITO: *A Framework for Model–based QoS–aware Development of UML–RT Systems*. In: *2nd Workshop on Model-Driven Embedded Systems (MoDES '04)*, Toronto, Canada, May 2004. Workshop at the IEEE Real–Time and Embedded Technology and Applications Symposium (RTAS 2004).

[12] LICCARDI, BENITO und GEORG FÄRBER: *Template-basierte Entwicklung von eingebetteten Systemen für Produkte mittlerer Seriengröße*. atp – Automatisierungstechnische Praxis, Seiten 26–36, jun 2000.

[13] LICCARDI, BENITO, THOMAS MAIER-KOMOR und HANS OSWALD: *Meta-Modell basierte Architektur Templates für die High-Level Exploration Eingebetteter Realzeitsysteme*. In: *8. Fachtagung Entwurf komplexer Automatisierungssysteme (EKA)*, 2003.

[14] LOHN, HOLGER: *Entwicklung und Implementierung einer Scheduling Management Unit (SMU) für SoPC Systeme*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, August 2004.

[15] OBERMAIER, MAXIMILIAN FRANZ: *Entwicklung und Implementierung eines transaktionsbasierten EDF Schedulers für mixed Hard– und Soft Realzeitsysteme*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Oktober 2003.

[16] PATSIOPOULOS, NIKOLAS: *Development and Implementation of a Realtime Embedded System on the Basis of a Softcore Microcontroller in a FPGA*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Rohde&Schwarz GmbH, Juli 2002.

[17] SCHOLZ, RAINER und GREGOR JAKOB: *Implementierung einer domänenspezifischen Sprache zur Spezifikation und automatisierten Retargierung von hardwarenahen Software–Treibern*. Diplomarbeit, Januar 2003.

[18] SCHÄTZ, BERNHARD, MICHAEL FAHRMAIR, MICHAEL VON DER BEECK, PETER JACK, HANS KESPOHL, ALI KOÇ, BENITO LICCARDI, SANDRA SCHEERMESSER und ALBERT ZÜNDORF: *Entwickluung, Produktion und Service von Software für eingebettete Systeme in der Produktion*. VDMA Verlag GmbH, 2000.

Software Entwurfswerkzeuge

[19] *Systems Modeling Language (SysML) Specification*. http://www.omg.org/ cgi-bin/doc?ad/04-10-02, October 2004. version 0.85R1.

[20] ARTISAN SOFTWARE. http://www.artisansw.com/.

[21] ETAS GMBH & CO.KG: *ASCET–SD*. http://en.etasgroup.com/products/ascet_sd/.

[22] IBM: *Rational Rose Technical Developer*. http://www-306.ibm.com/ software/awdtools/suite/technical/.

[23] I-LOGIX: *Rhapsody*. http://www.ilogix.com/rhapsody/rhapsody.cfm.

[24] MATHWORKS, THE: *MATLAB/Simulink*. http://www.mathworks.com/.

[25] TELELOGIC: *Tau Suite*. http://www.telelogic.com/products/tau/index.cfm.

Scheduling — Laufzeitsysteme

[26] *eCos RTOS*. http://sources.redhat.com/ecos.

[27] *OSEK/VDX Time-Triggered Operating System, Version 1.0*. http://www.osek-vdx.org/mirror/ttos10.pdf.

[28] ABENI, LUCA und GIORGIO BUTTAZZO: *Integrating Multimedia Applications in Hard Real–Time Systems*. In: *Proceedings of 19th IEEE Real–Time Systems Symposium*, December 1998. Madrid, Spain.

[29] BAKER, T. P.: *A stack–based resource allocation policy for real–time processes*. In: *IEEE Real–Time Systems Symposium*, December 1990. Lake Buena Vista, Florida, USA.

[30] BAKER, T. P.: *Stack–Based Scheduling of Real–Time Processes*. Real–Time Systems Journal, (3), 1991.

[31] BARUAH, SANJOY K., JAYANT HARITSA und NITIN SHARMA: *On-line scheduling to maximize task completions*. In: *Proceedings of the 15th IEEE Real-Time Systems Symposium*, Seiten 228–236, 1994.

[32] BARUAH, SANJOY K. und JAYANT R. HARITSA: *ROBUST: A Hardware Solution to Real-Time Overload*. In: *Measurement and Modeling of Computer Systems*, Seiten 207–216, 1993.

[33] BARUAH, SANJOY K. und JAYANT R. HARITSA: *Scheduling for Overload in Real–Time Systems*. In: *IEEE Transactions on Computers*, Band 46. September 1997.

[34] BARUAH, SANJOY K., GILAD KOREN, D. MAO, BHUBANESWAR MISHRA, ARVIND RAGHUNATHAN, LOUIS E. ROSIER, DENNIS SHASHA und FUXING WANG: *On the Competitiveness of On-Line Real-Time Task Scheduling.* Real-Time Systems, 4(2):125–144, 1992.

[35] BATE, IAIN und ALAN BURNS: *An Integrated Approach to Scheduling in Safety–Critical Embedded Control Systems*. Real–Time Systems, 25:5–37, 2003.

[36] BURLESON, WAYNE, JASON KO, DOUGLAS NIEHAUS, KRITHI RAMAMRITHAM, JOHN A. STANKOVIC, GARY WALLACE und CHARLES C. WEEMS: *The Spring Scheduling Co-Processor: A Scheduling Accelerator.* In: *ICCD*, Seiten 140–144, 1993.

[37] BUTTAZZO, GIORGIO C.: *Rate Monotonic vs. EDF: Judgment Day*. Real–Time Systems, 29(1):5–26, January 2005.

[38] CACCAMO, MARCO, GIORGIO BUTTAZZO und LUI SHA: *Handling Execution Overruns in Hard Real–Time Control Systems*. In: *IEEE Transactions on Computers*, Band 51. Juli 2002.

[39] CACCAMO, MARCO, GIORGIO C. BUTTAZZO und DEEPU C. THOMAS: *Efficient Reclaiming in Reservation–Based Real–Time Systems with Variable Execution Times*. IEEE Transactions on Computers, 54(2), Februar 2005.

[40] CHETTO, HOUSSINE und MARYLINE CHETTO: *Some Results of the Earliest Deadline Scheduling Algorithm*. In: *IEEE Transactions on Software Engineering*, Band 15, Oktober 1989.

[41] CLARK, R.K.: *Scheduling Dependent Real–Time Activities*. Doktorarbeit, Carnegie Mellon University, 1990.

[42] DAMM, A., J. REISINGER, W. SCHWABL und H. KOPETZ: *The real–time Operating System of MARS*. ACM SIGOPS Operating Systems Review, 23:141–157, 1989. ISSN:0163-5980.

[43] DI NATALE, MARCO und JOHN A. STANKOVIC: *Dynamic End–to–End Guarantees in Distributed Real–Time Systems*. In: *Proceedings of the 15th IEEE Real–Time Systems Symposium*, Dezember 1994. San Juan, Puerto Rico.

[44] GARCIA, J. J. G. und M. G. HARBOUR: *Optimized priority assignment for tasks and messages in distributed hard real-time systems*. In: *WPDRTS '95: Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, Seite 124, Washington, DC, USA, 1995. IEEE Computer Society.

[45] GARDNER, MARK K. und JANE W.S. LIU: *Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload*. In: *Proceedings of the 11th Euromicro Conference on Real–Time Systems*, June 1999. York, England.

[46] GERBER, R. und S. HONG AN M. SAKSENA: *Guaranteeing End–to–End timing Constraints by Calibrating Intermediate Processes*. In: *Proceedings of the 15th IEEE Real–Time Systems Symposium*, Seiten 192–203, December 1994. San Juan, Puerto Rico.

[47] GONZÁLEZ HARBOUR, M. und J. C. PALENCIA: *Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities*. In: *Proceedings of the 24th IEEE International Real–Time Systems Symposium*, Dezember 2003. Cancun, Mexico.

[48] GRESSER, KLAUS: *Schedulability Analysis for Event–Driven Real–Time Systems*. Number 268 in *Fortschrittsberichte VDI, volume 10*. VDI–Verlag, Düsseldorf, 1993. Dissertation.

[49] GUTIERREZ, J. C. PALENCIA and MICHAEL GONZALEZ HARBOUR: *Schedulability Analysis for Tasks with Static and Dynamic Offsets*. In *19th Proc. of the IEEE Real–Time Systems Symposium (RTSS)*, Dezember 1998.

[50] HEITMEYER, CONSTANCE and DINO MANDRIOLI: *Formal Methods for Real–Time Computing*. John Wiley & Sons, 1996.

[51] HILDEBRANDT, JENS and DIRK TIMMERMANN: *An FPGA Based Scheduling Coprocessor for Dynamic Priority Scheduling in Hard Real–Time Systems*. In *FPL*, pages 777–780, 2000.

[52] KOBAYASHI, HIDENORI and NOBUYUKI YAMASAKI: *RT–Frontier:A Real–Time Operating System for Practical Imprecise Computation*. In *Proceedings of the 10th IEEE Real–Time and Embedded Technology and Applications Symposium*, Mai 2004. Toronto, Canada.

[53] KOLLOCH, THOMAS: *Scheduling with Message Deadlines for Hard Real–Time SDL Systems*. PhD thesis, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2002.

[54] KOREN, GILAD and DENNIS SHASHA: *D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems*. SIAM Jornal on Computing, 24(2):318–339, 1995.

[55] KOREN, GILAD and DENNIS SHASHA: *Skip-over: Algorithms and complexity for overloaded systems that allow skips*. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Dezember 1995. Pisa, Italien.

[56] LIPARI, GIUSEPPE and GIORGIO BUTTAZZO: *Schedulability Analysis of Periodic and Aperiodic Tasks with Resource Constraints*. Journal of System Architecture: EUROMICRO Journal, Special Issue on Real–Time Systems, 46(4):327–338, Februar 2000. ISSN:1383–7621.

[57] LI, PENG and BINOY RAVINDRAN: *Fast, Best–Effort Real–Time Scheduling Algorithms*. In *IEEE Transactions on Computers*, volume 53, pages 1159–1175. September 2004.

[58] LIU, C. L. and JAMES W. LAYLAND: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the ACM, 20(1):46–61, 1973.

[59] LIU, JANE W. S.: *Real–Time Systems*. Prentice–Hall International, Inc., 2000.

[60] LOCKE, C.D.: *Best–Effort Decision Making for Real–Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.

[61] MARZIARIO, LUCA, GIUSEPPE LIPARI, PATRICIA BALBASTRE and ALFONS CRESPO: *IRIS: A new reclaiming algorithm for server–based real-time systems*. In *Proceedings of the 10th IEEE Real–Time and Embedded Technology and Applications Symposium*, May 2004. Toronto, Canada.

[62] MESCHI, ANTONIO, MARCO DI NATALE and MARCO SPURI: *Earliest Deadline Message Scheduling with Limited Priority Inversion*. In *4th International Workshop on Parallel and Distributed Real–Time Systems at the 10th IEEE International Parallel Processing Symposium*, April 1996. Honolulu, Hawaii.

[63] MÄKI-TURJA, JUKKA and MIKAEL NOLIN: *Efficient Response–Time Analysis for Tasks with Offsets*. In *10th Proceedings of the IEEE Real–Time Applications Symposium*, Mai 2004. Toronto, Kanada.

[64] NIEHAUS, DOUGLAS, KRITHI RAMAMRITHAM, JOHN A. STANKOVIC, GARY WALLACE, CHARLES C. WEEMS, WAYNE BURLESON and JASON KO: *The Spring Schedul-*

ing Co-Processor: Design, Use, and Performance. In *IEEE Real-Time Systems Symposium*, pages 106–111, 1993.

[65] OREHEK, MARTIN: *Modellierung und effiziente Implementierung eingebetteter Realzeitsysteme*. PhD thesis, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2003.

[66] PFEFFERL, JOHANN: *Laufzeitsystem für Realzeitalgorithmen der Mechatronik*. Number 562 in *Fortschrittsberichte VDI, volume 10*. VDI–Verlag, Düsseldorf, 1998. Dissertation, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München.

[67] POGGI, ENRICO, YEQIONG SONG, ANIS KOUBAA and ZHI WANG: *Matrix-DBP For (m, k)-firm Real-Time Guarantee*. In *RTS'2003*, Paris, France, April 2003.

[68] RAJKUMAR, RAGUNATHAN: *Synchronization in Real–Time Systems A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[69] RAMANATHAN, P.: *Overload Management in Real-Time Control Applications Using (m, k)-Firm Guarantee*. IEEE Transactions on Parallel and Distributed Systems, 10(6), 1999.

[70] RAMANATHAN, PARAMESWARAN and MONCEF HAMDAOUI: *A Dynamic Priority Assignment Technique for Streams with (m, k)-Firm Deadlines*. In *IEEE Transactions on Computers*, volume 44, Dezember. ISSN:0018–9340.

[71] REGEHR, JOHN and JOHN A. STANKOVIC: *HLS: A Framework for Composing Soft Real–Time Schedulers*. In *Proceedings of the 22nd IEEE Real–Time Systems Symposium*, Dezember 2001. London, UK.

[72] REGEHR, JOHN D.: *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. PhD thesis, Faculty of the School of Engineering and Applied Science University of Virginia, 2001.

[73] SHA, L., R. RAJKUMAR and J. P. LEHOCZKY: *Real–time synchronization protocol for multiprocessors*. In *Proceedings of IEEE Real–Time Systems Symposium*, Dezember 1988. Huntsville, Alabama, USA.

[74] SHA, L., R. RAJKUMAR and J. P. LEHOCZKY: *Priority inheritance protocols: An approach to real–time synchronization*. IEEE Transactions on Computers, 1990.

[75] SILLY, MARYLIN: *A dynamic scheduling algorithm for semi-hard real-time environments*. In *Proceedings of the 6th Euromicro Workshop on Real–Time Systems*, pages 130–137, Juni 1994. Odense, Dänemark.

[76] SILLY, MARYLIN: *The EDL Server for Scheduling Periodic and Soft Aperiodic Tasks with Resource Constraints*. In *The International Journal of Time–Critical Computing Systems*, volume 17, pages 87–111. Kluwer Academic Publishers, 1999.

[77] SPURI, MARCO and GIORGIO C. BUTTAZZO: *Efficient Aperiodic Service Under Earliest Deadline Scheduling*. In *Proceedings of the 15th IEEE Real–Time Systems Symposium*, Dezember 1994. San Juan, Puerto Rico.

[78] SPURI, MARCO and GIORGIO C. BUTTAZZO: *Scheduling Aperiodic Tasks in Dynamic Priority Systems*. In *Real-Time Systems*, volume 10, pages 179–219. Kluwer Academic Publishers, 1996.

[79] STANKOVIC, JOHN A., MARCO SPURI, KRITHI RAMAMRITHAM and GIOGIO C. BUTTAZZO: *Deadline Scheduling for Real–Time Systems*. Kluwer Academic Publishers, 1998.

[80] TEMPELMEIER, THEODOR: *Antwortzeitverhalten eines Echtzeit–Rechensystems bei Auslagerung des Betriebssystemkerns auf einen eigenen Prozessor*. PhD thesis, Technische Universität München, Fakultät für Mathematik, 1980.

[81] TINDELL, K.: *Adding Time-Offsets to Schedulability Analysis*. Technical Report, University of York, Computer Science Dept, YCS-94-221, 1992.

[82] TINDELL, K.: *Using Offset Information to Analyse Static Priority Pre–emptively Scheduled Task Sets*. Technical Report, University of York, Computer Science Dept, YCS 182, 1992.

[83] TINDELL, K., ANDY BURNS and ANDY WELLINGS: *An Extendible Approach for Analysing Fixed Priority Hard Real–Time Tasks*. Technical Report, University of York, Computer Science Dept, YCS 189, 1992.

[84] WEST, RICHARD, YUTING ZHANG, KARSTEN SCHWAN and CHRISTIAN POELLABAUER: *Dynamic Window–Constrained Scheduling of Real–Time Streams in Media Servers*. In *IEEE Transactions on Computers*, volume 53. June 2003.

[85] ZLOKAPA, GORAN: *Real–Time Systems: Well–Timed Scheduling and Scheduling with Precedence Constraints*. PhD thesis, University of Massachusetts, Department of Electrical and Computer Engineering, 1993.

Mathematik, Algorithmen, Datenstrukturen

[86] *Statistics of Extemes*. Columbia University Press, New York and London, 1958.

[87] *Extreme Value Theory in Engineering*. Academic Press, Inc., 1988.

[88] CORMEN, THOMAS H., CHARLES E. LEISERSON und RONALD L. RIVEST: *Introduction to Algorithms*. McGRAW–HILL Book Company, 1997.

Standards/Spezifikationen

[89] OBJECT MANAGEMENT GROUP (OMG), http://www.omg.org/cgi-bin/doc?ptc/2004-09-01: *UML^{TM} Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*.

[90] OBJECT MANAGEMENT GROUP (OMG): *UML^{TM} Profile for Schedulability, Performance, and Time Specification*, Februar 2004. http://www.omg.org/cgi-bin/doc?ptc/2004-02-01.